A TOOL FOR ANALYSING DYNAMIC MEMORY ALLOCATORS

Petr Muller
Master Degree Programme (2), FIT BUT
E-mail: xmulle13@stud.fit.vutbr.cz

Supervised by: Tomáš Vojnar
E-mail: vojnar@fit.vutbr.cz

ABSTRACT

This paper describes a tool for the monitoring and analysis of the most important dynamic memory allocator metrics under various use cases. At first the problem which this tool aims to solve is introduced, followed by the tool design description. The current status and a sample measurement is also briefly discussed

1 INTRODUCTION

Dynamic memory allocator is a fundamental part of an operating system, providing runtime memory allocation to programs. Almost every program needs this; it is neither possible nor efficient to allocate the whole needed memory at the compile time. Performance of a program can partially depend on the allocator performance. However, "performance" does not have a clear meaning: it can either mean temporal efficiency (how quickly the program performs its tasks) or spatial efficiency (the program is using the minimal space needed). Moreover, the allocator performance is affected by the usage pattern, which is determined by the user program. Operating system vendors usually provide a generic allocator tuned to perform "well" for the usual use cases. But sometimes the performance is crucial and using a different allocator performing exceptionally well for the specific program or in the specific environment can improve it.

To be able to decide about using a specific allocator, some amount of information about their advantages and disadvantages is needed. The tool we describe here aims to provide all the relevant information about the allocator performance for an arbitrary usage pattern.

2 PERFORMANCE PROBLEMS CAUSED BY AN INAPPROPRIATE ALLOCATOR

The introduction mentioned two allocator performance aspects: spatial and temporal efficiency. Both of these criteria are affected by the algorithm which the allocator uses. Allocator usually requests large areas of memory from the operating system. An allocator operates on this continuous space, providing smaller parts of it to the user program. The dynamic allocation of storage is an online algorithm, which means the allocator must respond to the request immediatelly without knowledge about future requests, and it cannot change its decision later. This leads to

the fact that a choice made can later turn out to be suboptimal, or even totally wrong. It can be proven that a allocator optimal for every usage pattern cannot be constructed [2].

The spatial efficiency metric of an allocator is quite straightforward—the only criterion is that an allocator should not waste space already provided by the operating system. Ineffective layout of allocated space causes *fragmentation*. Two kinds of fragmentation exist: external and internal [2]. Both fragmentation types can cause unnecessary memory consumption.

The meaning of temporal efficiency is more complicated. There are two performance criteria. The first one is straightforward: the direct cost of single (de)allocation request. The second type is the performance of the whole program affected by allocator decisions indirectly. In certain environments, the memory placement itself can affect the program performance. An example of such environment is a multithreaded program running on a multiprocessor computer with both per-CPU and shared caches [1], where poor placement of memory owned by different threads can lead to false sharing or cache trashing. In a threaded environment, the performance can be hit also by the allocator design; if the allocator is considered a critical section and is guarded by a lock, threads cannot allocate memory simultaneously and have to wait for each other.

To allow the user a right choice of an allocator, a tool providing information how the allocator suffers by the described problems is needed. The goal of this work is to provide such tool.

3 DESIGN OF THE ANALYSIS TOOL

In order for the analysis tool to be universal, it needs to be able to provide measurements and other data for as wide area of program use cases as possible. Further, an ability to provide information about the allocator performance when it is used by a specific program is important.

The tool we design consists of two parts: first part creates a scenario program with a memory allocator usage pattern, the second part runs the scenario and collects data. The scenario is described in a domain specific language and it carries information about number of threads and the order, size and duration of the allocations done by different threads. This file can either be created by the user, generated randomly, or created by a tool capturing the usage pattern of a real running program. Several approaches were considered for the capturing tool, with the probable outcome of combinating a use of LD_PRELOAD [3] with a fake malloc implementation with capturing malloc and free calls by an external tool such as ltrace or Systemtap.

A scenario file is translated into a C program. In addition to the memory allocation and freeing commands, this program contains hooks where the data collection probes can register to get data about the execution. It is then compiled to a binary program by using standard C compiler. In order to test the same scenario binary with different allocator implementations, the memory allocator will be linked to the binary using the LD_PRELOAD [3] environment variable.

The scenario binary is run multiple times, enough to provide a sufficient amount of samples to obtain a statistically sound result. Different measures are collected by separate programs either hooked internally in the binary, or external tools observing the behavior of the program. Different measures need different approaches to collection. For example, the probes which collect the data about the spatial memory layout in the process address space after each memory allocation request take some time. If these probes were running simultaneously with the probe measuring the total scenario runtime, the time measurement would be incorrect, because they would contain the overhead of the layout examination. To avoid the incorrect results, the harness

lets the user choose the metrics he wants to collect, and warns about the conflicting ones. Only the probes for the selected metrics will be collecting data during one scenario run.

After the run, the data will be analysed and a statistical output provided to the user. Visual information can also be provided, such as visualization of the address space, showing the memory layout. This can be extended to a "player" showing an animation. The tool will provide the most important information about the impact on performance of the particular scenario.

4 SAMPLE MEASUREMENT

To show that allocator performance can affect program, a sample measurement was performed using a prototype version of the tool. The scenario options were set as: 8 threads doing the same work. This work is five million malloc and free calls of a 100000 integers array. The environment was a dual core Intel Core 2 Duo computer running 2.6.32.9-70 Linux kernel. Temporal efficiency of two allocators was compared: tcmalloc from Google performance tools and ptmalloc2 supplied in glibc. The former allocator claims to perform better in a multithreaded environment. Measurements in a table below confirms this claim.

Allocator	Userspace runtime	Kernel runtime
glibc-2.11.1-1.i686	44.61 seconds	9.36 seconds
tcmalloc-1.5-1.i386	34.27 seconds	1.07 seconds

5 CONCLUSION

This paper describes a design of a tool providing information about various aspects of an allocator performance. This design is based on a study of the possible problems caused by an allocator, and identification of the factors in both the environment and the usage pattern affecting performance. In the current state of the project, several prototypes of data collecting tools are created, and they will be integrated in a framework. The draft versions of the scenario generator, simulator and program usage pattern capturing tools were created and tested.

ACKNOWLEDGEMENT

This work was partially supported by the BUT FIT grant FIT-S-10-1 and the research plan MSM0021630528.

REFERENCES

[1] Drepper, Ulrich: What every programmer should know about memory, 2007, <http://people.redhat.com/drepper/cpumemory.pdf> [online]

[2] Wilson, Paul R. et al.: Dynamic Memory Allocation: A survey and critical review. In: Proc. Int. Workshop on Memory Management, Kinros, Scotland, 1995

[3] Loosemore, Sandra et al.: The GNU C Library Reference Manual, 2007, Free Software Foundation