Embedded System Memory Allocator Optimization Using Dynamic Code Generation

YVES LHUILLIER, CEA, LIST, LCE laboratory
DAMIEN COUROUSSÉ, CEA, LIST, LaSTRE laboratory

Embedded systems often exhibit memory organizations far from those of general purpose computing systems. Distributed and private memories, absence of address virtualization are frequent burdens of system and application developer. Thus, dynamic memory allocation in the context of embedded systems is a difficult issue. Moreover, with the increasing amount of parallelism in embedded systems architectures and program, an additional stress is put on dynamic memory allocation, due to the out-of-order nature of parallel applications. This paper presents a flexible memory allocator able to handle complex memory organizations of embedded systems. The memory allocator leverages dynamic code generation so that flexibility is not at the expense of performances. In fact, we show that combining dynamic code generation and runtime adaptation, can give a 56% speedup on memory allocator's allocation and release operations.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; D.3.4 [**Programming Languages**]: Processors—*Translator writing systems and compiler generators*; C.1.4 [**Processor Architectures**]: Parallel Architectures—*Distributed architectures*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: memory allocation, embedded systems, dynamic code generation, runtime monitoring

ACM Reference Format:
Lhuillier, Y. and Couroussé D. 2012. Embedded System Memory Allocator Optimization Using Dynamic Code Generation. ACM Trans. Embedd. Comput. Syst. 7, N, Article ?? (January 2012), 12 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Because of power and area constraints, memory organizations of embedded systems are often rather different than those of general purpose systems. In contrast with conventional memory hierarchy with multiple cache levels, embedded systems often contain explicitly addressable memories (scratchpads) that can be distributed, shared or tightly coupled memories [Banakar et al. 2002; STMicroelectronics and CEA 2010]. Moreover, difficulties associated with explicit memory addressing are often increased by the absence of address virtualization (MMU-less systems). As a consequence, managing fragmented, heterogeneous memories of embedded system is often a strong issue for system and application developers [Banakar et al. 2002; McIlroy et al. 2008].

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 100230 and from the national programmes / funding authorities.
Author's addresses: Y. Lhuillier, CEA, LIST, laboratoire LCE, Saclay F-91191, Gif sur Yvette Cedex, FRANCE; D. Couroussé, CEA, LIST, laboratoire LaSTRE, Batiment CTL, 7 allée de Palestine, 38610 Gières, FRANCE

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

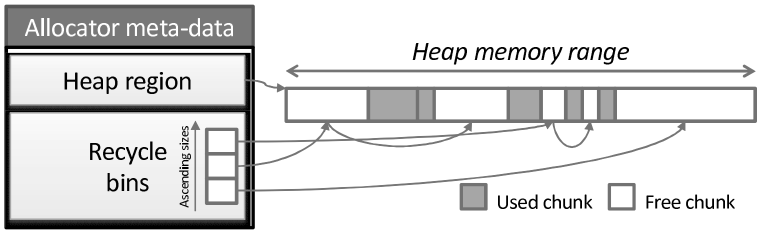

Fig. 1. Schematics of Doug Lea's dlmalloc data structures.

Parallel applications, due to the out-of-order nature of their execution can take advantage of efficient dynamic memory allocation for communications between tasks. Thus, dynamic memory allocation may be critical for performances even in embedded systems where software developers sometimes prefer spending additional efforts to explicitly handle memory management. However, memory allocators, even those especially designed for embedded systems (e.g. *Newlib* [community 2010]), are not adapted to fragmented memories, because they all assume a unique heap mapped in a contiguous address range. Thus, on one hand, memory allocators for embedded systems need to be much more flexible than memory allocators of general purpose or high performance computing systems. On the other hand, embedded systems software cannot afford loosing cycles in flexible, generic and unoptimized runtime functions.

This paper presents an embedded system memory allocator, leveraging dynamic code generation to enhance its ability to handle complex memory organization while keeping, and even increasing its performances. Section 2 presents the general principles of memory allocators, and the proposed implementation of an optimized memory allocator for embedded systems. Section 3 provides details on the tools and the platform used for experimentations. Section 4 discusses the behavior and performances of our proposed memory allocator. Finally, Section 5 and Section 6, present an overview of related works, and a summary of key advantages and limitations of our approach.

2. MEMORY ALLOCATORS FOR EMBEDDED SYSTEMS

2.1. General purpose memory allocators

General purpose memory allocators all share a common ground of software technologies. All GP-memory allocators manage a data structure keeping track of available chunks of memory (free chunks). For all memory allocators, the map of free chunks is always implemented using associative arrays; array of linked lists in [Lea 2000; community 2010] red-black tree, or treaps-based maps in [Evans 2006]. Those associative arrays store free chunks of memory according to their size.

Figure 1 shows the data structures involved in the classic memory allocator of Doug Lea [Lea 2000] (called *dlmalloc*, base of Newlib and GlibC allocators). As in many other general purpose memory allocators, Lea's *dlmalloc* data structures general heap information (base address, size) along with the map of heap's free fragments (called free memory chunks). The heap free chunks are looked up using an associative array implemented using a classic hash table: array of linked-lists. Each linked-list corresponds to a range of free chunk sizes and is sorted according to the size, in ascending order. Following Doug Lea's conventions, the hash array items containing linked-list of free chunks are called bins (or recycle bins). Bins are the place where the memory allocator looks for free chunks, during a `malloc` operation, and store them back, during a `free` operation. The function that associates a bin to an input size is called the bin hash function.

The `malloc` operation scans the recycle bins for a big enough free chunk of memory. The scan operation starts with the bin potentially containing a free memory chunk of size immediately greater or equal to the requested size (using the bin hash function). Then, the scan continues with larger bins if no free memory chunk has been found previously. In each bin, chunks are ordered according to their size, so that the allocator always returns the smallest free chunk immediately greater or equal to requested size. If the elected free chunk is larger than the requested size, the chunk is split and the remainder is re-inserted to the recycle bins.

Another concern of memory allocators is to minimize heap fragmentation. In order to do that, the *dlmalloc* allocator uses a minimal and yet very efficient technique, coalescing chunks with neighboring free chunks during `free` operations. This is possible because allocated and free chunks contain information, in their header, giving the exact address of previous and next heap chunks (with respect to their physical address). This technique minimizes efficiently fragmentation as long as allocated memory is released reasonably frequently (no memory leaks, and few long life memory allocations).

Finally, general purpose memory allocators often assume that memory heaps are large and contiguous. This assumption is absolutely correct in general purpose computing systems where heaps are uniquely associated with processes, and where the memory management units perform logical to physical translations so that process memory spaces are large and flat.

2.2. Embedded system constraints

In embedded systems, computing architectures differ significantly from general purpose ones. Those differences are mainly due to different objectives and constraints; embedded systems cannot afford the power consumption and silicon area assumed by general purpose architectures. As a direct consequence, embedded systems use simpler hardware designs often leveraging a lightly higher effort from the developer.

Memory sub-systems, in embedded systems are often less transparent than in general purpose computer architectures. Caches are removed, in favor of explicitly addressable Tightly Couple Memories (TCM). In order to deliver high performance, those TCMs are distributed amongst the computing resources of the embedded system architectures. Every computing core is associated to a local TCM or scratchpad where it can store frequently used data. Multiple cores accessing shared data can in turn have common scratchpads, normally delivering lower performances because of longer communications and multiple access ports. We observe in many embedded system architectures [STMicroelectronics and CEA 2010] that memory organizations converge toward hierarchic layouts. Nevertheless, unlike in general purpose architectures, hierarchic memories are TCMs (not caches), leaving the developer responsible for explicit data management and placement (software caching).

In a split memory organization, memory allocation is done on a per-scratchpad basis [McIlroy et al. 2008]. Because different scratchpads have different access costs and some scratchpads are even not accessible from all computing resources of the chip, the uniform heap concept of general purpose system is no longer adaptable. Most of the time, one instance of memory allocator is associated to each scratchpad of memory architecture. Since scratchpads have different sizes and different usages (in term of data lifetime, and memory allocation grain), those memory allocators must be flexible to adapt to variable-size scratchpads and specific usages.

2.3. Software architecture of a flexible memory allocator

In order to deal with the constraints of embedded systems, memory allocators need to be able to manage multiple heaps distributed in different memory of different sizes.

A *dlmalloc*-based memory allocator (the one used in our experimentations) needs two modifications to provide this ability.

First, the memory allocator should be multi-instantiable to address multiple heaps. This implies using an object-oriented approach, encapsulating allocator's global variables (heap bounds, recycle bins) in a data structure that can be instantiated multiple times. This data structure has to be passed to each allocator function so that the function knows the particular instance of memory allocator it is dealing with. The additional pointer indirection required to parameterize the allocator instance has hardly any impact on performance since the compiler often manages to keep the allocators' instance address in a dedicated register.

Another important constraint is the ability to manage heaps of different sizes. The original *dlmalloc* is designed for large heaps, thus, its number of recycle bins (hash table width) is large (127 bins). This hash table is completely over-dimensioned for heaps of few kilobytes since a 127-bins structure consumes more than 2KB of memory. The second modification brought to the *dlmalloc* allocator consists in making the free chunk hash table's implementation configurable. This flexibility is provided by having a variable number of recycle bins, and thus a variable bin hash function. Initially, the bin hash function is a statically-compiled built-in function of the memory allocator. To make this function configurable, a bin hash function pointer is added to the allocator's instance data, so that different memory allocators can have different bin hash functions (and thus different bin counts). Multiple variations of the original bin hash functions were written for different heap size (from few kilobytes to several megabytes). The additional indirection of the bin hash function pointer has a cost, slightly higher than instance data indirection, but less than 10 cycles on `malloc` or `free` operations, which costs around 350 cycles.

With the modified software architecture of the memory allocator, it is possible to provide a flexible memory allocator able to manage different heaps of different sizes. The performance overhead of this flexibility is kept minimal, and we will see in next section that performances can even be improved using dynamic code generation.

2.4. Increasing performances using dynamic code generation

To adapt memory allocation to multiple scratchpads with variable sizes and to different runtime behavior of applications, an idea would be to have a large number of bin hash functions and to choose the correct one at runtime. In practice, this number would be too large to address efficiently all possible cases. The idea behind the memory allocator proposed in this article is to start with a reasonable (but suboptimal) hash function, to monitor runtime behavior of the memory allocator, and to regenerate periodically a bin hash function using dynamic code generation.

The periodical regeneration of the bin hash function is done through a call to a function (called `rehash`) which collects information about the memory allocator behavior, and then computes and generates a new optimized bin hash function. In our implementation, the `rehash` function is called periodically by the user, but nothing prevents to make this call hidden behind `malloc` and `free` calls.

The information collected by the `rehash` function comes from the instrumentation of `malloc` and `free` operations. These instrumentations try to capture the activity in each bin (more detail on instrumentation will be given in later sections). Based on this bin activity distribution, the `rehash` function tries to compute a new bin hash function that would rebalance this bin activity. Once this optimization performed, a dynamic code generator effectively writes the machine code for the corresponding bin hash function.

3. EXPERIMENTAL FRAMEWORK

3.1. Code generation framework

3.1.1. `DeGoal`: building fast binary code generators. We have chosen to use `DeGoal`, a tool designed at CEA-LIST to build fast and portable binary code generators. The design around this tool is based on its ancestor named `HPBCG`, initially developed at Versailles University [Charles and Sajjad 2009; Sajjad et al. 2009]: `HPBCG` was designed to generate program instructions at runtime in a very efficient way (the generation cost was about tens of instruction clocks per instruction generated). Code generators were written in an ASM-like language, and transformed to C language by an automatic source-to-source translation. Code generators were architecture-dependent, but the porting of code generators to a new architecture was made easy through architecture description files. `DeGoal` tries to overcome the limitations of `HPBCG` by allowing the development of architecture-independent code generators. It is composed of tools for the parsing of the architecture description files, source-to-source tools for the translation of code generators descriptions into standard C source files, and a set of architecture description files for various architectures. Architecture-dependent features can be introduced both statically at compile time, and dynamically when the code generator effectively produces the program instructions.

`DeGoal` allows to temparate in time and in architecture space the phases of code generation and of code execution. The ability to separate along time code generation and code execution is fundamental in the context of memory allocation: our aim is to increase performance by using a memory allocator optimized according to the runtime context, but at the same time we want to keep a control over the time at which the code generation is performed, so that we will not impact the performance of critical application components. The spatial separation of code generation and code execution makes possible to run the code generator on one processor architecture, while the generated code is executed on a different processor architecture. This makes of `DeGoal` a very interesting candidate for many-core platforms with heterogeneous processor architectures. The target platform we have used, presented in section 3.2, is currently composed of two kinds of processor architectures. We hence keep a very high flexibility for the allocation of the code generation of the platform's hardware resources.

3.1.2. Kernels and compilettes. The two categories of software components around which our code generation technique is built are called *kernels* and *compilettes*:

Kernel. A kernel is a small portion of code, which is part of a larger application, and which is most of the time under strong performance constraints; our technique focuses on the optimization at runtime of these smalls parts of a larger application in order to improve kernel's performance. In the context of this paper, good performance is understood as low execution time and/or low memory footprint.

Compilette. A code generator. A compilette is designed to generate the code of kernels at runtime. It can be understood as a small compiler that is executed at application's runtime. We use the term *compilette* to underline the fact that due to performance constraints this small runtime compiler does not embed all the optimization techniques usually carried out by a static compiler. The binary code of a compilette is generated during the static compilation along with the rest of the application.

3.1.3. Workflow of code generation. The building of an application using `DeGoal` is detailed in figure 2 and explained below:

Writing the source code (application development time). This task is handled by the application developer, and/or by high-level tools if any. Using `DeGoal` the source code

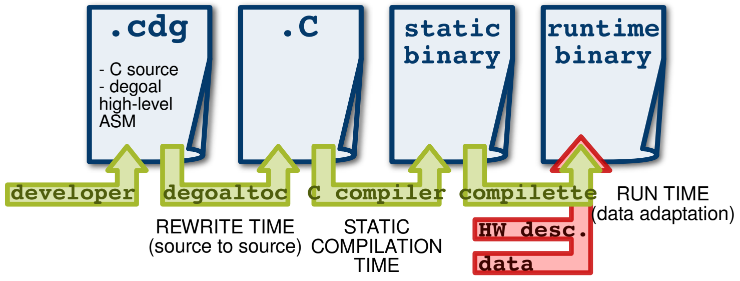

Fig. 2. `DeGoal` workflow: from the writing of application's source code to the execution of a kernel generated at runtime

of compilettes is written in specialized `.cdg.` files, while the rest of the application software components are written in C source code. `.cdg` files combine standard C language and a dedicated high-level ASM language, which allows for the description of code generators in an architecture-independent way.

Generation of C source files (rewrite time). This step consists in a source-to-source transformation: the `.cdg` source files mixing high-level ASM instructions and standard C are translated into standard `C` source files by `degoaltoc`, which is one of `DeGoal` tools. At this phase architecture-dependent features can be introduced in the C source files generated, for example register allocation and vectorization support.

Compilation of the application (static compilation time). The source code of the application now consists in a set of standard C source files, including the source code of the compilettes. The binary code of the application is produced by a standard C compiler. This step is the same as in the development of a standard C application.

Generation of kernel's binary code (runtime). At runtime, the compilette generates the binary code of the kernel(s) to optimize. This task can be executed on a processor that is different of the processor that will later run the kernel. Furthermore, the compilette's processor and the kernel's one do not necessarily need to have the same architecture. A compilette can be run several times, for example as soon as the kernel needs to be regenerated for new data to process. We have detailed on figure 2 two particular inputs of the compilette: data and hardware description. The originality of our approach indeed relies in the generation of a binary code optimized for a particular set of application data. At the same time, the code generation is able to introduce hardware-specific features, for example specialized instructions.

Kernel execution (runtime). The program memory buffer generated by the compilette is run on the target processor (not shown in figure 2).

3.2. Target architecture

3.2.1. Platform 2012. In this article, we target a STMicroelectronics embedded platform called Platform 2012 [STMicroelectronics and CEA 2010] (P2012). It is a large scale, scalable multi-core fabric, under development by STMicroelectronics and CEA. This many-core architecture is modular through its cluster-based structure. The fabric is composed of multiple clusters connected through an asynchronous network-on-chip allowing each cluster to have its own voltage and frequency domain. Each P2012 cluster aggregates a multi-core computing engine, called ENCore. The ENCore cluster, shown in Figure 3 includes a number of processing elements (PEs) varying from 1 to 16. Each PE is built with a configurable and extensible processor from STMicroelectronics called STxP70-4, described in section 3.2.2.

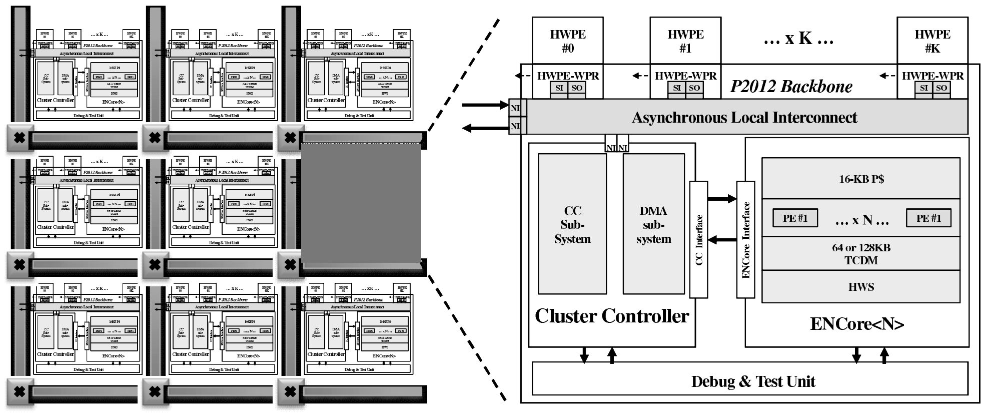

Fig. 3. Architecture overview of the P2012 computing fabric.

The P2012 memory organization follows the hierarchic architecture of the fabric itself. The fabric itself is associated to an on-chip DDR3 memory called the external memory. The ENCore cluster comes with a 128KB multi-bank memory shared by the 16 cores of the cluster. Finally each computing core has local cache memories for instructions and data (some declination of the P2012 architecture replace the data cache by a 32Kb tightly coupled memory).

A lightweight runtime system, called the “P2012 simple runtime” has been developed for the P2012 platform to bring to developers necessary functions to load applications, manage threads, communications and synchronizations on the platform. The P2012 runtime provides a *dlmalloc*-based memory allocator (the one used in this article). Experiments of this article focus on allocation in the cluster shared memory.

3.2.2. Processing core: STxP70. The STxP70-4 processor is a 32-bit RISC core from STMicroelectronics. It comes with a variable-length instruction encoding and a dual VLIW architecture allowing two instructions to be issued and executed at each cycle. The processor also provides guarded execution, with 4 guards registers, which allows some optimizations that our code generator uses extensively.

The P2012 SDK is delivered with a full toolchain for compiling, debugging, profiling and simulation in functional and cycle-accurate modes. In the context of this paper, our experimental results are based on the cycle-accurate simulator of the STxP70 core.

3.3. Dynamic code generation in our memory allocator

The bin hash function associates a bin index to an input size. The bin index allows accessing an array item holding a linked-list of free memory chunks. In our `malloc` implementation bin indices correspond to descending inputs size. For example a bin hash function of 3 bins covering ranges from 0 bytes to 4 kilobytes could be described has `{[0x100:0x1000) => 0, [0x10:0x100) => 1, [0x0:0x10) => 2}`. Thus, the bin hash function generator takes a list of input size ranges, along with a list of output bin indices, and dynamically generates the code of the corresponding bin hash function using a recursive algorithm described below.

3.3.1. Code generation of the hash function. The hash function code generator uses a recursive dichotomy algorithm to build a binary comparison tree and to encode the corresponding machine code sequence. The pseudo-code of a simplified version of the hash function code generator is given in figure 4.

```
def gen_bin_hash( out_code, in_bounds ):
    if len( in_bounds ) == 1:
        out_code.gen_return( in_bounds.index )
    jump = out_code.gen_compare_and_jump( in_bounds.middle )
    gen_bin_hash( out_code, in_bounds.left )
    jump.set_target( out_code )
    gen_bin_hash( out_code, in_bounds.right )
```

Fig. 4. Pseudo-code of a simplified version of the hash function code generator

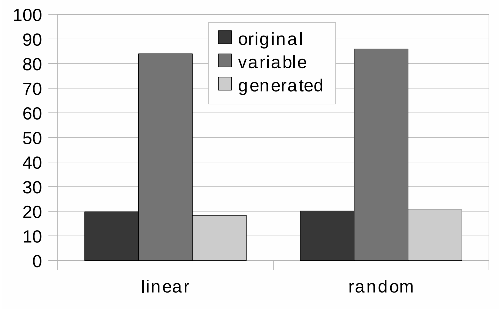

Fig. 5. Average execution time (in cycles) of all hash functions on two different datasets

The algorithms recursively divides the input bounds array to generate the necessary `return` and `compare_and_jump` machine instructions. The actual code generator is an optimized version of this pseudo-code that: 1/ leverages the VLIW architecture to produce bundles of two parallel instructions and 2/ performs software pipelining on comparisons, taking advantages of the 4 guard registers available in the STxP70-4. The code generator produce a 127 bins hash function in about 40.000 cycles, which corresponds to 37.5 host instructions per generated instructions.

3.3.2. Performance of generated hash functions. Figure 5 shows the average execution time of three hash functions: the original hash function (`original`), a software configurable hash function (`variable`) and our dynamically generated hash function (`generated`). The `original` and `variable` functions were compiled using the maximum optimization level of the compiler. Measurements were performed on a two large sets of input values: a linear scan of all the possible input values (`linear`) and a random sequence of input values (`random`).

We observe on both input data sets that our generated function performs as well as the original function. The `variable` hash function is a configurable function which computes the hash index using a dichotomy algorithm with a memory-stored array of input size ranges. This version of the hash function is up to 4 times slower, which confirm that conventional software approach fails to balance flexibility and performance in that case. Finally, we observe that all functions are hardly insensitive to the input dataset, showing that the STxP70 static branch predictor, doesn't request further optimizations from our code generator.

4. MEMORY ALLOCATOR RUNTIME PERFORMANCES

We have shown, in previous sections, how the memory allocator was modified, using dynamic code generation, to allow runtime (re)configuration of its free chunk associative array. The following sections describe how this configurability is used, in combination

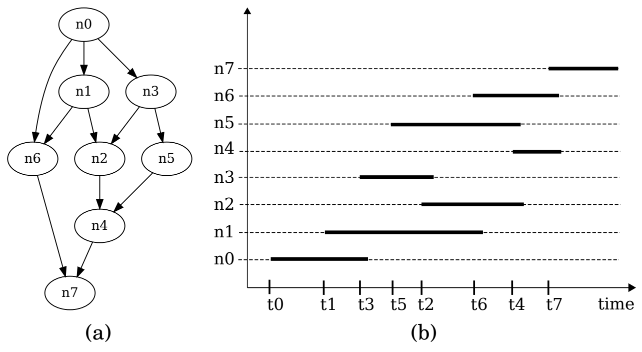

Fig. 6. Sample allocation graph and timeline of allocated buffers for one possible execution

with runtime monitoring, to dynamically adapt it to the effective behavior of memory allocations.

4.1. Experimental setup for measuring memory allocation

In order to evaluate our memory allocator on a large set of memory allocation schemes, we developed a synthetic random allocation benchmark generator. The generator uses a graph allocation model that is able to reproduce the runtime behavior of parallel execution model based on task-level parallelism [Frigo et al. 1998; Ojail et al. 2011; Apple Inc 2010] (TLP). In these execution models, tasks are dynamically spawned in a fork-join (diamond) fashion, and communication between tasks is done through shared buffers. The shared buffers may be statically allocated at unique memory locations, but dynamic allocation is often mandatory to limit the memory consumption.

Figure 6 shows the principle of the graph allocation model. Each graph vertex corresponds to a task that performs exactly one buffer allocation and potentially several buffer releases. Graph edges correspond to transitions from one task to another. A buffer allocated by one task is only released when all the “next” tasks are started. A task buffer thus represents the communication between the task (producer) and the consumers. The timeline on figure 6(b) illustrates a possible execution of the graph model on figure 6(a), where on the X-axis tn corresponds to the starting time on the task n, and the bold black lines on the figure represent memory buffer timelines.

Using this graph model, we were able to model existing applications along with a large set of randomly generated graphs (with random allocation size for each vertex). It allowed us to stress our memory allocator and to investigate monitoring and reconfiguration strategies on a large set of memory allocation patterns.

4.2. Exploiting runtime effective behavior of memory allocation

We saw, in section that in Section 2.4 that our *dlmalloc*-based memory allocator provides a `rehash` function that allows optimizing the memory allocator, at runtime, based on feedback of `malloc` and `free` instrumentation. Next sections discuss the details of the instrumentation and runtime optimizations.

4.3. Adaptation algorithms

Memory allocator instrumentation. In order to be able to reconfigure the memory allocator, and especially the size ranges associated with each bin, we need to define a cost function (one per bin) that reflects the load of each bin. Then using this metric, we need to adapt the hash function so the overall performance of the memory allocator is improved. The idea behind configuring the hash function is that the more balanced the

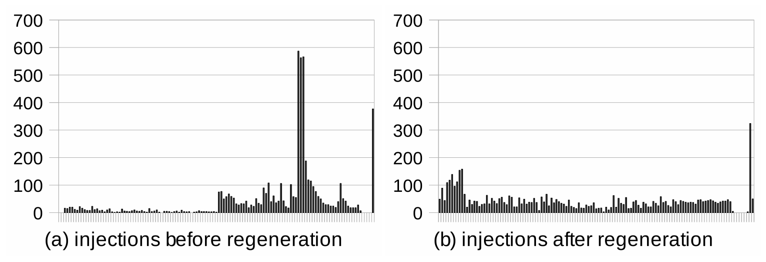

Fig. 7. Bin distribution (from lower to larger sizes) of free chunks injections

hash table is, the less operations will be performed on each bin: a `malloc` operation will find the best bin with a minimal linked-list traversal and a `free` operation will perform sorted insertion of the free memory chunk in short linked-list. Note that in the initial memory allocator, linked-list manipulations of the `free` and `malloc` operations account for 66% of the whole operations.

Several instrumentation techniques were tested to determine the cost function that better reflects the activity of each bins. A good activity metric of each bins is necessary so that the feedback is as effective as possible. We implemented counters, for each bins, that counts: 1/ free memory chunk extraction during `malloc` operations, 2/memory chunk insertion during `free` operations, and 3/ linked-list iterations for both `free` and `malloc` operations. The regeneration of the hash function succeeds in (re)balancing all possible counter combination. Nevertheless, only the `rehash` function based on the "free chunk insertions" counter effectively translates into improved performance. This result comes from the fact that monitoring free chunk insertion is the best way to measure in which bins free chunks effectively are. Finally, counters monitoring link traversals have the potential to be more precise, but their runtime overhead always degrades performances.

Bin load balancing. The algorithm responsible for load balancing the hash table is implemented in the `rehash` function. This algorithm follows a simple linear redistribution of bins' intervals according to the cost function (free chunk insertion) associated to each bins. Figure 7 shows the distribution of free chunks insertions in each bins.

Figure 7(a), shows the distribution before regeneration of the bin hash function (call to `rehash`) and Figure 7(b) shows the equivalent bin distribution after a call to `rehash`. Both bin distributions are observed after the execution of the same synthetic benchmark (see section 4.1) generating 4096 `malloc` calls (and 4096 `free` calls). On this specific benchmark this load balancing operation ends in a 1% performance improvement. We will see in next section that a single call to the `rehash` function can sometimes degrade performances, but performances are hardly always improved after the second call (always after the third).

4.4. Runtime adaptation of the memory allocator

To measure the performance improvment provided by the optimized memory allocator, we used the synthetic allocation benchmarks (section 4.1). The synthetic benchmarks are run several times interleaved with optimization steps (calls to `rehash`). Average execution times of `malloc+free` are collected at each iteration (before `rehash` calls). To investigate the memory allocator's sensitivity to different allocation patterns, we randomy generated 1.000 different synthetic benchmarks and measured worst cases, best cases and average performance. Figure 8 shows the results of the experimentations. The X-axis follows the successive optimization steps (iterations). The 0 value corresponds to the initial state, before any call to `rehash`. The Y-axis reveals the

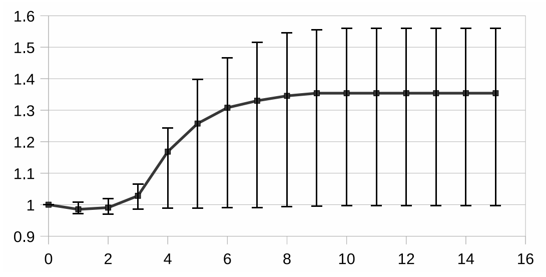

Fig. 8. Average speedup of `malloc+free` operations after successive iterations of the `rehash` optimization. Speedup are given according to the initial execution time (before the first call to `rehash`).

speedup provided by each call to `rehash` on average `malloc+free` execution time. Initially,(`malloc+free` operations have an average execution time of about 680 cycles). Results show the dynamic adaptation of the memory allocator to the runtime allocation behavior of the benchmark. The memory allocator is hardly always more efficient, than initially, after 3 calls to the `rehash` operations. After 9 calls, an average speedup of 35% and a maximum speedup of 56% is observed on `malloc` and `free` calls.

5. RELATED WORK

There is an extensive amount of literature about dynamic compilation, mainly related to Just-In-Time compilers (JITs), which are mainly used to accelerate interpreted languages [Aycock 2003], but also native machine code [Bala et al. 2000]. JITs dynamically select the parts of the program to optimize without a priori knowledge on the input code. This usually means a large footprint and a significant performance overhead. in order to target embedded systems, some research works have tried to tackle these limitations: memory footprint can be reduced to a few hundreds of KB [Gal et al. 2006], but the binary code produced is often of lower quality because of the smaller amount of optimizing intelligence embedded in the JIT compiler [Shaylor 2002].

On the contrary our approach is based on the optimization of code regions with performance constraints that are *a priori* known. It then becomes possible to design an *ad hoc* small specialized compiler, embedding low level optimization techniques and able to exploit specialized instructions of the target processor. The compromise is not anymore on the quality of the code produced, but on the fact that only selected parts of the application will be generated at runtime.

The approach chosen in `DeGoal` is similar to partial evaluation techniques [Consel and Noël 1996], where the aim is to exploit runtime context information to produce code of better quality as compared to the output of a static compiler. In partial evaluation the aim is to generate dynamically the binary code in a minimal number of operations: the main techniques used are: selecting code templates, filling pre-compiled binary code with runtime values and relocating jump addresses. Partial evaluation however is not able to exploit specialized instructions of the target processor, which on the contrary `DeGoal` is able to do at the expense of a lightly higher code generation cost [Brifault and Charles 2004; Sajjad et al. 2009].

To our knowledge, no work on memory allocator optimization leverages dynamic code generation nor JIT compilation. However, memory allocator optimizations are mainly based on improvement of free memory chunk storage [Berger et al. 2002; Evans 2006; McIlroy et al. 2008]. Thus, our memory allocator can be thought as the combination of free chunk storage optimization and runtime optimization with code generation.

6. SUMMARY

This article presents a new memory allocator design optimized for embedded systems. This memory allocator leverages runtime monitoring and dynamic code generation to improve (de)allocation functions. The user, by periodically calling an optimization function (`rehash` function), can improve performances of `malloc` and `free` functions by up to 56%, after 9 calls to the `rehash` function.

Future studies sill focus on improving our solution to increase the convergence speed of the `rehash` function. Also, we plan to release the user from explicitly calling this function and hide the `rehash` call behind `malloc` and `free` calls. And finally, we will investigate changing the number of bins at runtime.

REFERENCES

APPLE INC. 2010. Grand Central Dispatch (GCD) Reference. http://developer.apple.com.

AYCOCK, J. 2003. A brief history of just-in-time. *ACM Computing Surveys 35*, 97–113.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. PLDI '00. ACM, New York, NY, USA, 1–12.

BANAKAR, R., STEINKE, S., SIK LEE, B., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *In Tenth International Symposium on Hardware / Software Codesign (CODES), Estes Park*. ACM, 73–78.

BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. 2002. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '02. ACM, New York, NY, USA, 1–12.

BRIFAULT, K. AND CHARLES, H.-P. 2004. Efficient data driven run-time code generation. In *proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*. LCR '04. ACM, New York, NY, USA, 1–7.

CHARLES, H. AND SAJJAD, K. 2009. HPBCG High Performance Binary Code Generator. http://code.google.com/p/hpbcg/.

COMMUNITY, N. 2010. A memory allocator. http://sourceware.org/newlib/.

CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to c. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*. 145–156.

EVANS, J. 2006. A scalable concurrent malloc(3) implementation for freebsd. In *BSDCan conference*. Ottawa, Ontario, Canada.

FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. 1998. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not. 33*, 212–223.

GAL, A., PROBST, C. W., AND FRANZ, M. 2006. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*. VEE '06. ACM, New York, NY, USA, 144–153.

LEA, D. 2000. A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

MCILROY, R., DICKMAN, P., AND SVENTEK, J. 2008. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the 7th international symposium on Memory management*. ISMM '08. ACM, New York, NY, USA, 31–40.

OJAIL, M., DAVID, R., BEN CHEHIDA, K., LHUILLIER, Y., AND BENINI, L. 2011. Synchronous reactive fine grain tasks management for homogeneous many-core architectures. In *proceedings of the 2parms Workshop, ARCS'11.* Italy.

SAJJAD, K., TRAN, S.-M., BARTHOU, D., CHARLES, H.-P., AND PREDA, M. 2009. A global approach for MPEG-4 AVC encoder optimization. In *14th Workshop on Compilers for Parallel Computing*. Zurich, Switzerland.

SHAYLOR, N. 2002. A just-in-time compiler for memory-constrained low-power devices. In *Proceedings of the 2nd Java&#153; Virtual Machine Research and Technology Symposium*. USENIX Association, Berkeley, CA, USA, 119–126.

STMICROELECTRONICS AND CEA. 2010. Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. In *CMC Research Workshop on STMicroelectronics Platform 2012*. Toronto: s.n.