Available online at www.sciencedirect.com

SCIENCE DIRECT®

The Journal of Systems and Software 72 (2004) 443–453

The Journal of Systems and Software

www.elsevier.com/locate/jss

The design and analysis of a quantitative simulator for dynamic memory management ☆

Chia-Tien Dan Lo *, Witawas Srisa-an [1], J. Morris Chang [2]

Department of Computer Science, University of Texas at San Antonio, 6900 N. Loop 1604 W., San Antonio, TX 78249, USA

Received 30 November 2000; received in revised form 17 July 2002; accepted 30 September 2002
Available online 5 December 2003

Abstract

The use of object-oriented programming in software development allows software systems to be more robust and more maintainable. At the same time, the development time and expense are also reduced. To achieve these benefits, object-oriented applications use dynamic memory management (DMM) to create generic objects that can be reused. Consequently, these applications are often highly dynamic memory intensive. For the last three decades, several DMM schemes have been proposed. Such schemes include first fit, best fit, segregated fit, and buddy systems. Because the performance (e.g., speed, memory utilization, etc.) of each scheme differs, it becomes a difficult choice in selecting the most suitable approach for an application and what parameters (e.g., block size, etc.) should be adopted.

In this paper, a DMM simulation tool and its usage are presented. This tool receives DMM traces of C/C++ or Java programs and performs simulation according to the scheme (first fit, best fit, buddy system, and segregated fit) defined by the user. Techniques required to obtain memory traces are presented. At the end of each simulation run, a variety of performance metrics are reported to the users. By using this tool, software engineers can evaluate system performance and decide which algorithm is the most suitable. Moreover, hardware engineers can perform a system analysis before hardware (e.g., modified buddy system, first fit, etc.) is fabricated.

© 2003 Elsevier Inc. All rights reserved.

Keywords: Dynamic memory management; Garbage collection; Java virtual machine; Tools; Object-oriented programming; DMM simulator

1. Introduction

Object-oriented programming features such as dynamic binding and abstract data type promotes the reuse of software modules. Software reuse leads to faster and cheaper software development. At the same time, software becomes more robust and easier to maintain because its structure is inherently decoupled into classes and objects (Pressman, 1997). To achieve these benefits, object-oriented languages (e.g. C++ and Java) heavily rely on the use of dynamic memory management (DMM) to create generic objects (e.g. class libraries, templates) that advocate the practices of software reuse. As a result, the frequency of dynamic memory allocation and deallocation in object-oriented applications can be as much as 10 times higher than procedural counterparts.

In C++, dynamic memory is allocated via the operator *new*. The operator returns a pointer to an allocated memory area. The memory is then deallocated by the operator *delete*. Most compilers simply map these operators directly to the *malloc*( ) and *free*( ) functions of the standard C library (Applegate, 1994). Therefore, C++ applications which have been developed to be general and reusable, invoke the *malloc*( ) and *free*( ) functions frequently. Studies have shown that DMM can consume up to 38% of the execution time in C++ applications (Calder et al., 1994). Apparently, the

☆ This work is partially supported by National Science Foundation under grants CCR-0098235 and CCR-0113335.

* Corresponding author. Tel.: +1-210-458-7433; fax: +1-210-458-4437.

E-mail addresses: danlo@ieee.org (C.-T.D. Lo), witty@cse.unl.edu (W. Srisa-an), morris@iastate.edu (J. Morris Chang).

[1] Address: Department of Computer Science and Engineering, University of Nebraska at Lincoln.

[2] Address: Department of Electrical and Computer Engineering, Iowa State University.

performance of DMM can have a substantial effect on the overall performance of object-oriented systems.

Additionally, the object-oriented approach is also being used in client/server applications (e.g. Web server, Database server, and Mail server). Client/server applications often possess high degree of concurrency. A study has shown that the allocation and deallocation rate in a web server during peak time can be as high as one million *malloc* and *free* calls per second (Larson and Krishnam, 1998). Moreover, Java programming language has also been used in distributed programming environments (e.g. Java web computing (Chung and Kim, 1998)) because of its portability and ease for multithreading. In Java, dynamic memory is also allocated via the operator *new*. However, allocated memory areas are reclaimed automatically by the run-time system, *garbage collector*. It is worth noting that Java applications allocate dynamic memory even more frequently than C++. For example, to allocate an array of class objects, we need to dynamically allocate each array element. Thus, the performance of DMM can have a great impact on the overall performance of such systems. Software engineers should carefully consider which DMM scheme is the most suitable for their applications.

Amdahl's Law reminds us that the opportunity for improvement is affected by how much time the event consumes. Thus, making the common case fast will tend to enhance the performance better than optimizing rare cases. As stated earlier, more than one-third of execution time in C++ applications can be spent on managing dynamic memory. Obviously, one of the improvements that can be made to the overall performance of object-oriented applications is to improve the performance of DMM. Various allocation schemes (e.g. first fit, best fit, segregated fit, buddy system, etc.) are being used to manage dynamic memory. Each scheme has its own advantages and disadvantages. For example, the buddy system is known for its speed but suffers from high internal memory fragmentation. One of the most comprehensive surveys on dynamic memory allocation was written by Wilson, Johnstone, Neeley and Boles (Wilson et al., 1995). In their survey, the focus was on the potential performance issues (i.e. speed and memory utilization) of different allocation approaches. To make accurate assessments on the performance of each scheme in real applications, a simulator is needed to dynamically provide the performance metrics of different approaches to DMM. There also have been research efforts in providing hardware supported for DMM (Chang and Gehringer, 1996; Chang et al., 1999; Lo et al., 1998; Lo et al., 2001a). This intensifies the need for a simulator because hardware approaches are difficult to get accurate assessments without the actual implementation. However, the implementation can be costly. A simulator can be used to assess several policies at the same time whereas the real implementation cannot. Thus, a research effort in implementing a DMM simulator is crucial.

In this paper, a DMM analysis tool and its usage are introduced. This tool simulates four dynamic memory allocation schemes that include first-fit, best-fit, binary buddy system, modified buddy system and segregated list (please refer to previous work for detailed description of each scheme). This tool receives DMM traces and performs DMM operations according to the scheme designated by the user. At the end of each simulation run, the simulator generates different performance metrics including watermark, the number of occupied objects, the amount of dynamic memory required, the amount of memory requested, number of DMM calls (*malloc*, *free*, and *realloc*), the internal fragmentation, the external fragmentation, average object size, page fault ratio, allocation look-aside buffer (ALB) hit ratio, etc. The possibilities of the simulator are tuning hardware design of a dynamic memory manager for a specific application, adjusting memory utilization for an application in an embedded environment, etc. By using this tool, software engineers can analyze the performance metrics and select the most suitable DMM scheme to be used in their applications. Moreover, hardware engineers can perform a system analysis before hardware (e.g., modified buddy system, first fit, etc.) is fabricated.

The remainder of this paper is organized as follows. Section 2 summarizes previous work in DMM. Section 3 provides a high-level description of the proposed approach. Section 4 delineates the performance metrics that the tool should generate and elaborates the analysis of simulation results. Section 5 depicts the design of the proposed simulator. Section 6 concludes this paper.

2. Previous work

In recent research, studies have shown that object-oriented applications written in C++ can run up to 10 times slower than comparable C programs. This execution time overhead can largely be contributed to inefficient heap management algorithms. Since object-oriented applications tend to be more memory intensive than procedural ones, the ability to effectively manage heap memory is crucial to the performance of such systems. In software approaches, the DMM execution time is linearly proportional to the number of objects existing in the system. For example, as the calls to *malloc* intersperse with the calls to delete, the heap becomes increasingly fragmented and the occupied list and the free list become longer as well. A thorough study on memory fragmentation has been conducted by Johnstone et al. (Johnstone and Wilson, 1998; Johnstone, 1997). They have shown that almost none of the wasted memory is due to true fragmentation based on eight C or C++ programs (Johnstone and Wilson, 1998).

However, some hardware approaches such as the modified buddy system (Chang and Gehringer, 1996; Chang et al., 1999; Lo et al., 1998; Chang et al., 2000; Lo et al., 2001a) and Java programs have not been studied. This has led to our approach in investigating the behaviors of Java programs and hardware approaches. Although a thorough survey pertaining to DMM policies is done by Wilson et al. (1995), we summarize and complement a hardware approach in the following sections.

In first fit approach, the list is searched from the beginning. The free block that is large enough to satisfy the request is then selected. If the block is larger than necessary, it is split and the remainder is put back on the free list. The major problem with first fit is that the larger blocks at the beginning of the list are often split first. Since these blocks are often larger than the requested size, the remaining portions are put back on the list. If the splitting occurs frequently, the beginning of the list may have many small-unusable blocks. Therefore, the search time may increase since the search must go past them each time a larger block is requested (Wilson et al., 1995).

In best fit, the list is searched to find the smallest free block large enough to satisfy a request. The basic rationale for this approach is to minimize the amount of fragmented space (caused by splitting). However, the problem may still occur if the selected block is not a perfect fit to the requested size. In such situation, splitting may put small memory blocks back to the list. These small blocks may be unusable. Additionally, the search in best fit is very exhaustive. Thus, this scheme may not scale well to large heaps with many free blocks (Wilson et al., 1995).

Segregated fit uses an array of free lists where each list holds free block of a particular size. One of the fastest allocators available today is Doug Lea's implementation (Lea, 1996). In this scheme, every heap consists of 128 linked lists. For the first 64 lists, each list represents one size. The first list has the size of eight bytes and the subsequent 63 lists have eight bytes separation between them. For example, list 1 would contain the objects of size eight bytes. List 2 would contain the objects of size 16 bytes and list 3 would contain the objects of size 24 bytes. For the last 64 lists, the size granularity is not fixed. These lists are used to store objects larger than 512 bytes and the first fit approach is used in these lists. The major problem with segregated free list is low memory utilization (i.e. the percentage of occupied memory versus the amount of heap memory available to a process) because lists of a certain size may not be used as frequently as others.

Buddy system is a variation of segregated lists that supports limited but efficient kind of splitting and coalescing (Lo et al., 1998; Wilson et al., 1995; Lo et al., 2001a). In a binary buddy system, when a block of a given size is to be allocated, it locates a block that is at least as large as the allocation request, and whose size is a power of two. The block is split in half as many times as possible, until it can no longer be split while still satisfying the memory request. When a block is split, its two halves are known as buddies. At the time a block is freed, if the block's buddy is also free, the buddy system coalesces the two buddies immediately, making available a larger piece of memory that can be used later. One of the major advantages of buddy systems is its simplicity of address computation. In buddy systems, when a block is freed, its buddy can always be found by a simple address computation. On the other hand, buddy systems can suffer from high internal fragmentation. Buddy system is used in the BSD UNIX implementation of the *malloc* function.

The modified buddy system (Chang and Gehringer, 1996; Chang et al., 1999; Lo et al., 1998; Srisa-an et al., 1999; Chang et al., 2000) uses a bitmap to record which regions of the heap are occupied and which regions are free. A bitmap is a simple vector of one-bit flag, with one bit corresponding to a system specified memory unit (each bit can represent 4 bytes, 8 bytes, or 16 bytes depending on the implementation) of the heap area (Wilson et al., 1995). Two advantages of the bitmap approach are the support of localized searching and the improvement in cache locality. In a bit-map scheme, the searching of free memory can be indexed by address order where searching can begin at a specific address. Additionally, the bitmap can be stored separately from the objects. Thus, it can improve the locality of searching. On the other hand, the disadvantage of the bitmap approach is linear search time. If searching begins at the beginning of the bitmap every time, the search time is linearly proportional to the number of free objects. On the other hand, if searching begins from a certain address to a certain address, the search time is only linearly proportional to the size of area searched (Wilson et al., 1995). In the modified buddy systems, the internal fragmentation, a well-known drawback in the buddy system was improved.

3. High-level description of the dynamic memory simulation

3.1. Overview

Memory allocation patterns generated from different applications usually have different characteristics. These differences often impact the performance of a particular algorithm. For example, if a program often makes requests of size 2^n, the binary buddy system would be the most suitable for this application. On the other hand, if the requested sizes are randomly distributed, binary buddy system may result in more internal fragmentation. This performance variation creates the need to

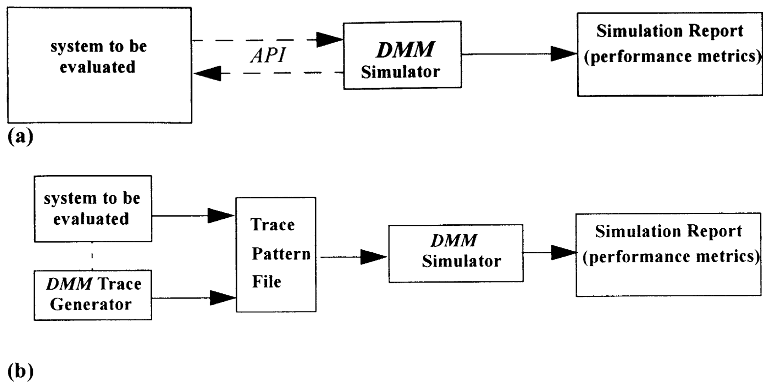

Fig. 1. Two types of simulation environments: (a) on-line simulation, (b) off-line simulation.

evaluate different algorithms before or during a system development. Typically, two types of performance evaluations can be applied: on-line evaluation and off-line evaluation.

Normally, on-line evaluation needs to include a dynamic memory module and collect the required information after the system finishes its execution. Performance indices such as response time and total allocated memory size can then be obtained. To switch among various algorithms, the on-line approach needs to rebuild the whole system to incorporate the new testing modules. Fig. 1(a) illustrates the basic procedure to perform on-line simulation. On the other hand, the off-line evaluation allows different patterns to be tested without additional compilation. Consequently, the trace data can be efficiently simulated for different algorithms. The drawback of the off-line evaluation is its lack of measuring the real elapsed time during the execution. However, the number of allocations can be used to indicate timing with regard to DMM. Therefore, the off-line evaluation model is adopted in our simulator. Furthermore, the first step to conduct an off-line simulation is to obtain dynamic memory traces. Fig. 1(b) depicts the steps necessary to perform off-line simulation.

3.2. Trace pattern generation

There are several stages in dynamic memory simulation: acquisition of allocation/deallocation patterns, simulating a system via the use of these patterns, collecting outputs from the simulator, and analyzing the output. A typical simulator should accept input from a file and generate necessary output data by applying different algorithms. Tracing memory patterns is the first step to perform off-line simulations. Generally, there are two types of patterns that can be used: synthetic patterns and real program tracing patterns.

3.2.1. Synthetic patterns

Synthetic patterns can be generated by calling the random number library routines such as *rand*( ) and *srand*( ) in the standard C library. Synthetic traces are never representative of real allocation behavior (Johnstone and Wilson, 1998). They are provided for reference only. Here are some suggestions for generating synthetic patterns.

(a) Create randomly sized blocks and delete some in a 3:1 ratio until 2 MB is reached, then delete all blocks. This test measures an allocator's ability to allocate into a partially fragmented heap.
(b) Create many randomly sized blocks, then delete all in reverse order. This is a basic test measuring allocating and deallocating in a straight line.

3.2.2. Real traces

The synthetic patterns can be used as test criteria. However, they do not reflect the actual behavior of a particular program. To perform analyses on DMM in real applications, trace patterns from real programs must be generated. However, acquiring tracing data from a real program requires some advanced software techniques. The techniques used to generate traces from a program can be classified into two categories: source code insertion and object code insertion. The source code insertion needs, of course, the source code of a test program and DMM libraries. The object code insertion, in contrast, requires object code modifications by using special tools such as binary editors.

3.2.2.1. Object code insertion. The object code insertion or binary editing has different design goals and offers a library of routines for modifying object code directly. The object code insertion can be performed either before linking or after linking. *Pixie* [3] is a program that does object code insertion after linking whereas *Purify* (Hastings and Joyce, 1992) does it before linking. Object code insertion before linking may be easier because the

[3] *Pixie* is a program that *MIPS* computer systems distribute to insert profiling code directly in an executable *MIPS* program. Detailed usage can be referenced through *IRIX* man pages.

code has not been relocated. Basically, these tools will read object files generated form existing compilers, adds trapping instructions without disturbing the symbol table or program logic, and feeds the output to existing linkers. However, it is hard to modify the low-level machine languages directly and the modification is hardware dependent and highly non-portable. Due to the hardware complexity, the object code insertion approach is error prone and requires full understanding of a hardware architecture.

3.2.2.2. Source code insertion. When a test program makes a *malloc* function call, the block size information will be passed to the library which stores the object code for DMM routines. A memory request record can be obtained by trapping the *malloc* function call to a user-defined routine instead of the system default one. To do so, a DMM package with source code needs to be instrumented to leave traces. For example, the entry point for a *malloc* function needs to be added a log line.

The added line can print a record showing the memory request with size (bytes), the starting address for the allocated block and a tag "*m*" indicating the *malloc* operation type. This information can be obtained by adding a line right before every return statement in a *malloc* routine. A DMM routine may have several exit points. Normally, the return value will be *NULL* for a *malloc* routine if it fails to allocate the requested memory size. By reading the tracing patterns, the starting address component can be used to distinguish whether a memory request succeeds. To simplify the insertion task and keep the integrity of an *malloc* design, the original *malloc* is renamed to *primitive_malloc* and a new trapped routine named *malloc* is added. Furthermore, for each memory request, the program calls the trapped *malloc*; the trapped *malloc* calls *primitive_malloc* and print a log record.

Some concerns in using *I/O* routines are that the buffer could interfere with the output data in a multi-process program and the routines themselves may call *malloc*. The *printf*( ) used to log a line must not call *malloc*; otherwise, the program behavior can be totally distorted. It is easy to check whether a *printf*( ) calls *malloc* by putting one *malloc* in a test program. However, there is a *printf*-style routine that guarantees not to allocate memory (Flux, 2002). In a typical Unix system, there are three types of *I/O* buffer methods: fully buffered (e.g., puts( )), line buffered (e.g., *printf*( )) and non-buffered (e.g., *write*( )). The *printf* function is lined buffered if it outputs to a terminal, i.e., terminal *I/O*. However, it becomes fully buffered if the output is redirected to a file. During a process creation, if the *I/O* buffer is not flushed, both the parent process and its child process will output to the same buffer that in turn duplicates the output. This problem can be solved by setting the *I/O* to be non-buffered or using a non-buffered *I/O* such as *write*( ). An *API*, *setbuf*, can be used to set *I/O* buffer size. By setting buffer size to zero, the statement is equivalent to set the I/O to be non-buffered. A memory tracer for both single threaded and multi-threaded applications must be unbuffered I/O and guarantees not to allocate memory.

In a multithreaded program, although different threads in a process actually share the same heap area, it is not necessary to differentiate the source of memory requests. However, a thread identifier (TID) can be added into the log if the per-thread dynamic memory operations are significant. The thread library API, *pthread_self*, will return the TID of the current running thread which is defined in *POSIX.1b* standard.

After modifying the source code for a DMM package, it can be compiled into object code as a user-defined repository. The routines can be linked into a tested program by a compiler, e.g., *gcc test.c dmmlib.o* where *dmmlib.o* is the object code for the modified DMM package. We currently adopt this technique to generate traces from real programs. It is true that the system shared library of the DMM can be replaced with the modified library. However, the dynamic memory operations of an application may call a static DMM library instead. To make sure those operations get recorded, the static linking is required.

3.2.2.3. Pros and cons. The advantages and disadvantages of the source/object code insertion techniques are as follows. First, the setup cost is low for the object code insertion compared with recompilation of the source code. By inserting object code directly, it is much faster than recompilation because re-translation from high-level language programs to machine code is avoided. Second, object code insertion is convenient. It is hard to build a large system if the source code resides in many directories (although some make utilities can help). On the other hand, the object code is already aggregated by a linker. It is simply one entity and can be easily handled. Third, object code insertion is multilanguage support; many languages are quite similar at the object code level. Developing an object code insertion tool can be used for applications written in different kinds of languages. Lastly, the object code insertion is more complete. All of the code including third-party and vendor libraries can be instrumented. Basically, there is no restriction to use the technique and especially no source code is needed. Unfortunately, the object code insertion heavily depends on hardware systems such as instruction-set structures and operating systems. Porting an object code insertion tool into another platform poses some essential problems. By contrast, obtaining decent source code for a representative system is a major obstacle for the source code insertion technique. Only few of benchmark programs are available such as *SPEC* series benchmarks (SPEC, 2002), *GNU GPL* software packages (GNU,

2002) and *Java SPEC* (SPECjvm98, 1998). Moreover, it is crucial to analyze source code in many aspects of a typical software engineering process such as the estimation of lines of code, function points, etc.

3.3. Outputs of the simulator

The simulator will output performance metrics such as total memory required, internal and external fragmentations, time used for DMM and the like. Detailed performance indices will be elaborated in Section 4. The performance metrics can be further analyzed to provide significant guidance for system designers to select the best DMM algorithm.

4. Performance metrics and analyses

A good dynamic memory manager must have better values for the following performance metrics. Some of the performances such as time spent in allocating a memory block can only be collected through on-line simulation.

4.1. General performance metrics

Minimizing time: Speed is one of the most important goals of an allocator. A high speed dynamic memory allocator, deallocator should be as fast as possible. In some applications such as a long-running server programs, allocation and deallocation frequency could be very high compared with reallocation. Hence, they must be very fast. Typically, the program may trade speed with space. If memory utilization is not the major factor, a policy with higher watermark may be a choice. To collect the time spent in a dynamic memory operation, the resolutions of a regular *clock*( ) *API* is not accurate enough. Instead, a real-time timer is needed. For example, most systems provide the *POSIX.1b* timer whose resolution is in nanoseconds. A *Linux* timer with resolution in nanosecond is proposed in (Wang and Lin, 1998).

Minimizing space: A good allocator should obtain as little memory from the system as possible. Both internal and external fragmentation should be minimized in order to achieve high memory utilization. To measure the total memory required for DMM, we adopt a watermark which is a pointer indicating the highest memory the *malloc* has ever reached. The watermark represents the amount of heap memory needed to run that application. Later on, the application may return the unused portion of the heap back to the system but the watermark remains unchanged. A good allocator will result in a lower watermark. The lower the watermark, the less memory a program requires. In other words, the memory utilization will be better.

Maximizing locality: Allocating chunks of memory that are typically used together near each other. This helps to minimize page faults and cache misses during execution and hence improves performance. A locality factor can be defined as the standard deviation of all the allocated memory addresses. As the value of the locality factor becomes lower, the overall locality improves. For example, there are three objects with addresses 0, 100 and 200 in memory system *A*. However, the three objects may have addresses 0, 10 and 20 in memory system *B*. The standard deviation of addresses of objects in *A* is 10 times larger than that of memory system *B*. Consequently, memory system *B* has better locality than memory system *A*.

Minimizing system bus traffic: Frequent transferring of data between *CPUs* and main memory during DMM and garbage collection periods may cause system buses to be saturated. Thus, the system performance may be degraded. The system bus bandwidth should be shared reasonably between all computations to improve overall performance of an application. In (Chang and Gehringer, 1993), objects are created and garbage collected in the cache because of the objects' short life in order to alleviate the system bus traffic. However, bus traffic is hard to measure and it will be contributed into total system elapsed time.

Reducing costs for cache coherence: A potential problem causing cache performance to be degraded is the cache coherence issue in a multiprocessor system. This can also increase the cache miss rate and the system bus traffic. By associating each thread with a private heap can alleviate this problem.

Thread independence: A memory object allocated by a thread can be used by another thread. Besides, the memory object can be freed by a third thread even though the original allocated thread has died. The requirement may be significant in some server applications.

Deterministic turnaround time: In real-time applications, the turnaround time need not only be short but also has to be bounded. In other words, both dynamic memory managers and garbage collectors should not result in nondeterministic behavior and must have a short and bounded turnaround time.

Low input sensitivity: A good dynamic memory manager should behave well for all sets of input patterns, i.e., insensitivity to input patterns, which are the combinations of *malloc*'s and *free*'s. It is not very useful if working for one set of input. However, due to different characteristics of programs, it is hard to have a design that suits for all instances. Consequently, it is acceptable that a dynamic memory manager works well for most input patterns.

4.2. Analyses of output metrics

We evaluate our scheme on several memory allocation and deallocation traces from various C, C++, and

Java programs. These programs are drawn from different application areas, including a compiler (*gcc*), a *CAD* tool (*electric*), a *PDF* document viewer (*xpdf*), a visual calculator (*calcJ*), a compressor (*compress*), a Java virtual machine (JVM) checker (*check*), a Java expert shell system (*jess*), a database system (*db*), a Java compiler (*javac*), an *MPEG* decoder (*mpegaudio*), a raytracer (*mtrt*) and a Java parser generator (*jack*). All programs are publicly available applications. The first two programs are written in C. *Xpdf* is written in C++, and the rest are written in Java, most of which are from *SPECjvm98* (SPECjvm98, 1998).

The outputs of the simulator are various metrics that programmers can use to evaluate the performance of each allocation scheme. The reported metrics are due to interactions between the program and the policy of the allocator being studied. Table 1 illustrates some performance metrics such as number of *malloc* and *free* calls, average *malloc* size, and total memory requested from a system. These metrics can be used to analyze whether a program is dynamic memory intensive and the maximal amount of memory requested. From Table 1, *CalcJ* has the highest dynamic memory intensity. The results also show that Java programs tend to consume more dynamic memory compared to C or C++ programs. Note that *gcc* uses an "obstack" allocation mechanism, an extension of C language, to optimize the allocating and deallocating objects in stack-like ways (Johnstone and Wilson, 1998).

In a typical DMM implementation, larger objects are separated from smaller ones. Larger objects often need special manipulations using different algorithms. In our simulator, we filter out objects whose sizes are larger than 4 KB (one page, typically). Table 2 shows the characteristics of eight benchmarks with an 8-byte block from *SPECjvm98* suite (SPECjvm98, 1998) which confirm the result in (Dieckmann and Hölzle, 1999). Note that the average allocation size is the average size of all small object requested. Also, the free counts are the memory deallocations by the Kaffe 1.0.5 GC system. In all benchmarks, up to about 50% of memory operations are deallocations. Because deallocations are done by garbage collector, this indicates that *SPECjvm98* benchmark programs frequently invoked the garbage collector.

Table 3 illustrates the dynamic memory utilization among different algorithms. For example, the watermark for the binary buddy system in *gcc* is 220 KB whereas the watermark for first fit is 216 KB. The analysis of this table determines the memory utilization of each scheme. Furthermore, we use the first-fit policy as a benchmark because it has the best memory utilization (Wilson et al., 1995).

Table 1
Characteristics of some program traces

Name	Malloc calls	Free calls	Avg. malloc size (bytes)	Total memory requested (bytes)
gcc	2526	1077	1594.40	427,587
electric	3333	167	48.76	162,533
xpdf	10,859	9141	45.50	494,109
calcJ	34,033	13,107	421.50	14,346,429

Table 2
Characteristics of *SPECjvm98* program traces (small objects)

Name (block size)	Malloc calls	Free calls	Avg. malloc size (bytes)	Total memory requested (bytes)
compress(8)	47,737	0,122	70.65	3,372,588
check(8)	54,208	35,471	72.57	3,933,691
jess(8)	116,995	91,819	64.82	7,583,072
db(8)	57,319	38,989	74.49	4,269,807
javac(8)	99,154	69,338	68.87	6,828,423
mpegaudio(8)	61,680	36,293	69.01	4,256,462
mtrt(8)	367,451	348,562	73.38	26,964,046
jack(8)	1,040,380	1,003,433	46.79	48,678,920

Table 3
The watermarks among different algorithms

Trace programs (block size)	First fit watermark in bytes	Modified buddy watermark in bytes	Doug Lee's F.F. watermark in bytes	Binary buddy watermark in bytes
gcc (8)	216,150	220,668	241,855	219,647
electric(16)	19,542	19,563	79,199	22,671
xpdf(16)	28,507	30,987	90,471	35,583
calcJ(16)	831,044	903,336	1,228,143	1,556,479

The tool also provides information such that fragmentation, ALB hit ratio, heap page fault ratio, scatter factor (SF), etc. Due to the space limit, the simulation result for one of the *SPECjvm98* benchmarks is shown in Table 4. The others can be found in (Lo et al., 2001a; Lo et al., 2001b). The significance of Table 4 is the demonstration of a fair comparison among three policies based on the simulation results. The ALB hit ratio is crucial to the performance of the modified buddy system (Chang et al., 2000). The watermark clearly shows how efficient a management scheme is. The lower watermark means the higher efficiency of that system. The page fault is still one of the highest performance penalties in computer systems today. Page fault ratio details the potential page miss penalty that a particular system would have to pay.

It is worth noting that the SF represents the average of unoccupied memory size divided by the total number of holes (smaller SF indicates a more external fragmented heap). We also can compare Kaffe 1.0.5 GC system with the modified buddy system in terms of page fault counts. The page fault rates are measured by monitoring every memory access (write and read) according to memory size ranging from 100×4 to 300×4 K. The details of the measurements can be found in (Lo et al., 2000; Lo et al., 2001b). Table 5 shows the page

Table 4
The output of the simulator for the compress benchmark (*SPECjvm98*)

Compress	Modified buddy system						First fit						Binary buddy system					
	Internal frag.	External frag.	ALB hit rate	Water mark	Page fault ratio	SF	Internal frag.	External frag.	ALB hit rate	Water mark	Page fault ratio	SF	Internal frag.	External frag.	ALB hit rate	Water mark	Page fault ratio	SF
FIFO strategy with memory size 100 4k-pages (in bytes)	123,196 (3.52%)	174,552	76661/77859 (98.46%)	1,337,360	565/181453 (0.311%)	11.63	123,196 (3.52%)	32,888	76910/77859 (98.78%)	220,668	403/181453 (0.222%)	21.6	123,196 (3.52%)	28,096	76650/77859 (98.45%)	1,597,440	620/181453 (0.3417%)	60.72

faults can be eliminated up to 74.4% by applying the new policy, the modified buddy system when setting a heap size of 300 4 KB pages and a *Random* page replacement policy. The major reason is that Kaffe 1.0.5 GC system uses one-size-object per page heap layout whereas different object sizes can be allocated in a page in the modified buddy system.

5. Simulator design

A simulator is built based on the performance metrics (proposed in the earlier section) written in C. Fig. 2 shows the architecture of the simulator. The current version of the simulator can accept both synthetic and real traces. The synthetic traces have been introduced in Section 3.2.1. Techniques used in the real traces such as those of C/C++ and Java programs will be delineated in later sections. Furthermore, DMM policy, ALB policy (e.g., *FIFO*, *Random*, *LRU*, etc.), Page Replacement Policy (e.g., *FIFO*, *Random*, *LRU*, etc.) or other DMM parameters (e.g., block size, page size, large object size, etc.) are designated before the simulator starts. The output of performance metrics is generated according to the DMM parameters and the DMM policies such as first-fit, segregated-list, binary buddy systems, best fit, and modified buddy system.

5.1. C++ Program tracing

For programs written in C++, the dynamic memory allocation/deallocation patterns can be logged by overloading the *new* and *delete* operators. Because most C++ programs use *malloc* and *new* interchangeably, trapping *malloc* may not be sufficient because they may have different implementations. Moreover, by overloading *new* and *delete* for special situations, a program can be made more efficient for DMM. For example, a program may have to run for a very long time with restricted resources in embedded and real-time systems. Such systems may also require that memory allocation always takes the same amount of time and there is no allowance for heap exhaustion or fragmentation. A customized memory allocator by overloading *new* provides a solution and in the overloading function, memory allocation/deallocation patterns can be recorded for a further analysis.

In C++, there are three types of overloading for *new* and *delete* operators: overloading global *new* and *delete*, overloading *new* and *delete* for a class and overloading new and delete for arrays. Overloading global new and delete is a common approach to collect memory allocation/deallocation patterns because of its simplicity. Additionally, overloading delete operations can help programmers to detect logical errors. For example, if the program tries to free a *NULL* pointer, the overloaded

Table 5
Page fault analysis for compress (Kaffe GC System/Modified buddy system)

Compress	Heap size (number of 4 KB pages)				
	100 kaffe/mod (%improve)	150 kaffe/mod (%improve)	200 kaffe/mod (%improve)	250 kaffe/mod (%improve)	300 kaffe/mod (%improve)
FIFO	738/565(23.4%)	450/359(20.2%)	275/167(39.3%)	167/89(46.7%)	92/31(66.3%)
Random	850/620(27.1%)	549/399(27.3%)	358/262(26.8%)	230/129(43.9%)	117/30(74.4%)
LRU	654/532(18.7%)	399/315(21.6%)	220/154(30.0%)	133/77(42.1%)	78/27(65.4%)

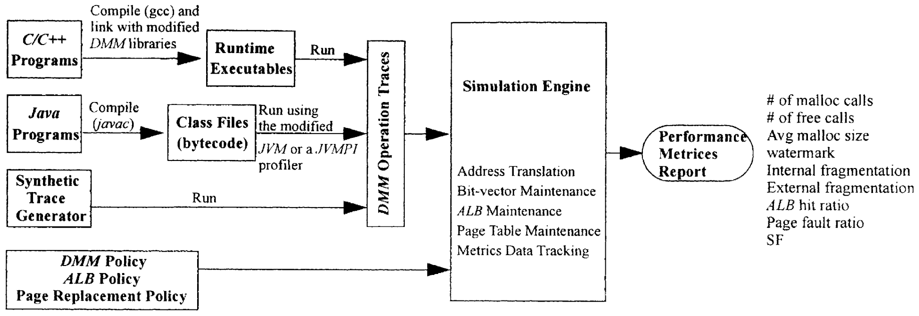

Fig. 2. The architecture of the simulator.

delete can display an error message to indicate that the program logic may be incorrect. Because most C++ programs, such as *xpdf*, may use both *malloc* and *new* to request dynamic memory, trapping *malloc* may not be enough if they have different implementations.

When the global allocator tries to allocate several copies of an object with special sizes, the memory utilization may be very bad. For example, an object of size 17 bytes is to be allocated by a buddy system. The system will always give 32 bytes to the object. By overloading new and delete for the particular class, a better DMM mechanism can be provided on a per-class basis. The overload operator *new* and *delete* for a class are invoked whenever an object of that class is created. However, the global operator new is called to allocate enough storage for the array all at once and the global delete is called to release that storage if an array of those class objects were created. To trace this type of array allocation, one can overload the special array versions of operator *new* and operator *delete* for the class.

5.2. Java program tracing

In Java, the *delete* or *free* function have been removed from the language specification. Instead, a garbage collector is responsible to release the storage used for an object when it becomes a garbage. The garbage collector will first call *finalize*( ), and reclaim the object's memory on the next garbage collection pass. Memory tracing patterns can be obtained by modifying a JVM. Most of the JVMs are written in the C language. Every JVM has its own implemented routines for DMM. In *Kaffe* (Kaffe, 2002), all the routines required for garbage collection and memory management are stored under mem directory. By modifying the routines, all the memory tracing patterns can be collected. Currently, we have successfully traced several versions of JVMs such as *Kaffe* and *JDK1.2*.

In JDK1.2, a profiler interface called the Java virtual machine profiler interface (JVMPI) is provided for developers to design profilers. The JVMPI is a two-way function call interface between the JVM and an in-process profiler agent. On one hand, the virtual machine notifies the profiler agent of various events, corresponding to, for example, heap allocation, thread start, etc. On the other hand, the profiler agent issues controls and requests for more information through the JVMPI. For example, the profiler agent can turn on/off a specific event notification, based on the needs of the profiler front-end. By using JVMPI, all the dynamic memory traces can be obtained (Javasoft, 2002). The traces can be fed into our simulation engine for a further analysis.

5.3. Performance metrics report

The simulator generates performance metrics based on the designated algorithm. Performance metrics such as *water_mark*, *internal fragmentation*, *external fragmentation*, etc., will be illustrated later. Due to their special characteristics, more performance indices are reported for buddy systems. These include ALB size, bit-vector length and bit-vector replacement policy in ALB (Chang et al., 1999; Chang et al., 2000). A bit-vector is used to represent allocation status on which a buddy system can manipulate its corresponding memory space. Hence, several bit-vectors are required to represent a whole heap. However, only one bit-vector is active at a time. To accelerate fetching bit-vectors, an ALB is used to cache currently used bit-vectors. These parameters are related to overall performance and hence are investigated.

In order to make a fair comparison among different algorithms. We adopt a bit-map approach to implement the basic data structure representing allocation status. Each bit in a bit-map represents the allocation status for a block of memory in the heap space. A block size represents the smallest memory chunk that can be allocated. Typically, the block size is 8 bytes. Total occupied memory size is the total number of all occupied memory blocks times the block size at any instant. Theoretically, by the end of a program's execution, the total occupied memory size will be zero. If the chosen block size is small, the bitmap length would be longer. On the other hand, a bigger block size will cause higher internal fragmentation.

During DMM operations, buddy systems could have some memory chunks large enough to satisfy the requested size. However, the system may not be able to use them. Those memory chunks are referred to as blind spots (Lo et al., 1998; Lo et al., 2001a). One solution to solve the blind spot problem is by shifting the bit-vectors. In our simulator, a maximum number of shifting bits can be set until the blind spots are found. The number of shifting bits required to find blind spots are recorded.

Total memory (bytes) required by *malloc* is an indicator showing whether a program is dynamic memory extensive. Total memory requested from the operating system indicates the efficiency of memory utilization. In our simulator, a watermark is used to represent total requesting memory. Furthermore, the total memory overhead can be defined as $100 \times ((water\_mark) - total\_malloc\_blocks)/(water\_mark)$. A good dynamic memory manager should have lower memory overhead.

Two categories of fragmentations are defined: internal and external fragmentations. The internal fragmentation is defined as $total\_malloc\_blocks \times block\_size - total\_malloc\_size$. Two different sources can causes internal fragmentation. One is from the block size and the other is from the round-up allocation size. For example, in Doug Lee's *malloc* (Lea, 1996), all the memory request sizes will be round-up to multiples of eight. This is because the *Sparc* architecture requires floats to be 8-byte aligned. Consequently, memory request of size 9 will cause 7 bytes as internal fragmentation.

External fragmentation, however, is caused by the allocation scheme used. At any given time, the total unoccupied memory size is defined as the subtraction of watermark and total occupied memory. We also use the average of unoccupied memory size divided by the total number of holes as a SF. The bigger the SF, the more the objects spread over the heap. The SF also indicates that an allocator would have more chance to reuse holes under the watermark.

Average malloc size is defined as the average size of all the requested sizes. The simulator allows objects of different sizes to be managed differently. For example, if the object sizes are specified to be from 0 bytes to 4 KB, the simulator would only receive input within that range. Any objects larger than 4 KB are regarded as large objects. This feature provides the programmers with the flexibility to manage different size objects with different schemes.

6. Conclusion

There are various approaches to manage dynamic memory. Such schemes include first fit, best fit, segregated fit, and buddy systems. The performance (speed and memory utilization) varies from one scheme to another. The software engineers often face difficult choices in selecting the most suitable approach for their applications. In this paper, a quantitative systematic DMM simulator that can simulate several DMM algorithms is presented. This tool receives DMM traces and performs DMM operations according to the scheme designated by the user. At the end of each simulation run, different performance metrics such as the watermark, the number of occupied objects, the internal and external fragmentation, etc., are reported to the users. By applying this approach, both hardware and software engineers can evaluate the system performance in regard to DMM and make the best decision in selecting a DMM policy and its corresponding parameters for their applications.

References

Applegate, A.D., 1994. Rethinking memory management. Dr. Dobb's J., 52–55.

Calder, B., Grunwald, D., Zorn, B., 1994. Quantifying behavioral differences between C and C++ programs. J. Program. Lang. 2 (4), 313–351, Technical Report CU-CS-698-94, Computer Science Department, University of Colorado, January 1994.

Chang, J.M., Gehringer, E.F., 1993. Evaluation of an object-caching coprocessor design for object-oriented systems. In: Proceedings of IEEE International Conference on Computer Design, 3–6 October 1993, pp. 132–139.

Chang, J.M., Gehringer, E.F., 1996. A high-performance memory allocator for object-oriented systems. IEEE Trans. Comp. (March), 357–366.

Chang, J.M., Srisa-an, W., Lo, C.D., 1999. Introduction to DMMX (Dynamic Memory Management Extension). In: Proceeding of ICCD Workshop on Hardware Support for Objects and Microarchitectures for Java, 10 October 1999, Austin, TX, pp. 11–14.

Chang, J.M., Srisa-an, W., Lo, C.D., 2000. Architectural support for dynamic memory management. In: Proceedings of IEEE International Conference on Computer Design, 17–20 September 2000, Austin, Texas, pp. 99–104.

Chung, C., Kim, S., 1998. A dualthreaded Java processor for Java multithreading. In: Proceedings of 1998 International Conference on Parallel and Distributed Systems, pp. 693–700.

Dieckmann, S., Hölzle, U., 1999. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99). In: Lecture Notes on Computer Science. Springer Verlag, Lisbon, Portugal.

Flux, 2002. Memory Debugging Utilities: liboskit_memdebug.a. Department of Computer Science, University of Utah. Available from http://www.cs.utah.edu/flux/oskit/html/oskit.html.

GNU, 2002. GNU Free Software Directory Project. Available from http://www.gnu.org/directory.

Hastings, R., Joyce, B., 1992. Purify: fast detection of memory leaks and access errors. In: Proceedings of Winter Usenix Conference, January 1992.

Javasoft, 2002. Available from http://www.javasoft.com, 2002.

Johnstone, M.S., 1997. Non-compacting memory allocation and real-time garbage collection. Ph.D. dissertation, The University of Texas at Austin, December 1997. Available from http://www.cs.utexas.edu/users/oops/papers.html.

Jhonstone, M.S., Wilson, P.R., 1998. The memory fragmentation problem: solved? In: International Symposium on Memory Management, October 1998. Vancouver, British Columbia, Canada, pp. 26–36.

Kaffe, 2002. Available from http://www.kaffe.org.

Larson, P.A., Krishnam, M., 1998. Memory allocation for long-running server applications. In: Proceedings of 1988 International Symposium on Memory Management, 1998, pp. 176–185.

Lea., D., 1996. A memory allocator (and from comments in source). Available from http://gee.cs.oswego.edu/dl/html/malloc.html, 1996.

Lo, C.D., Srisa-an, W., Chang, J.M., 1998. Boundary analysis for generalized buddy systems. In: 1998 Proceedings of International Computer Symposium, December 1998, Tainan, TAIWAN, pp. 96–103.

Lo, C.D., Srisa-an, W., Chang, J.M., 2000. Page replacement performance in garbage collection systems. In: Proceedings of 13th International Conference on Parallel and Distributed Computing Systems, 8–10 August 2000, Las Vegas, Nevada, pp. 374–379.

Lo, C.D., Srisa-an, W., Chang, J.M., 2001a. Performance analysis on the generalized buddy system. IEE Proc. Comp. Dig. Tech. 148 (4/5), 167–175.

Lo, C.D., Srisa-an, W., Chang, J.M., 2001b. A study of page replacement performance in garbage collection heap. J. Syst. Soft. 58, 235–245.

Pressman, R.S., 1997. Software Engineering: A practitioner's approach, fourth ed McGraw Hill. (pp. 614–617).

SPEC, 2002. Standard Performance Evaluation Corporation. Available from http://www.spec.org.

SPECjvm98, 1998. Standard Performance Evaluation Corporation. SPECjvm98 Documentation, Release 1.0. August 1998. Online version at http://www.spec.org/osg/jvm98/jvm98/doc/index.html.

Srisa-an, W., Lo, C.D., Chang, J.M., 1999. A hardware implementation of realloc function. In: Integration The VLSI Journal, vol. 28. Elsevier Science, pp. 173–184.

Wang, Y.C., Lin, K.J., 1998. Enhancing the real-time capacity of the Linux kernel. In: 5th International Workshop on Real-Time Computing Systems and Applications (RTCSA '98), 27–29 October 1998, Hiroshima, Japan.

Wilson, P., Johnstone, M., Neely, M., Boles, D., 1995. Dynamic storage allocation: a survey and critical review. In: Proceedings of 1995 International Workshop on Memory Management, 27–29 September 1995, Scotland, UK.

Chia-Tien Dan Lo received the Ph.D. degree in computer science from Illinois Institute of Technology, Chicago, the M.S. degree in Electrical Engineering from National Taiwan University, Taiwan and the B.S. degree in Applied Mathematics from National Chung-Hsing University, Taiwan in 2001, 1992 and 1990, respectively.

In 2002, he joined the department of computer science at University of Texas at San Antonio, Texas, as an assistant professor. From 1992 to 1994, he served as a second lieutenant for military and joined Data Automation Project Group for the Army. After 1994, he spent two years as a research assistant in Institute of Information Science, Academia Sinica, Taipei, Taiwan. His work there was implemented to a verifier that can be used in real-time concurrent systems. In 1995, he founded a computer company specialized in embedded system design, small business networking services and database applications. At the beginning of 1998, he received the Dean's scholarship and joined the computer system group at IIT. He started his teaching career in 1999 and courses he has taught include the Unix Systems Programming and the Computer Network.

His research interests include VLSI Design, Operating Systems, Neural Networks, Concurrent Algorithms, Computer Communication Networks, Artificial Intelligence, Computer Architecture, and Computing Theory. His current research thrusts are Dynamic Memory Management in Java, C and C++, Hardware Support for Java Virtual Machine, Reconfigurable Computing, Low-power Embedded Systems, Process Migration and Multithreaded Systems.

Witawas Srisa-an received the B.S. degree in Science and Technology in Context and M.S. degree in Computer Science from Illinois Institute of Technology. In 1999, he received the Dean's Scholarship to pursue his Ph.D. study and joined the Computer Systems Laboratory at IIT under the advisory of Dr. Morris Chang. He received in Ph.D. in Computer Science from Illinois Institute of Technology in 2002.

He is currently an assistant professor in the Department of Computer Science and Engineering at University of Nebraska at Lincoln. His courses include Operating System Kernels and Object-Oriented Instrumentation. His research interests include computer architecture, object-oriented programming, dynamic memory management and garbage collection, hardware support for garbage collection, Java, and C++ programming languages.

Ji-en Morris Chang received the B.S. degree in electrical engineering from Tatung Institute of Technology, Taiwan, the M.S. degree in electrical engineering and the Ph.D. degree in computer engineering from North Carolina State University in 1983, 1986 and 1993, respectively.

In 2001, Dr. Chang joined the Department of Electrical and Computer Engineering at Iowa State University where he is currently an Associate Professor. His industrial experience includes positions at Texas Instruments, Microelectronics Center of North Carolina, and AT&T Bell Laboratories, Allentown, Pennsylvania. He was on the faculty of the Department of Electrical Engineering at Rochester Institute of Technology, and the Department of Computer Science at Illinois Institute of Technology (IIT). In 1999, he received the IIT University Excellence in Teaching Award.

Dr. Chang's research interests include: Wireless Networks, Object-oriented Systems, Computer Architecture, and VLSI design and testing. His current research projects are supported by three NSF grants. He was a consultant for Kodak Inc. He has been a consultant for Toko America Inc. since 1997.

Dr. Chang has been serving on the technical committee of IEEE International ASIC Conference since 1994. He also served as the Secretary and Treasurer in 1995 and Vendor Liaison Chair in 1996 for the International ASIC Conference. He was the Conference Chair for the17th International Conference on Advanced Science and Technology (ICAST 2001), Chicago, Illinois, USA.