Estimating Internal Memory Fragmentation for Java Programs under the Binary Buddy Policy*

Therapon Skotiniotis

College of Computer Science
Northeastern University
Boston, MA 02115
skotthe@ccs.neu.edu

Morris J. Chang

Department of Electrical & Computer Engineering
Iowa State University
Ames, IA 5001
morris@iastate.edu

Abstract

Dynamic memory management has been an important part of a large class of computer programs and with the recent popularity of Object Oriented programming languages, more specifically Java, high performance dynamic memory management algorithms continue to be of great importance. In this paper, an analysis of Java programs, provided by the SPECjvm98 benchmark suite, and their behavior, as this relates to fragmentation, is performed. Based on this analysis, a new model is proposed which allows the estimation of the total internal fragmentation that Java systems will incur prior to the programs execution. The proposed model can also accommodate any variation of segregated lists implementation. A comparison with a previously introduced fragmentation model and with actual fragmentation values, is performed. The idea of a test-bed application that will use the proposed model to provide to programmers/developers the ability to know, prior to a programs execution, the fragmentation and memory utilization of their programs, is also introduced.

Index Terms: memory fragmentation, Buddy System, Java Virtual Machine.

1. Introduction

In the realm of the Java world, dynamic memory and the management of it, is an integral part of the language. Java, as an Object-Oriented language, as well as the Java Virtual Machine (JVM), make substantial usage of dynamic memory[19], elevating the importance of automatic memory storage and management in applications implemented in the Java language.

Automatic storage management does provide certain benefits to both the programmer and the system as, on the one hand, programmers do not have to spend time in explicitly managing their programs memory, and on the other, management of dynamic memory is ensured to be correct. From previously published work, and from experience, having to explicitly manage your program's dynamic memory needs, adds to the program's complexity[11] and increases the program's size in terms of lines of code[9]. As a result, programs tend to be harder to code, debug and manage.

With Garbage Collection being an integral part of the Java language, dynamic memory management is no longer an option to programmers, as it was with languages like C++. Zorn[11] clearly concluded that explicit management was falsely believed to perform better that automatic dynamic memory management, making its use more popular among developers.

Well known memory management policies[1] have been studied, modifications and measurements, on both the algorithms and their implementations, have been made so as to ensure which one policy brings about the best possible results. These studies are generally split into two categories. One category has to do with the analysis of memory management policies using synthetic traces. The second category refers to studies that were performed using actual real life programs.

Using synthetic traces alone, allows for inconsistencies as well as results that do not relate to real life situations, but allows for the analysis of long running, or deviating from normal behavior, programs. The second category of results, using actual programs, does reflect real life scenarios as closely as possible. This technique though brings about the problem of overspecification. With overspecification here we mean that the results drawn are specific to the application and programming language implementation. This may not apply to all programs implemented in that specific programming language.

Getting the best of both worlds, real life programs can be used to study their memory behavior. Based on these real life scenarios a mathematical model could be extracted that will be able to generalize results and as such allow the analysis of longer running systems with deviating behavior.

Java's platform independence has incorporated certain unique language features in order to abide to the "write once run anywhere" policy that the language supports. One of these features is the way that memory is packed, used and manipulated by the JVM so as to ensure proper execution of a

1.With the term policies here we refer to the strategy used to allocate / deallocate memory and not to the actual implementation.

* Partial support for this work was provided by the Nation Science Foundation through grand CCR-0098235

program regardless of the underlying platform.

Our decision to concentrate on the Buddy System group of policies is based on the following observations. First that the last attempt to analyze the performance of the Binary Buddies and provide a model with which one could approximate the policies behavior, was by Page and Hagins[1] in 1986, based on synthetic traces rather than real life programs. The study by Page but also later studies by Zorn et al. [11][12] showed overwhelming high memory fragmentation values for Binary Buddies (50%). As a result of these observations the policies have been thought of as memory wasteful, resulting to an underutilized memory system discouraging its use. Even though Binary Buddies provide one of the fastest allocation mechanisms [13][10].

Segregated lists, which Buddy Systems are a variation of [10], are used in the implementation of Transvirtual's JVM implementation Kaffe and Sun's JDK. Major differences being that Buddy systems allow for blocks equal to powers of 2, while segregated lists allow for any different denominations. This exponential spacing inhibited in Buddy systems allows for the greatest internal fragmentation between these two systems, and as such, can be used as an upper bound measure for internal fragmentation in our study.

Dieckmann and Holze's[2] recent study of Java's allocation behavior has brought to light some interesting facts concerning object sizes and request patterns, from which the Binary Buddy policies could benefit in order to achieve lower levels of fragmentation. Finally the fact that Binary Buddies have the fastest allocation/deallocation mechanism gives us good grounds to believe that, in the Java environment, the Buddy System policies could provide better overall performance[16][17][18], as well as, better memory utilization than what was generally believed in the past.

Through the analysis of the well founded group of programs provided by SPECjvm98, a mathematical model is then derived. The proposed mathematical model aims at providing an estimation of memory fragmentation that Java programs will incur without the need of an actual run of the application. That is, an estimate using the proposed model will simply require information obtained at compile time and/or after a control flow analysis on your program has been performed, with provided sample input. Information obtained from this specialized tool can then indicate fragmetnation values, but also blocks of code that heavily use your heap.

The next section provides related work on the subject, section 3 is concerned with the analysis and the extraction of program traces. Section 4 elaborates on the proposed mathematical model and section 5 concludes our work.

2. Related Work

Internal Fragmentation is essentially the extra unused memory that a policy or system will provide on top of the actual memory size that was requested by the program. The executing program, according to its memory needs, requests specific memory chunks so that it can proceed with its execution. Any system is thus faced with the problem of accommodating an unknown number, as well as sequence, of requests that a program will produce from a predefined fixed total memory that is available to the system. Over the years different policies have been introduced, each of which attempts to provide better memory usage based on two main criteria, performance and space usage.

Performance here refers to the speed of a policy's allocations. How fast the policy decides and allocates the memory chunk that will accommodate a program's request. Space usage looks at how effectively memory is being used for each allocation, but also, the policy's effectiveness for all allocations over the specified fixed memory space (i.e. a 40Mbytes of memory should be able to accommodate if not 40 close to 40Mbytes of requests).

Dynamic memory management policies that have been widely used and studied include first fit, best fit, buddy algorithms and segregated lists. Even though actual implementations may have been tailored so that to provide some extra optimizations, the allocation strategies are essentially the same and thus we will refrain from analyzing specific implementations of policies.

First-fit is the straightforward policy in which, from a list of available free locations the first one that is equal or greater than the program's request is allocated. In the case that the request is smaller than the available memory block we split the block and return any free space back to the free list. Best-Fit, will allocate the best possible fit from the list. Finally the Buddy policies, introduced by Knowlton [5], deal with free lists of blocks with sizes of 2^n. A request made by the system is rounded up to fit a block, any memory allocated in excess of the request is termed as internal fragmentation. In the case where a block of a specific size is no longer available, the search moves upwards to the next list of available block sizes where if an available block is found it is then split in halves. The process, if needed, is repeated until the required block size is attained and unused blocks are placed to their appropriate free lists. Other alterations of Buddy systems, Fibonacci buddy and weighted Double Buddies, were created with variations in their predefined block sizes so as to alleviate internal fragmentation. Segregated lists follow the same idea as Buddy systems but the block size for each list is predefined and unique (i.e. 2, 4, 6, 8 etc.) providing narrow spacing among block sizes.

From the definitions of the policies given above, one could deduce that both first-fit and best fit have a time penalty associated with their allocation technique, the penalty of traversing a list in order to find the appropriate block of memory to assign. Buddy systems, as well as segregated lists, provide faster allocation but suffer from internal fragmentation. Buddy systems being the worst due to the wider spacing of block sizes than segregated lists.

In [14] a comparison was made, using a synthetic tool, among these policies. The simulation was based on probabilistic distributions of object size as well as lifetimes with an end result of first-fit outperforming best-fit, under Knuth's optimizations, and buddy systems comparing to this results. More recent comparisons by [14],[12] indicated that the buddy system policies consistently outperformed the rest of the policies in speed, yet used the most space out of all of the policies tested.

Poor memory utilization in the buddy system policies was discussed by [3][4] but it was until [1] introduced a new alteration to the Buddy System policies, as well as mathematical formula with which one could calculate the mean internal fragmentation of programs. Page et. al.[1] based the derivation of their formula on a close analysis of the buddy system policy and then using synthetic traces validated, both the new Double Buddy policy, as well as their mathematical formula.

Throughout the literature on the subject, but more specifically [10][13], it has been shown that even though synthetic traces are more easily manageable and can provide a general picture of a policy's behavior at a high level, they need to be correlated with actual program traces. Most performance evaluations have been performed on real life applications under real life systems in order to provide the exact behavior of the system and programs in question. Actual traces are finite and thus cannot provide information for the behavior of long running programs. Achieving the best of both worlds, a synthetic trace that can be generated by a model derived from actual program traces will be the closest approximation to real life scenarios under the least possible error due to simulation.

Zorn and Grunwald [13] have performed an evaluation of memory allocation models of real life C/C++ programs using the technique described above. Measurements of speed, memory utilization as well as fragmentation were reported, as well as the validity of different established mathematical models was checked against their empirical data. Their attempt was to show the validity of the models against the extracted data. There was no attempt to develop a new mathematical model based on the information collected by their test programs.

A recent study by Dieckmann and Holze [2] on the allocation behavior of the SPECjvm98 [6] benchmark programs has brought to light some interesting results. Allocation patterns, and their understanding, is of great importance to memory management since these patterns dictate the behavior of a program's/system's memory needs. These patterns have a direct dependence both on the programs that are executed but also more generally on language characteristics, this is the reason why many different researchers have independently studied this topic in many different languages.

In this paper a study of the fragmentation behavior of Java programs under the buddy system policy is performed. A mathematical model is derived from the extracted data of the Java benchmark programs found in SPECjvm98. The proposed model allows for an estimation of total internal fragmentation to be obtain at compile time and/or after a control flow analysis step, that will only identify which object sizes have the two highest request frequencies. A specialized tool can be build to obtain object frequencies but also report back on blocks. of code and their corresponding memory usage. These pieces of information can then be used to alter the code in order to achieve higher stability, reliability as well as program efficiency.

3. Benchmarks and Experimental Results

All of the results and measurements presented in this paper are based on the programs provided by the SPECjvm98 benchmark suite. SPECjvm98 was released in August 1998 by the System Performance Evaluation Corporation (SPEC)[6].

Each of the benchmark programs attacks one or more of the criteria that determine the efficiency of a JVM's implementation. A short description of the programs found in SPECjvm98 can be found in Table I

TABLE I DESCRIPTIONS OF THE SPECJVM98 BENCHMARK PROGRAMS

Program	*Description*
compress	Compress/decompress program based on Lempel-Ziv method
db	Performs multiple database functions on memory resident database
jack	A Java parser generator that is based on the Purdue compiler construction tolls set (early JavaCC)
javac	The JDK 1.0.2 Java compiler compiling 225,000 lines of code
jess	A Java expert shell system based on NASAs CLIPS expert system
mpegaudio	An ISO MPEG Layer-3 audio decoder
mtrt	A dual-threaded raytracer that works on a scene depicting a dinosaur

3.1 Program Traces and Behavior

In order to obtain traces from the execution of all of the SPEC programs a version of the JVM (1.2.2)[1] was instrumented so as to spit out information relevant to the allocation of objects. Object sizes were recorded as well as the unique object handler for each allocation. In order to capture the exact behavior of the JVM a record of garbage collection cycles was also extracted.

A simulator was build that mimics a memory management system that abides by the Buddy system allocation scheme. The trace output, obtained from the execution of the SPEC benchmark programs, was used as the input to our simulator. The simulator then produced a detailed trace of internal fragmetnation, memory usage as well as total heap size used by the Buddy System policy.

Acquiring the object allocation information from running the instrumented version of the JVM, one can look into the object allocation patterns. Allocation patterns here relate to object sizes and the frequency with which a specific object size is being requested by the executing program. Results are shown in Table II and Fig. 1 Requests as well as total memory allocated vary widely between the benchmark programs. This gives rise to the different scales that can be observed in Fig. 1 Still all of the programs under investigation show a very high frequency for object sizes between the range of 4 bytes to 512

1. We used the publicly available source distribution for JDK 1.2.2 running on an Intel Pentium III 600 MHz,256Mbytes of memory, Red Hat 6.2 [7] as our system test bed for both the trace extraction process as well as the simulation run of the programs.

bytes. Notable exceptions to the rule are *Compress*, *Db* and *Jack*, all of which exhibit high frequencies for larger object sizes as well.

Jack, for example, displays a unique feature of a pretty stable frequency for objects of sizes between 868 bytes to 1732 bytes. Jack being a compiler's compiler in SPECjvm98 it is actually given as input the source to generate itself. The task to generate a parser deals with acquiring definitions of tokens and a grammar's production rules. Based on these definitions the program has to then examine valid tokens, from a stream of characters, and attempt to group the tokens using the grammar's production rules. Jack has to then create the code for all the appropriate data structures, symbol table, abstract syntax tree but also blocks of code that will perform the basic lexical and semantic analysis. These blocks of code are created from predefined code within Jack but also from code that will be taken verbatim from the input file. Action rules for a grammar are an example. As a result of the above scenario Jack has to create blocks of code upon which it will have to append verbatim from the input, assemble the pieces together and then produce the resulting source code.

Db displays a high frequency of requests for large object sizes 40,000 to 61,000 bytes in size. Since it is a database system, the underlying database is queried for information. Entries in the database are removed and added at random, the sizes of both additions and subtractions vary widely. Having to manipulate large objects through multiple queries, the database requires large objects to hold and manipulate intermediate steps of queries that need to be buffered. This results in to abnormally high frequencies of both small and large objects during its execution.

Compress shows a high frequency of requests when it comes to object sizes of 6000 bytes and above. Common patterns found in the file are replaced by one smaller pattern enabling size compression of the input. These patterns can span through the file which now resides in multiple internal buffers. Replacing the patterns and appending this internal buffers together force the application to request objects of relatively large size.

Excluding the irregularities of the three programs described above, one can see that generally the distribution graphs approximate to an exponential decay function. Displaying yet again another observation made by [9] that the majority of objects in Object Oriented languages tend to be of a relatively small size. A similar scenario can be observed here with the exception that small size objects are considered to be the ones with size less that 1Kbyte rather than 4K as observed in [9]. Another point of interest is the fact that for all the programs tested with SPECjvm98 the request sizes of objects are all a multiple of 4. The reason behind this being the packaging of memory that the Java language as well as the JVM implementation imposes on all programs. As mentioned in [8] a JVM implementation has a specified size for a word (4 byte words). Having a word size specified, any object whether user defined or system[1] defined is rounded up, padded, so that it reaches the nearest word boundary.

TABLE II ALLOCATION DATA FOR SPECJVM98 PROGRAMS

Program	Class file size (Kbytes)	Total Requests (Mbytes)	Total Allocations[a] (Mbytes)	Number of requests
compress	17.4	105	159	11397
db	9.9	60	76	3215561
jack	129.4	135	217	6871327
javac	548.3	140	204	5943357
jess	387.2	210	295	7939418
mpegaudio	117.4	0.924	1.44	14963
mtrt	56.5	84	115	6643962

a. The total size of allocations under the Binary Buddy allocation policy

In this way data are stored in contiguous packets, making memory addressing easy for accessing but also ensuring platform compatibility.

3.2 Fragmentation

Fig. 2 displays the information generated by our simulator, given as input the program traces from SPECjvm98. Fragmentation was calculated as the difference between the assigned memory block and the requested memory block.

Displayed, for each program, we can see the total fragmentation incurred at 100 request intervals. Peaks found on each graph denote garbage collection cycles that remove garbage objects, and as such, the fragmentation incurred from these garbage objects was subtracted from the system's total internal fragmentation. In this way we could depict the real values for fragmentation as they would be incurred by a real life system.

Compress shows a high increase in its fragmentation towards the end of its execution. This can be attributed to the final steps of the compression/decompression algorithm where intermediate buffers are concatenated together to result to the final output file (compressed or decompressed). *Db* as a database system performs repeated operations on its data moving around small as well as large objects. The repeated peaks in the graph show how intermediate manipulation of tables as well as files affect the memory requests of the database system.

Jack with the second highest number of requests from the rest of the programs displays more fluctuations to its internal memory fragmentation. Compiler generation steps are performed during which data structures are being updated and temporary files are created. At the end of each compiler generation step garbage is collected. Approaching the end of its execution larger intermediate generated blocks of code are being handled explaining the gradual increase in the programs total internal fragmentation.

1.By system here we refer to libraries and not to predefined types as byte, float, char, even though, with the exception of byte, all the rest are constructed based on the word size imposed by the JVM implementation

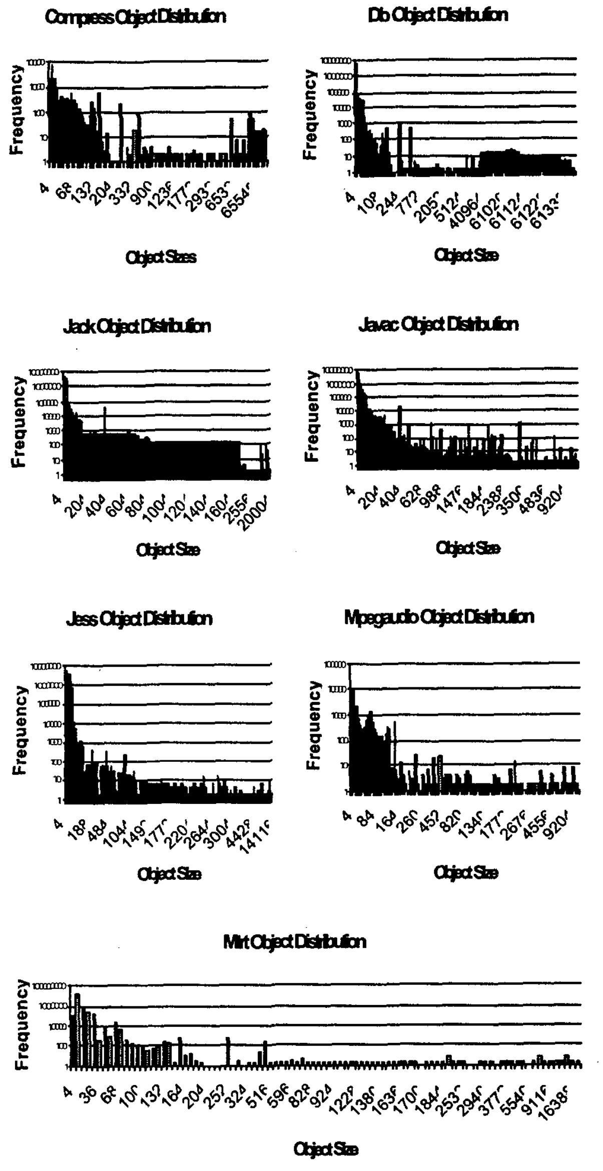

Fig. 1 Frequency Vs Object Size

On a similar vein *Javac*, goes through its own compilation steps displaying a similar behavior as Jack, only this time with fewer requests, the graph is more sparse.

Jess, the highest in object requests, iterates through its set of rules and data in order to solve a number of puzzles. Each rule, within a loop, validates itself against the puzzle data. This consistent and continuous checking causes requests and garbage to be created by each check, explaining the high number of requests but also the continuous rapid fluctuations due to garbage collection cycles.

As another means of fragmentation measurements, researchers in the past[1][12] have also reported the percentage of fragmentation a system has suffered. This percentage measurement is essentially the fraction of total internal fragmentation over total memory used by a program during execution. Fig. 3 shows the percentage of memory fragmentation for all benchmark programs after every 100 allocations.

With the exception of *Compress* and *Mpegaudio* all other programs display a percentage of fragmentation in the range of 35 to 31 percent. *Mtrt* having an even lower percentage (29% - 27%) for the majority of the programs execution time.

Compress and Mpegaudio do have a higher percentage of fragmentation (37% - 40%) throughout their execution. Compress, due to the program's high requests in large object sizes, in relation to the wider spacing of the Buddy System's allocation blocks at high powers of *n*, the percentage of fragmentation tends to be at its highest values. Mpegaudio having the smallest total memory usage, gives a higher percentage of fragmentation even though the program displays similar object size requests as most of the other benchmark programs.

4. Models of Fragmentation

In this section previous work on models of fragmentation for the Binary Buddy system is analyzed as well as a new model is proposed. Work, by Page [1], is examined in order to show its validity in the context of the Java Language using the SPECjvm98 benchmarks. Also, the steps taken to derive the proposed equation are shown.

4.1 Page's Model

The equation used by Page[1] for estimating the mean internal fragmentation was

$$Frag = \sum_{r = r_{min}}^{r_{max}} p(r) \times F(r)$$

$$b_i \geq r > b_{i-1} \quad F(r) = b_i - r$$

where $F(r)$ denotes the fragmentation incurred by the request of size *r*. Simply the difference in size between the allocated block (b_i) of the Buddy System Policy and the actual requested size (r). $p(r)$ denotes the probability of request *r* occurring.From the equation we can state that fragmentation depends on the number of requests that do not fit exactly our block (b_i) but also how frequent these cases are.

Looking first at request size, the bigger the spacing between the Buddy blocks the greater the fragmentation incurred by a request that does not fit our block size exactly. Second the frequency of these requests will have a linear relation with the value of fragmentation that will be incurred by the memory system.

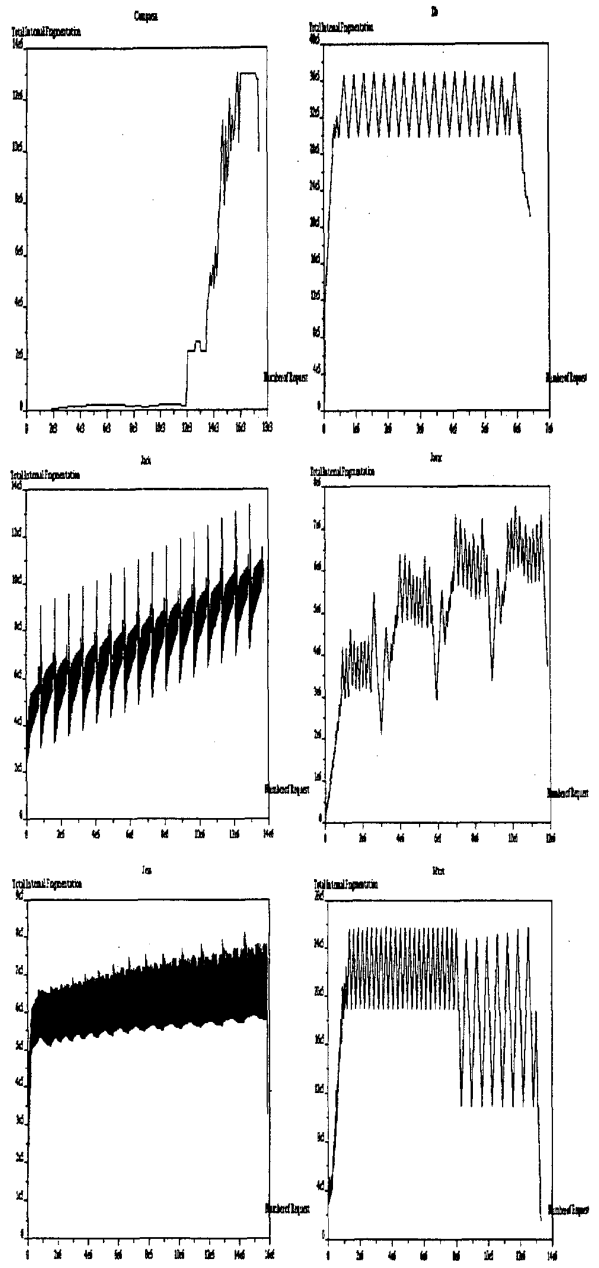
Fig. 2 Fragmentation Vs Requests

Calculating internal fragmentation for each allocation can be directly inferred from the definition of internal fragmentation,($F(r) = b_i - r$). As for the distribution of requests, since here we are talking about the mean internal fragmentation, a probability function ($p(r)$) is used to capture the behavior of a program's request patterns. Essentially, one can plug-in the request distribution that abides to the behavior of a program, whether that would be an exponential or normal distribution, for example, and estimate the mean internal fragmentation.

Page's model accommodates, through the probability function, any possible program behavior. In order to do so though one needs to first acquire a full program trace, i.e., for each object size you should obtain the total number of requests.

4.2 Calculating Total Internal Fragmentation

Java programs did show a decreasing exponential distribution of requests (section 3.). Also, all of the requests made were of a multiple of 4 in size. Starting from scratch we have

$$TotalFrag = \sum_{r_{min}}^{r_{max}} f(r) \times Frag(r) \tag{1}$$

$$Frag(r) = (b_i - r)^{b_i \geq r > b_{i-1}}$$

where*f(r)* denotes the result of the frequency function value at request size r. *Frag(r)*can be tailored, in this case, to calculate more effectively the fragmetnation that each allocation will incur. We thus alter the equation to

$$Frag(n) = (b_i - (4 \times n)) \tag{2}$$

where $\{b_i \geq (4 \times n) > b_{i-1}\}$

In this way we are generalizing the fact that all request will be a multiple of 4 and that the Binary Buddy block assigned to it will have the relation as described in *(1)*. Java programs did display, in their majority, an exponential decrease behavior of the form.

$$f(r) = 10^{c^{\left(\frac{1}{r}\right)}} \tag{3}$$

The choice of the constant c is the value of the object size, which holds the highest frequency of requests for a programs execution. In this way the equation will mimic the programs distribution function so as to define the highest starting peak of the inverse exponential distribution of request sizes. Using *(2)* and *(3)* we can rewrite *(1)*to give

$$TotalFrag = \sum_{n=1}^{n_{max}} 10^{c^{\left(\frac{1}{r}\right)}} \times (b_i - (4 \times n)) \tag{4}$$

The variable n in equation *(4)* can be altered to mimic the sequence of request sizes made by the programs (all of which start from 4), simplifying the equation to

$$TotalFrag = \sum_{r=4}^{r_{max}} 10^{c^{\left(\frac{1}{r}\right)}} \times (b_i - r) \tag{5}$$

In order to approximate the total internal fragmetnation of a programs execution an extra, constant value, needs to be introduced. This extra value, will facilitate a tuning mechanism by which one can approximate the programs value for total internal fragmentation.

$$TotalFrag = \sum_{r=4}^{r_{max}} k \times 10^{c^{\left(\frac{1}{r}\right)}} \times (b_i - r) \qquad (6)$$

The constant k found in *(6)* will be, essentially, the most common factor of fragmentation that the system incurs throughout its execution. That is the fragmentation values that are most common, highest frequencies, from all the requests made by the program.

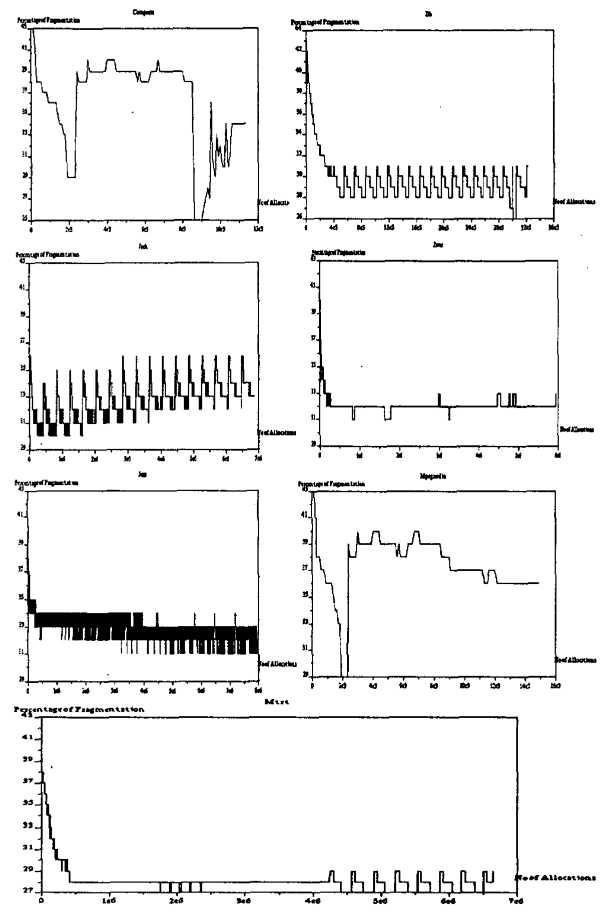

Fig. 3 Percentage of Memory Fragmentation Vs Number of Allocations

From the analysis of the results, we have found that the most appropriate value for k can be determined from the 2 most dominant request sizes which introduce a fragmentation value greater than 0. Adding the internal fragmentation, that is introduced by the 2 dominant request sizes, will provide the best possible value for k.

As an example, from Jack's allocation trace, the dominant object sizes are 4, 12 and 36 with their corresponding allocation percentages, over total allocations, being 33%, 32%, 22% respectively. For allocations of size 4 the system will not incur any fragmentation but for 12 and 36 we will have 4 and 28 bytes of fragmentation respectively, the sum of which (32), is the value for k to be.

Looking at Table III, we can see that the proposed model does provide a better estimation for 3 out of the 6 programs (Jack, Jess, Javac). The estimations made for Compress and Db show a large percentage of error for the proposed model. As it was discussed in 3.1 above, both of these two programs exhibit an allocation pattern that is different from the allocation patterns of the other programs under investigation. As a result the distribution model that was used (exponential decay) does not approximate the actual object distribution.

Estimations made on Mtrt, using the proposed model, are still close to the actual values for fragmentation even though Page's model provides a better approximation. Reasons for this behavior can be attributed to Mtrt's very low frequency of requests for object sizes more than 512 bytes. The number of these different object sizes greater that 512 bytes being also large allows calculation of mean values to be more precise than exponential.

4.3 Observations

The two factors concerned with fragmentation are object sizes and the frequency of their requests. The determining factors for an object's size are: first, the sizes that are assigned to primitive types, second the different combinations of the primitive types that a program will use in order to create user defined types and other data structures.

Frequency of requests correlates to the physical positioning of allocation calls within the source code. Object sizes can be calculated at compile time, something that is already done by a compiler, for example, in order to calculate sizes of Activation Records for methods. Frequency of requests is more dynamic, as the control flow of the program generally depends on user input or other real time environment variables of the program. Ideally, the above proposed model could be used within a specialized application that would be able to take as input a program's source and input parameter values in order to perform an execution analysis. That is, object sizes would be calculated as well as a refined control flow analysis will be performed in order to extract information concerning the frequency of object requests.

Actual and accurate frequency numbers are not needed for the proposed model. Instead, for the constant c the object size with the highest request, and for the constant k the 2 objects sizes with the highest request that contribute towards internal fragmetnation. Therefore, an aggregate estimation performed after compile time by this specialized tool, could extract these pieces of information. An estimation of the total internal fragmentation can then be obtained.

These, estimated values, in collaboration with the software metric formulas as presented in [15] can provide software performance results at an earlier stage in the software development cycle. Furthermore, memory footprints could be estimated at compile time, as well as the ability to identify better memory watermark points for garbage collection to yield higher memory utilization.

TABLE III COMPARISON OF RESULTS BETWEEN PAGE'S MODEL, PROPOSED MODEL AND ACTUAL FRAGMENTATION.

Programs	Constant c	Constant k	Actual Fragmentation (Mbytes)[a]	Page's result (Mbytes)	Proposed Model (Mbytes)	Percentage Error of Proposed Model	Percentage Error of Page's Model
Compress	12	16	54	53.56	72	33	**0.8**
Db	12	16	15	15.75	21	16	**5**
Jack	4	32	73	68.71	71.2	**2.5**	5.9
Javac	12	16	64	59.43	62.8	**1.8**	7.1
Jess	20	32	83	78	87	**4.6**	6
Mtrt	12	16	31	30.56	29.91	3.5	**1.4**

a. Actual Fragmentation refers to the total fragmentation incurred by the program excluding garbage collection cycles.

The majority of todays JVM implementations make an extensive use of segregated lists implementations. To accommodate a system that uses segregated lists instead of Buddy systems the following alteration to our model needs to be made.

$$TotalFrag = \sum_{r=4}^{r_{max}} k \times {10^{c}}^{\left(\frac{1}{r}\right)} \times (b_i - r)$$

$$b_i \geq r > b_{i-1}, \text{and } b_i \in \{s_0 s_1, \ldots, s_n\}$$

where the set $\{s_0 s_1, \ldots, s_n\}$ denotes the set of the predefined block size of each free list (in the case of Kaffe this will be {16,24,32,40,48, etc.}).

5. Conclusion

This work has provided a study of fragmentation of the JVM under the Binary Buddy system using the well founded set of applications provided by SPEC. Actual fragmentation values under the Binary Buddy system were also given, showing a 37% fragmentation rather than 50% as it was previously believed. Based on our analysis we have then presented a new model with which one can calculate the total internal fragmentation that the JVM will incur. We have shown that the proposed model can be also applied to a segregated lists implementation, one of the most popular choices among the latest implementations of popular JVMs. With the assistance of a specialized application, our proposed model can be used to provide run time information at an earlier stage of the applications development cycle, at compile time. This information can then be used to provide metric values for the application's efficiency, reliability and stability[15].

6. References

[1] Ivor P. Page and Jeff Hagins, "Improving the Performance of Buddy Systems", *IEEE Transactions on Computers* VOL C-35,NO 5. 1986

[2] S. Dieckmann and U. Holzle. "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmark", *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, Lisbon, June 1999, Springer Verlag

[3] P.W.Purdon and S. M. Stigler, "Statistical Properties of the Buddy System", *Journal of the ASM*, Vol. 17 pp.683-697, Oct. 1970

[4] J. L. Peterson, "Dynamic Storage Allocation with Buddy Systems", *Proc 4th. Texas Conf. Comp. Architect*. Nov. 1975 pp. 2B4-1 - 2B4-6.

[5] K.C. Knowlton. "A Fast Storage Allocator", *Communications of the ACM*, vol. 8. pp.623-625. Oct. 1965

[6] *Standard Performance Evaluation Corporation.* URL http://www.spec.org/osg/jvm98

[7] *Red Hat Linux.* URL http://www.redhat.com

[8] *The Java Virtual Machine Specification.* URL http://ava.sun.com/docs/books/vmspec/index.html

[9] Doug Lea. A Memory Allocator. (and from comments in source).

[10] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review", *Proc. 1995 International Workshop on Memory Management*, Kinross Scotland,UK,September 27-29, Spinger Verlag

[11] B. Zorn,"The Measured Cost of Conservative Garbage Collection".*Software Practise and Experience*, Vol.23, No.7: 733-756, July 1993

[12] D. Detlefs, A. Dosser, B. Zorn,"Memory Allocation Cost in Large C and C++ Programs". *Software Practise and Experience*, 24(6):527-542, 1994

[13] B. Zorn, D Grunwald, "Evaluating Models of Memory Allocation". *ACM Transactions on Modeling and Computer Simulation*, 4(1), 1994

[14] David G. Korn and Kiem-Phong V. "In search of a better malloc". *Proceedings of the Summer 1985 USENIX Conference*, pages 489-506, 1985

[15] R. S. Pressman. *Software Engineering: A Practitioner's Approach*, pages 448-517. McGraw-Hill, 4th edition, 1997.

[16] *The DaCapo Project.* URL http://www-ali.cs.umass.edu/DaCapo/index.html

[17] C. D. Lo, W. Srisa-an and J. M. Chang, "Performance Analysis on the Generalized Buddy System", to appear in *IEE Computers and Digital Techniques Journal*, 2002

[18] J. M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems" *IEEE Transactions on Computers*. March, 1996. pp. 357-366

[19] Q. Yang, W. Srisa-an, T. Skotiniotis, J. M. Chang. "A Cycle-accurate Per-thread Timer for Linux Operating System".To appear in the *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS-2001), Tucson, Arizona. Nov.4-6, 2001.