

Reducing Internal Fragmentation in Segregated Free Lists using Genetic Algorithms

Christian Del Rosso
Nokia Research Center
Itämerenkatu 11-13, 00180
Helsinki, Finland

christian.del-rosso@nokia.com

ABSTRACT

In this paper we present an approach for improving memory efficiency using genetic algorithms. More precisely, we improve the internal memory fragmentation by finding the optimal configuration of a segregated free lists data structure. We have used trace instrumentation to generate the workload of memory allocations and deallocations from significant scenarios. The genetic algorithm used the workload as input to generate the optimal configuration among the huge number of potential solutions by evolving an initial population (a set of potential solutions). In practice, memory configurations are created on the empirical evidence based on the fact that the system works. However, a more scientific and rational approach is possible by using genetic algorithms. The approach we have used was fast and effective in providing the configuration parameters for the segregated free lists. The result is based on the use of heuristics and provides an excellent choice when a brute force approach is not feasible. Moreover, the use of genetic algorithms shows that the software engineering discipline can benefit from different research areas where complexity, adaptation and evolution are involved.

Categories and Subject Descriptors: D.2.8 [Software Engineering]:Metrics-Performance measures; D.4.8[Operating Systems]: Performance-Measurements; D.2.5 [Software Engineering]:Testing and Debugging-Tracing;

General Terms: Algorithms, Performance, Design

Keywords: dynamic memory, optimization, performance, genetic algorithms

1. INTRODUCTION

Software applications use dynamic memory known as heap, when memory is needed at run-time. The dynamic memory management system is the infrastructure in the operating system that provides the requested memory to the applications. Whenever a *C malloc()* or a *C free()* function call is called in a program, we access the operating system's dy-

namc memory management system to allocate or free dynamic memory.

The role of the dynamic memory management system is to handle the requests for memory blocks and at the same time to efficiently use the memory. The study of dynamic memory optimization does not revolve solely on the decision of the best algorithm to be used. Several parameters influence dynamic memory usage and to tackle the problem we need to understand and study the topic from a wider perspective. For example, patterns of data allocations and heap layouts affect memory usage and system performance. The correct configuration of the heap parameters has a big impact on the dynamic memory management system and subsequently, on the performance of the entire system.

In this paper we will focus on the study of the segregated free lists layout, addressing in particular the topic of how to configure the free lists to satisfy our system's needs. Segregated free lists are preallocated chunks of memory with a predetermined size and are divided into bins. We have used trace instrumentation of system allocations and deallocations by running significant scenarios in a Nokia handset. The traces of allocations and deallocations were stored in log text files and used for the analysis. The method uses heuristics based on genetic algorithms to find an optimal configuration for the segregated free lists. The optimal configuration implies finding the number of memory chunks and the size of each bin to satisfy the size of allocations and deallocations of the applications. The search space for the solution is large and a brute force approach is not feasible. Therefore, an approach based on genetic algorithms is used to find an optimal solution.

Segregated free list is a heap layout with several advantages such as no external memory fragmentation and good performance speed. However, blocks configuration and bins size is an important issues to be addressed if we want to improve memory usage. The approach we have used presents a way to find an optimal segregated free lists configuration given the assumption that significant scenarios and data are used.

2. PROBLEM STATEMENT

Memory efficiency is related to the study of fragmentation. Fragmentation is the inability to allocate contiguous blocks of memory because the memory is divided into areas of used and not used blocks, see [7], [13]. Because of fragmentation the requests could not be served even though enough memory is available in the heap. The fact that the memory is divided into small areas of free blocks prevents allocations of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WISER'06, May 20, 2006, Shanghai, China.

Copyright 2006 ACM 1-59593-085-X/06/0005 ...\$5.00.

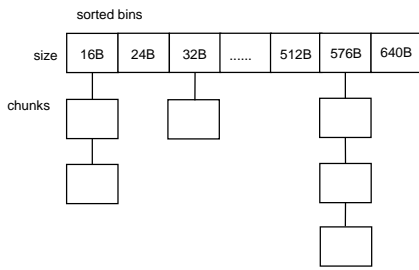


Figure 1: Pool System

large contiguous blocks. Allocators that compact the memory and garbage collector allocators can help in lowering fragmentation. However, such allocators have penalties in performance [4], [12] and allocators with explicit reference to main memory are still commonly used in the development of real-time and embedded systems. The previous definition is commonly defined as external fragmentation.

In the literature, another form of fragmentation is defined and it is called internal fragmentation. Internal fragmentation is the waste of memory when a memory request from the application is served by the dynamic memory management system with a block larger than is necessary, see [11]. In some allocators, a large enough block is allocated to serve the request but the remaining part is not split and contributes to lowering memory efficiency and to increasing the internal fragmentation. However, if the dynamic memory management system splits the blocks, the remaining free part of the block can be smaller than the minimum block size allowed in the system. In such cases two alternatives exist: either the whole memory block is allocated or the remaining part is left as free memory. In the first case internal fragmentation will increase, and in the second case external fragmentation will increase. In addition, the CPU accesses memory blocks aligned with its word size and all the blocks in the dynamic memory are word aligned.

Particularly in real-time embedded systems, when the applications allocate small-size blocks and time constraints are important, segregated free lists are used. Segregated free lists are lists of preallocated free memory with fixed size blocks determined statically at compile time. A set of bins of different size are defined and every bin contains a certain number of chunks of memory, see 1.

Segregated free-lists do not have the problem of external fragmentation, but rather one of internal fragmentation when a small block is allocated into larger blocks. In segregated free lists, the requests are served using a bin of the same size. However, if the exact size does not exist or that bin does not have free chunks of memory, the request is served by the next larger bin. When this occur, we have memory waste and an increase in internal fragmentation.

The approach currently used for the configuration of segregated free lists is empirical. A configuration is tested under a certain workload and it is compared with other configurations. If the configuration works well enough for the system considered, then it is used. However, this is not necessarily the right approach since problems arise when new features with different patterns of memory allocations are included.

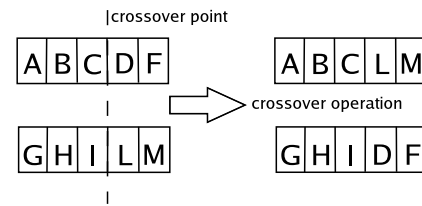


Figure 2: crossover

3. GENETIC ALGORITHMS (GA)

Genetic algorithms were pioneered by John Holland in the 1960s [6, 5]. However, the idea was only accepted and reached a wider audience during the 80s. Genetic algorithms provide a heuristic approach to function optimization problems. The analogy on which genetic algorithms are based is biological evolution. Genetic algorithms have the concepts of fitness, crossover, mutation, populations, and genes.

An encoded parameter is a gene, a chromosome is the string produced by concatenating all the encoded parameters and each chromosome is an individual and member of a population. The approach to problem solving is done by encoding the solution in a fixed length string.

The search for the optimal solution is iterative and includes the evolution of the population by mating, mutating and using a fitness function, continuing the analogy with biological evolution. The iteration ends when some termination conditions are reached. Below a pseudo code for a generic genetic algorithm:

```

-Initialize the population P
-Repeat for some length of time:
  -Create an empty population, P'
  -Repeat until P' is full:
    -Select two individuals from P based on
      some fitness criterion
  -Optionally mate, and replace with the offspring
  -Optionally mutate the individuals
  -Add the two individuals to P'
  -Let P now be equal to P'

```

The initial step is the generation of the population. Then, individuals are chosen according to a fitness criterion. The fitness function is problem-dependent and the ability to properly encode the problem is fundamental. Mating, also called cross-over, fills the role played by sexual reproduction.

When two strings are crossed, the result is two new strings, each containing part of the genetic material of the parents. Mutation is applied by randomly flipping bits in the string.

Generic algorithms are able to solve optimization problems, which are not feasible using a brute force search approach, by using heuristics and performing massively parallel searches. The members of the populations are potential solutions and even strings with completely different letters may have equal fitness. The implications are that crossing such pairs could result in drastic improvements in the average fitness. In addition, many such partial solutions can be searched for simultaneously, an *implicit parallelism* property, as referred to by Holland.

3.1 GA in Software Engineering

Genetic algorithms have already been used in several contexts by the scientific community and engineers to solve

practical optimization problems where the space of the solutions was too broad to be explored systematically.

In software engineering, the use of heuristics and genetic algorithms has been applied to tackle different parts of the software lifecycle. In a paper proposed to reformulate software engineering as a search problem, the authors have described several instances where the use of such techniques and heuristics could be applied [2]. Genetic algorithms have been used for clustering the software source code in modules [9]. The use of genetic algorithms has been experimented for automatic software test data generation [1]. The concepts of adaptation and evolution have been proposed for CPU load optimization. An approach was proposed in an article for self-tuning a system using genetic algorithms, [3]. Recently, with an announce in the Linux kernel mailing list, a patch has been submitted to tune the Linux operating system using genetic algorithms [10].

4. GA FOR DYNAMIC MEMORY CONFIGURATION

In a function optimization problem, the problem is coded as a function to be maximized or minimized. The problem depends on a set of parameters and the scope of the optimization function is to find the parameters that give the optimal solution in the solutions' space.

The solution would be easily found if the plot of the function could be done. However, in real-world scenarios, the function may be parameterized by thousands of values making a brute force search infeasible. Finding a good enough solution using heuristics in a reasonable amount of time means tremendous gains and advantages in terms of money and resources in an otherwise hard to solve problem.

The fitness of individuals is evaluated in a problem-dependent way. In our case, individuals are considered to be more fit if they produce less internal memory fragmentation given the assigned workload.

4.1 Optimizing Internal Fragmentation in Segregated Free Lists

When the system uses small block allocations and performance is one of the requirements, segregated free lists represent an excellent choice [11, 8]. As introduced in the problem statement section 2, using preallocated size free lists means to establish a priori the configuration of the data structure. At compile time a decision must be made that cannot be changed during run-time execution of the software. To reconfigure the segregated free lists at run-time may be possible, but the cost to performance may not be in practice feasible in some circumstances, for example when dealing with real-time systems. Therefore, in our case study we will consider only a static preallocated free lists data structure.

In this context, the problem we are trying to address is to find the optimal configuration for a segregated free lists data structure given the minimum and the maximum value of the bins, the processor word size and the data workload as input. A set of bins can easily be defined when the data structures that will be allocated by the application are known a priori. For example, the application uses 80 byte allocations to allocate a vector of 80 elements. In addition, when the application allocation patterns are unknown or unpredictable, bins can be created by spacing the bins using the processor word size.

However, it is not easy to determine the subset of bins that optimizes the function we chose, namely the reduction of internal fragmentation, especially when the distance between the minimum and the maximum bin values is high and the size of the allocations vary considerably in that range.

The number of bins between a minimum and a maximum value for the segregated free lists, spaced by the processor word size can be calculated by the following equation:

$$\frac{\text{maximum} - \text{minimum}}{\text{word}} + 1 = n \quad (1)$$

In addition, given a set of elements, determining the number of subsets of unique elements corresponds to calculate what it is called in combinatorial mathematics, the combination of the set. The order of the elements in the combination is not important but every combination contains a unique set of elements. The combination of a subset of k elements given a set of n elements is the binomial coefficient calculated using the following formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad (2)$$

The number of the subsets constitutes the solutions space and this number can easily become very large.

4.2 The GA Fitness Function

The fitness function, our optimization function, must reduce internal fragmentation and provide an optimal configuration for the segregated free lists for the selected workload. The parameters of the optimization function are the size of the bins and the amount of chunks for each bin. Additionally, we provide the minimum and the maximum value for bin size and processor word size in bytes. The word size number ensures that we work only according to the architecture of the targeted hardware, for example, for ARM7 the word size is 4 bytes.

Encoding of the solution in the GA format means that the parameters must be encoded in a fixed size string which the genetic algorithm will evolve toward the solution. The items below present the encoding of our problem for a genetic algorithm:

- Gene A characteristic of a chromosome - bin
- Chromosome A set of genes - memory configuration
- Population A generation of chromosomes

In addition, the crossover rate, the rate at which two chromosome are mixed to produce new chromosomes; and the mutation rate, the rate at which a gene is changed to a random value, are parameters of our algorithm.

5. EXPERIMENT

We generated the workload using a scenario-driven approach with trace instrumentation. While mathematical functions have been used to simulate the workload in several studies, Wilson et al. [14] have shown that real program behavior does not follow the independence of the data of the distributions.

The scenarios were defined using the user visible features of the system assessed, for example: browsing the web, sending 1 SMS. There is not a magic rule on how to choose

significant scenarios. However, the essential guideline was to choose scenarios which had demanding memory requirements. Scenario analysis and data analysis of several scenarios increased the understanding of the system studied.

During our experiments, we examined internal fragmentation and analyzed memory usage using a simulator environment. The simulation environment was designed and used to extract the metrics needed for the configuration of the segregated free lists. Some of the metrics extracted were: internal fragmentation, chunks used in each bin, number of overflows for each bin, graphs of memory usage by size and number of requests by size. The information above was necessary for understanding the characteristics of the patterns of memory allocations in the various scenarios.

In this case study, we used the data workload generated by the scenario *browsing the web*. Data were extracted by running the selected scenario and with the phone turned off as initial condition. A more extensive data collection and analysis would have been possible, however our scope, in this context, was to understand and test the approach with GA.

6. RESULTS

The *browsing the web* scenario generated a text log file with 238,348 memory allocations and deallocations. We decided to define the segregated free lists with bins between 16 and 536 bytes, to use a word size of 4 bytes (like the ARM7 processor) and to use a subset of 13 bins. Using formula 2, extracting n with the formula 1 and with $k=13$, the possible combinations of 131 elements in set of 13 unique elements are 507,698,714.

The genetic algorithm, written in C, found the optimal configuration in 1 min and 16 seconds using a Pentium 4 desktop computer. The GA solution contained the following bins:

[16], [24], [36], [40], [52], [64], [92], [148], [168], [188], [220], [296], [536]

The number of chunks returned did not produce overflows and reached an utilization of 100%. The highest internal fragmentation was 21,417 bytes and the average internal fragmentation was 13,688.98 bytes.

Using the simulation environment we systematically explored the remaining potential solutions. We found that the GA proposed the optimal solution to our problem. In addition, search for the potential solutions without GA took considerable more time.

7. CONCLUSION

Being able to find the optimal configuration for preallocated free lists is an important issue if we want to limit internal dynamic memory fragmentation in this data structure. The study of the workload of the system and an empirical configuration approach, "if it works, it is fine" has been used in the industrial realm. However, a more scientific approach with quantified evidence is necessary.

Fortunately, the optimization function has a clear and well defined form and a structured output: the size of the bins and the number of chunks in each bin. Given the large space of the solutions, we have used a heuristic approach based on genetic algorithms. The experiment has proved in practice that, using this approach, we have been able to

find an optimal configuration for the specific scenario in a limited time.

The use of heuristics based approach is useful when the solutions space is large and when it is not possible and reasonable to explore them all. As software systems increment their complexities, the use of complex adaptive system theories make sense. In the future, we will see more and more practical examples of applications and optimization problems benefitting from complex adaptive systems research. The convergence of complex adaptive systems and software engineers is increasingly creating new and interesting research topics and applications to be explored.

8. REFERENCES

- [1] M. Christoph, C., G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Transaction on Software Engineering*, 27(12):1085–1110, December 2001.
- [2] J. Clarke, J. Dolado, J., M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings-Software*, 150(3):161–175, June 2003.
- [3] G. Feitelson, D. and M. Naaman. Self-tuning systems. *IEEE Software*, 16(2):52–60, Mar/Apr 1999.
- [4] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for java. *Software: Practice and Experience*, 30(3):199 – 232, March 2000.
- [5] E. Goldberg, D. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1989.
- [6] H. Holland, J. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.
- [7] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? *Proceedings of the 1st international symposium on Memory Management*, 34(3), October 1998.
- [8] D. Lea. A memory allocator. <http://gee.cs.oswego.edu/dl/html/malloc.html>.
- [9] B. Mitchell, M. Traverso, and S. Mancoridis. An architecture for distributing the computation of software clustering algorithms. *IEEE/IFIP Working Conference on Software Architecture, WICSA*, 16(2):52–60, August 2001.
- [10] J. Moilanen. Linux: Tuning the kernel with a genetic algorithm. <http://kerneltrap.org/node/4493>, Jan 2005.
- [11] J. Noble and C. Weir. *Small Memory Software: Patterns for system with limited memory*. Addison Wesley, 2001.
- [12] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.
- [13] J. E. Shore. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the ACM*, 18(8):433 – 440, August 1975.
- [14] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Proc. Int. Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.