Real-Time Memory Management: Life and Times

Andrew Borg and Andy Wellings
University of York, UK
{aborg,andy}@cs.york.ac.uk

Christopher Gill and Ron K. Cytron
Washington University in St. Louis, MO, USA
{cdgill,cytron}@cse.wustl.edu

Abstract

As real-time and embedded systems become increasingly large and complex, the traditional strictly static approach to memory management begins to prove untenable. The challenge is to provide a dynamic memory model that guarantees tight and bounded time and space requirements without overburdening the developer with memory concerns. This paper provides an analysis of memory management approaches in order to characterise the tradeoffs across three semantic domains: space, time and a characterisation of memory usage information such as the lifetime of objects. A unified approach to distinguishing the merits of each memory model highlights the relationship across these three domains, thereby identifying the class of applications that benefit from targeting a particular model. Crucially, an initial investigation of this relationship identifies the direction future research must take in order to address the requirements of the next generation of complex embedded systems. Some initial suggestions are made in this regard and the memory model proposed in the Real-Time Specification for Java is evaluated in this context.

1 Introduction

Memory management is a major concern when developing real-time and embedded applications. The unpredictability of memory allocation and deallocation has resulted in hard real-time systems being necessarily static. Even when the analytical worst-case space and time requirements can be derived [28, 27], the high space and time overheads of traditional dynamic memory models such as fine grain explicit memory management and real-time garbage collection (GC) mean that it is hard to argue a case for these models in resource constrained environments. Alternative memory models such as that proposed in the Real-Time Specification for Java (RTSJ) can be shown to reduce these overheads but at a cost of significant development complexity. Identifying and characterising the criteria used to gauge different memory models is non-trivial. Arguing that one memory model is better than another must be qualified by a specification of the driving design forces. One important element of these driving forces is the application development cost of the chosen model. GC is the best choice in this regard as the additional development cost of using memory dynamically is small. However, the real-time community remains highly polarised as to whether real-time GC can ever reach the predictability and efficiency required by these systems. Critics of GC technologies argue that the space/time tradeoffs achievable by real-time garbage collectors are still unsuitable for the class of systems targeted by the static paradigm. In order to achieve the required performance in one semantic domain, the costs in the other are too high. On the other hand, critics of new memory models such as that proposed in the RTSJ argue that the burden of memory concerns placed on developers detracts from what is arguably the most attractive feature of developing large and complex systems in environments such as Java: freedom from memory concerns. The common ground between these two camps is that the success or failure of providing a suitable environment for developing the next generation of complex real-time and embedded systems hinges on getting the memory model right.

The motivation for this work comes from the recent debate about which memory model the RTSJ should adopt. Indeed, this continues to be the most contentious issue of the RTSJ. Java developers could adopt a static approach in the same way as the SPARK [5] but even the Ravenscar profile for Java [20] which also targets high-integrity systems recognises the need for a dynamic model by introducing a limited region-based memory model. The need for dynamic memory is however not a requirement only of real-time Java itself but is a requirement of the next generation of embedded real-time systems. The goals of this paper are threefold: (1) to identify the implications of using different dynamic memory models; (2) to motivate a new approach to developing dynamic memory models that goes beyond the "fine-tuning" approach of current research; (3) to show why the RTSJ's memory model may be a step in the right direction in this regard. However, it will also be argued that a number of changes to the model and the way it is used are necessary.

To this end, this paper provides three contributions. First, an analysis of fine grain memory management approaches from previous work is carried out in Section 2 in order to identify and quantify the space and time overheads of these approaches. Although the results described here make a strong

Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06)
0-7695-2619-5 /06 $20.00 © 2006 IEEE

case for an alternative memory model, current research either fails to provide models for which the overheads are sufficiently tight or proposes models which place a significant burden on the developer. Section 3 describes the second contribution of this paper, a novel, unified approach to distinguishing the merits of each memory model that is based on a hypothesis that highlights the relationship across three semantic domains: space, time and a characterisation of available information about the way memory is used by the application, such as the average size of object or a characterisation of their lifetime. This allows the derivation of a classification of memory models, thereby identifying the class of applications that benefit from targeting each model. The hypothesis developed in this section is used to make a case for the direction research needs to take in order to develop memory models suitable for the next generation of complex embedded systems. In particular, it is argued that rather than trying to adapt the memory models used in traditional environments to real-time ones, new memory models that directly address real-time requirements need to be developed. Section 4 describes the third contribution in this paper: an overview of a detailed investigation into the RTSJ memory model that has been carried out in order to analyse whether this model captures the shift in research strategy argued to be necessary in Section 3. Although it is argued that this is achieved to some degree, it can also be shown that the model's abstraction fails to allow an expression of common lifetime patterns or restricts the ability of this information to be expressed. Also, even when lifetime information can be expressed, implementations often fail to take full advantage of this information. Solutions to address this are briefly described. Finally, Section 5 identifies future work and concludes.

2 Fine Grain Models in RT Environments

The static approach to memory management is the traditional way of developing hard real-time systems. Assuming an object-oriented language, the application would allocate all necessary objects in a pre-mission phase and these would live for the duration of the mission phase. If memory constrained devices are the target of the application, developers are often forced to consider recycling objects, thereby adding another dimension of complexity to the development and maintenance processes. In some cases, this problem is aggravated as the object model may need to be broken in order to allow what would ordinarily be type-incompatible objects to replace each other. *Dynamic memory allocation and deallocation can address this problem if the reduced development complexity does not come at an unacceptable space and time overhead.*

Dynamic memory management is used in most development environments outside the real-time domain in the form of fine grain models whereby (assuming an object-oriented environment), memory allocation and deallocation is carried out at the granularity of a single object. In these models, allocation is explicit whereas deallocation can take the form of user-controlled or automatic deallocation by a garbage collector. A large amount of research effort has been invested in deriving algorithms for fine grain dynamic memory models. It is unsurprising therefore that recent research focuses for the most part on porting these models to the real-time domain. However, in proposing the use of these memory models in a real-time domain, the space and time overheads need to be quantified. This is investigated in the rest of this section and subsequently used to motivate research into an alternative memory model.[1] The sources of overheads in dynamic fine grain models are broadly as follows:

1. Both explicit and automatic memory models introduce space and time overheads due to fragmentation.
2. In automatic memory models, there is the additional overhead of identifying garbage.
3. If memory management is made to be incremental, an additional overhead for guaranteeing the mutual integrity of the program and collector is incurred.
4. In order to counter (1), both explicit and automatic memory models can use defragmentation. This approach merely shifts fragmentation overhead from the space domain to the time domain. Defragmentation is reassessed in Section 3.3 when discussing the Metronome collector and not discussed further in this section.

2.1 The Cost of Memory Fragmentation

Memory fragmentation introduces high pessimism and unpredictability in space and time requirements that is unfavourable in real-time environments [21]. An investigation into the results from past work is carried out next in order to quantify these overheads.

Measuring the Cost of Fragmentation: Space

A detailed investigation into the space overhead due to fragmentation in a number of Dynamic Memory allocation algorithms (DM algorithms) can be found in previous work by Neely [25] and Johnstone [17]. These results are intriguing in that they show that the observed fragmentation is least in the simpler policies such as first and best fit as opposed to more complex policies such as buddy algorithms. These simpler algorithms exhibit fragmentation that is also very low, typically under 3%[2] when averaged across all applications but rises to as high as 53% in more complex algorithms such as binary buddy. When taking into consideration the implementation costs of the policy (such as the data structures maintaining free lists) and machine requirements (namely byte alignment), these overheads increase to just 34% for

[1] No two page report can do justice to the large amount of research carried out in this area and the goal here is to only identify and loosely quantify the source of overheads. A good survey can be found in [32].

[2] $F = \frac{MaxHeapSizeAtMaxLiveBytes - MaxLiveBytes}{MaxHeapSizeAtMaxLiveBytes}$.

best-fit and first-fit policies and 74% for binary buddy. A conclusion that Johnstone draws from these results is that the fragmentation problem is solved, and has been solved for several decades. However, Johnstone's work is based on the observed rather than analytical worst case space requirements of applications. In a series of work between 1971 and 1977 [29, 30, 28], Robson derived the worst case memory requirements of the best-fit and first-fit policies whereas Knowlton [19] derived in 1965 the worst case for buddy systems. Given a maximum block size n and maximum live memory requirements of M, the worst case memory requirements are $M \log_2 n$ for first-fit, $2M \log_2 n$ for binary buddy and Mn for best-fit. Luby *et. al.* [22] showed that the worst case requirements for some segregated policies is similar to that for first-fit. Crucially, Robson showed that there exists an optimal strategy that lies between $\frac{1}{2}M \log_2 n$ and $0.84M \log_2 n$. The first-fit policy therefore provides a solution that is close to optimal. An interesting exercise is to compare these results to Johnstone's for observed fragmentation. The work of Johnstone and Robson would lead one to conclude that the first-fit policy is the best solution both in terms of observed and analytical worst case overheads. In fact, the most significant observation that can be drawn from a comparison of Johnstone's and Robson's work is that there exists a large discrepancy between the observed and analytical worst cases for all policies. For example, a program with a maximum live memory requirements of 1Mb and which allocates objects that range over a conservative size (say between 64 and 64k bytes) would still require 10Mb to guarantee against breakdown due to fragmentation when using first-fit and 20Mb when using the binary buddy algorithm. In addition to this, the memory requirements of the mechanism implementing the policy must also be considered. These total memory requirements are a significant order of magnitude higher than the 1.5Mb to 2Mb requirements one would expect in the observed worst case.

Measuring the Cost of Fragmentation: Time

An important contribution in the analysis of the time overheads incurred by DM algorithms was provided recently by Puaut in [27]. Puaut analysed the average and worst observed times of four applications using a number of different DM algorithms and compared them to the analytical worst case allocation and deallocation times of these algorithms. The average observed time overheads are similar across all DM algorithms and moreover, the worst case overheads are typically less in the simpler policies such as best-fit than in the more complex ones such as binary and Fibonacci buddy. The analytical worst case overheads however tell a different story. Here, the analytical worst case performance of best-fit and first-fit DM algorithms using a naïve mechanism is nearly a thousand times worse than that of the buddy systems which performs best for the analytical worst case. The significant time used by a DM algorithm in a typical program are immediately evident: the values for b-tree best fit and binary buddy would be respectively around 64% and 63% time overhead for a logic optimisation program called Espr; that is about 63% and 64% of program execution is used in servicing allocation and deallocation requests. In the worst case, these values jump to 96% and 71% respectively.

2.2 The Cost of (Non-Incremental) GC

When considering automatic memory management, it is often implied that the garbage collector also assumes the role of the DM algorithm and defragmentor, thereby executing four tasks: servicing allocations, locating garbage through a root scan and traversal of the object graph (*tracing*), freeing garbage (*sweeping*) and defragmentation. This blurs the distinction between DM algorithms and garbage collectors and limits a direct comparison between explicit allocation and deallocation memory models and automatic memory management models. In an effort to quantify each overhead, this paper maintains the distinction between the processes of allocation, tracing, sweeping and defragmenting memory. There is an additional overhead in a garbage collected environment over an explicitly managed one serviced directly by a DM algorithm that is highlighted by this abstraction: the time overhead involved in tracing that is not present when memory is managed explicitly and the space overhead due to the delayed deallocation of memory. Given this additional overhead, it would be expected that GC would automatically imply higher total overheads than an explicit memory management, in both the space and time domain. There are several cases in the literature in which this is argued not to be the case in the time domain [7, 15]. This phenomenon occurs because the delayed deallocation of objects in a garbage collected environment results in higher space overheads but incurs lower time overheads due to infrequent vertical switching between the application and underlying memory subsystem. However, existing garbage collectors make use of strategies that are absent in existing DM algorithms but that could be readily implemented. For example, a similar technique to reduce vertical switching could be used for explicit memory management with *free()* calls being delayed and a single call to the DM algorithm passing the addresses of all memory to be freed. Garbage collectors will therefore always incur additional overheads over explicit fine grain models due to tracing. This is an important observation as the cost of tracing becomes the single additional overhead between explicitly managed memory models and non-incremental automatic memory management models. The results in [2] for the Metronome collector show that tracing incurs the highest cost of all collector operations, including fragmentation. Section 3.3 revisits these results in detail.

2.3 The Cost of Incremental Collection

Irrespective of whether a work-based [4] or schedule-based [1] approach to real-time collection is adopted, the ad-

ditional time cost of an incremental approach over a stop-the-world one comes from one primary source: maintaining consistency between the mutator and collector through the execution of barriers. Quantifying this overhead is often difficult as the work done at each increment involves the execution of other tasks such as tracing and defragmentation. Since these overheads are being treated independently, the cost of a barrier here is considered only to be the cost of maintaining a suitable consistency between the mutator's view of the object graph and the actual object graph. In [33], Zorn shows that the cost of read barriers alone can incur a 20% penalty on application performance when executed in software though Cheng *et. al.* claim that their Metronome garbage collector can reduce this to 4% on average and 9% in the worst case. Considering that a read barrier based on pointer updates can be implemented with a handful of operations (an average four ALU/branch instructions in [33] and a compare, branch and load in [1]), these significant overheads are caused by the large number of times these barriers are executed.

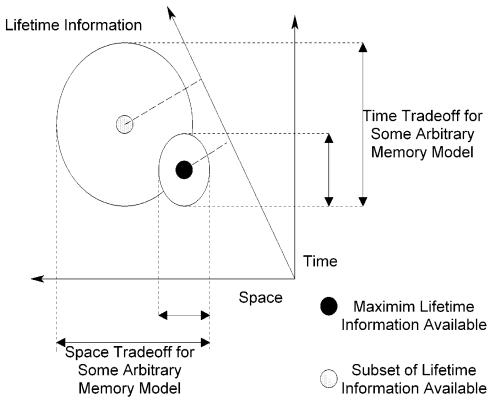

Figure 1. Space/Time in the Entropy Hypothesis

3 Towards a Classification of Memory Models

Although the worst case space and time overheads for fine grain models described in the previous section are clearly too high for resource-constrained environments, making the case for an alternative model is not easy. Crucially, it is unclear what direction research needs to take in order to develop these models. Although significant research effort has been invested in this area, particularly in the field of real-time collectors, the returns have been minor. This section introduces a novel way of classifying memory models that allows a direct comparison between them to be derived and also highlights the application classes for which suitable memory models still need to be developed. This comparison is based on an evaluation metric that consists of three parameters: **time overheads**, **space overheads** and **an expression of memory usage information**.

The relationship between space and time, although not always trivial, is in general described by a function in which an increase in overheads in one domain tends to result in a decrease in the other. The choice of whether to use a defragmentation algorithm is an example of this. The third parameter introduced in this evaluation metric captures the burden placed on the developer in describing the known information about how objects are used in the application. For example in an explicit fine grain model, this information is an expression of the lifetime of each object as specified by the *malloc()* and *free()* operations. Other models such as the Metronome collector discussed in Section 3.3 allows the expression of other information such as the average object size. Typically, a memory model is compared to another only in the space and time domains. The burden of describing the third parameter is rarely qualified in the traditional fine grain approaches introduced in Section 2, in all probability because explicit and automatic approaches at this granularity describe two extremes that are easy to characterise. In this section, it is argued that this burden as captured by the expression of this information is not independent of the space and time dimensions; rather it defines them.

3.1 Memory Management as an Entropy Problem

A description of the interrelation between the parameters of the evaluation metric can be argued by the *Entropy Hypothesis* which we propose. The Entropy Hypothesis states that the relationships between space, time and object information can be characterised as a form of entropy. We borrow the concept of entropy from Information Systems Theory [31] to which a parallel can be drawn. In information systems, entropy is a measure that is used to calculate the amount of information in a source or, equivalently, the *redundancy* in that source. The entropy gives a measure of the actual information in a system and dictates the maximum degree to which that system can be compressed and thereby the number of bits required to transmit that source. Every information source has a *maximum entropy* that sets a lower bound for the compression of that source through lossless algorithms. When a system is said to be at maximum entropy, it is implied that it exhibits maximum randomness or, equivalently, no information is known about the information source and lossless compression is impossible. However, if certain information is known about the source (i.e it is not completely random), then this can be used by a compression algorithm to reduce the number of bits required for transmission.

Our analysis of memory management techniques according to entropy is based on the hypothesis that the amount of available information about an application defines the space and time overhead domains of the application. Furthermore, just as known information of patterns in an information source

can be used to reduce the cost of transmission, so information about memory usage can be used to reduce the space and time overheads of memory management. Also, using the notion of maximum entropy, maximum randomness in an information source that results in high transmission costs can be compared to high space and time requirements in application execution. For a given amount of information about an application, a solution is defined in a memory/time trade-off space for which an independent function that depends on the chosen memory model will define the tradeoff in the time and space domains. The tradeoff between these domains is a potentially unique signature for that model and can identify at a fine granularity a ranking of models for a particular application class.[3] However, the bounds of this space are defined by the entropy of that information: with a given amount of information, there is a bound on the solution trade-off space. The Entropy Hypothesis is depicted in Figure 1 for a hypothetical application and memory model.[4] The available information about the application defines the space of memory/time trade-offs which memory models can achieve. If less information is available, the space and time requirements could increase but are always bounded from below by the space for which maximum information is available.

The Entropy Hypothesis thus defines a more abstract view of the memory management problem and the function of a memory model and hence motivates a more abstract definition of what a memory model is and the functions it performs. *A memory model is defined as a mechanism that allows expression of knowledge of memory usage and takes advantage of this knowledge in order to reduce time and space overheads.* The goals of a memory model are therefore twofold: providing a mechanism that allows this knowledge to be expressed and taking advantage of this knowledge. In promoting any memory model, the importance of making this knowledge as easy to obtain and express as possible cannot be overstated. This is particularly true of real-time environments as the guarantees the memory model can provide are directly related to the accuracy and precision of this knowledge. The ability to take advantage of this knowledge is equally important and can be used to give a comparative assessment of different implementations of the same model.

Using the Entropy Hypothesis, the investigation into the overheads of fine grain memory models described in detail in the previous section and the well-known overheads of the static approach, a spectrum of memory management technologies that describe the resultant overhead of some of these models can be defined. This spectrum is shown in Figure 2 where the space and time overheads of each solution are described in relation to the amount of information that is expressible and used by the memory model. This is based on

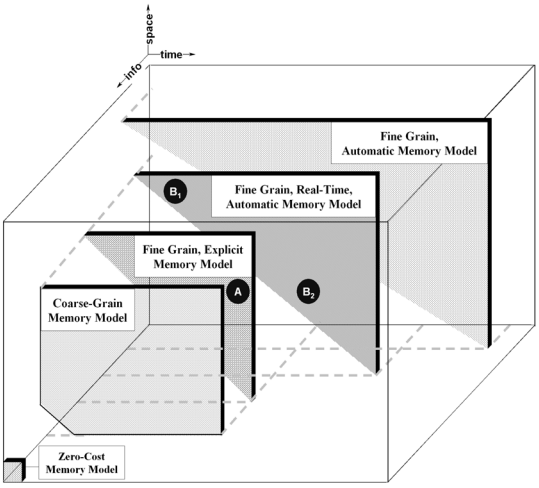

Figure 2. Classifying Memory Models

a representative hypothetical application of non-trivial complexity and may vary for other applications. The shape of the space is likely not as well defined in practice as the figure would lead one to believe, but for illustrative purposes the tradeoff in space and time is captured here by the triangle shape. In any case, an exact function describing this tradeoff is rarely available. Another simplifying assumption made here is that the general application under investigation terminates. If this were not the case then space requirements could be infinite, thereby removing the top horizontal edge. Finally, although the axis of the graph are left voluntarily general, it is the analytical worst-case overheads for the application that are considered here.

A hypothetical zero-cost memory model that incurs zero space and time overheads is shown in Figure 2. This is tantamount to there being complete knowledge of the application's memory usage available, which is leveraged in the implementation of the adopted memory model. Most models can in practice achieve very small space or time overheads but rarely can minimise both. For example, a fine grain model can minimise space overheads by defragmenting at every deallocation but then the time overheads are high. At the other extreme, near-zero time overheads can also be achieved at high space costs by never deallocating objects.[5] The argument made in Section 2.2 that explicit memory management can always be made to perform at least as well as automatic memory management is captured by the Entropy Hypothesis; the exact lifetime of objects is unknown before runtime and therefore the space/time tradeoff must be worse. The arguments made in [7, 15] where garbage collectors are argued to perform better than explicit allocation and deallocation are then captured

[3]The assumption here is that the tradeoff is defined for the worst-case values of the respective domains. There is a significant complexity in defining this function that is not addressed here.

[4]Note that the oval shape is not necessarily indicative of the true shape of this space.

[5]This is never exactly zero due to the time required to update the pointer to free memory.

by the tradeoff function. For example, in Figure 2, the points A vs. B_1 and B_2 represent the space and time costs for an application using an explicit memory model vs. an automatic memory model respectively. Clearly, the automatic memory model is more time efficient but less space efficient, at B_1. However, to achieve the same space overheads as A at B_2 the additional cost of tracing guarantees that a higher time overhead is incurred than in the explicit model. A case for the depicted location of coarse grain models in Figure 2 is made in Section 4. In these models, allocation and deallocation is carried out on a group of objects at one time rather than on a single object as in a fine grain model. Memory pools are an example of such models.

A caveat of the Entropy Hypothesis is that the decision of which memory model to use must be made before implementation of the application begins. For example, if an explicit coarse grain model such as a memory pool model is to be used, then the developer must identify appropriate clusterings to place inside each pool. In time-critical applications, consideration must also be given to the timing requirements of tasks. In quantifying the development costs of a chosen model and the corresponding expected time and space overheads, an important assumption is made: *the developer must target that memory model.* Targeting a memory model plays an important part in leveraging the advantages of that model. For example, if a memory-pool model is provided, then programming with a fine grain approach by placing one object in each pool will fail to achieve the reduced time overheads of the coarse grain approach. The importance of providing the right abstractions to capture this information is crucial. *The success of bringing dynamic memory models to constrained real-time environments must therefore lie in providing the right abstractions to capture information about memory usage in the application.*

The Entropy Hypothesis hints at the direction future research must take in order to fill the gap between the static approach and traditional fine grain models. In the absence of a proven lower-bound for the space-time tradeoff of existing memory models, there are two possible research directions that can be taken to improve on the static approach: invest further in refining existing fine grain models or derive alternative models that target regions of the trade-off space that fall between the static and dynamic approaches. The bulk of current research takes the former approach. For example the work in [24] describes an DM algorithm with constant worst case execution time which is shown to be lower than that of several other algorithms. The worst case space overheads due to fragmentation however are of order Mn, a value worse than that derived by Luby [22] using an algorithm that employs a segregated-fit policy which searches for a better fit. In both cases, the information expressed in these two models are identical and therefore, as would be expected by the entropy hypothesis, one only sees a space/time tradeoff between them. The Entropy Hypothesis makes a case for research to adopt the second research direction: developing memory models to allow more information to be expressed in order to reduce space and time overheads. Four questions that future research must therefore address are:

1. What types of information can be captured?
2. Which single/combination types of information best allows the shift towards the hypothetical zero cost model?
3. What is the best way to capture this information so as to place the least possible burden on the developer?
4. What is the best way to make use of the different types of information that can be captured?

The first of these questions is addressed next. The overheads for the Metronome collector are describe in Section 3.3 and, together with the analysis carried out in Section 2, is used to motivate a move towards allowing the expression of new types of information. In Section 4, coarse grain memory models and the RTSJ memory model are evaluated in order to show how this approach can achieve lower overheads. The third and fourth questions are answered in relation to the RTSJ memory model.

3.2 Information in Object-Based Systems

Information about objects in an application can be expressed either for a specific application or a more general application class. The difference between general and application-specific knowledge is mainly one of tuning in the adopted strategy; general information is used to define a memory model's basic strategy whereas application-specific information is used as a parameter to refine this strategy. General and application-specific information can be further decomposed into local and global information. In this case, the difference is the granularity for which the information is specified. At one extreme, fine grain allocation and deallocation using *malloc()* and *free()* operators is local information. At the other extreme, the information in the parameters described for real-time collectors are global to the whole program. Coarse grain models such as memory pools lie between these two extremes with aggregates being specified to define the lifetime of objects.

In order to justify a particular choice of expressing information of memory usage, a quantifying relationship would be desirable. The case for application-specific information being preferred over general information is clear when the worst case has to be considered; an application that does not fit a general model is often easy to develop and such an application will perform poorly. Nevertheless, the majority of the research into memory management has gone into identifying optimisations for the common case of the "average" application. For example, in Section 2.1, the term "strategies" was shown to have been coined by Neely and Johnstone [25, 16] to

describe how minor modifications to policy such as when coalescing occurs and choosing the block size could be used to significantly improve the average performance of a DM algorithm. The case for local as opposed to global information is more difficult to argue than that for application-specific versus global information as this implies a significant development burden. It would also be beneficial to be able to quantify the relationship between the same type of information. For example, it would be useful to be able to state that the accuracy of one example of global and application-specific information (for example the maximum allocation rate) is more important than another (for example the average object size). This would allows developers to identify the strategies required to satisfy the space and time restrictions of their application. Although no attempt is made in this paper to identify all possible ways of expressing information, a quick analysis can pinpoint the better options that address the major disadvantages of the fine grain explicit and automatic memory models identified in the previous chapter. The merits of coarse grain memory models can be argued by the development complexity of capturing the information required by these models and the space and time overheads achievable using this information.

Approximating development cost to the burden of expressing information about memory-related behaviour blurs the distinction between man-hour costs incurred through manually expressing this information and the time and monetary cost of tools that discover this information automatically. In most cases, automatic techniques are used for gathering application-specific information at compile-time.This is favoured by the developer as the development cost (by the definition given above) is offloaded from a man-hour cost into the cost of the tools. Tools for automatically calculating global parameters for fine-tuning garbage collectors are also common [23]. The most common automated way of obtaining local and application-specific information is through escape analysis [12]. This is typically used to supplement automatic fine grain models by reducing the heap size and therefore the time for a collection. Automated approaches for collecting information could be equally useful in developing real-time applications as in non real-time ones. As a pre-deployment investment, they can provide benefits insofar as they output conservative results that encompass the worst case. It is noted in passing that in some cases, the available information can be used to derive other information. For example, the maximum allocation rate of an application – a global piece of information – can be derived using analysis of the locations of explicit allocation and deallocation calls – a local piece of information.

Another consideration is that some information of memory usage may only become available at runtime. For example, it may be the case that although the exact lifetime of an object may not be known at compile time, this information becomes known at the time the object is allocated at runtime. In this particular example, it is known that this information is of little benefit to an allocation strategy as it can be shown that any allocation strategy that attempts to use this information is guaranteed to perform almost as bad as an allocation strategy where this information is unknown [18]. However, this result can not be generalised to all types of information.

Bench	m	s	T_{GC}	T_I	T_R	T_M	T_S	T_D
javac	86	172	2.21	0.001	0.061	1.973	0.137	0.124
db	82	137	2.63	0.001	0.043	2.408	0.148	0.163
jack	82	146	1.73	0.001	0.042	1.076	0.094	0.047
mtrt	80	122	1.59	0.001	0.046	1.386	0.115	0.078
jess	73	126	0.63	0.001	0.186	0.554	0.046	0.031
fragger	72	151	3.20	0.001	0.147	1.700	0.175	1.295

Table 1. Time and Space Overheads at 50% Utilisation in Metronome (Taken from [3])

3.3 The Metronome Collector

The Metronome collector is a time-based collector that uses a best-fit policy implemented with a segregated free list mechanism in its DM algorithm and an incremental mark-sweep collector that defragments when required. A read barrier is implemented to ensure moved objects are properly referenced by the application. In motivating the time-based approach of the Metronome collector [1, 3, 2], Bacon *et. al.* derive an analysis that allows a guarantee of the minimum mutator time. For a given time period in the application, the mutator and collector have two properties specified: for the mutator, the allocation rate over a time interval and the maximum live memory usage and; for the collector, the rate at which memory can be traced. By defining the frequency of invocation of the collector, the memory required for a given utilisation requirement is derived. Alternatively, the maximum available memory is specified and the minimum guaranteed utilisation is derived.

The results from Metronome are of particular use to this investigation as the cost of each of the four processes of garbage collection are broken down and quantified. Ignoring allocation costs, collection overheads are broken down into the costs of initialisation and termination of the collector (T_I), root scanning (T_R), marking (T_M), sweeping (T_S)and defragmentation (T_D). For a 50% utilisation across the selection of applications from the SPECjvm98 benchmarking suite, the time overheads (in seconds) are reproduced in Table 1 where the amount of live memory (m) and the maximum heap size (s) in Mb are also shown. It is interesting to note that the majority of the collection cycle is spent tracing, with the time taken for defragmentation being significant only in the "fragger" application.

The results from the research for Metronome initially appear promising. By requiring the user to specify information about the pattern of object usage, the space and time over-

heads are significantly smaller than those shown to hold in the worst case for an explicit model using a DM algorithm. The information that needs to be specified includes the average object size and locality in the size. These parameters are used to reduce the pessimism in the worst case overheads incurred during tracing and defragmentation. Tools for automatically calculating these parameters for fine-tuning garbage collectors are becoming more common [23]. The magnitude of the overheads of the DM algorithm due to fragmentation described in the last section in comparison to the results achieved here require further investigation in relation to the Entropy Hypothesis. The space overheads are as little as two and half times the amount of live memory. This is achieved by capturing the pessimism of the worst-case fragmentation through a factor λ that specifies the locality of size of objects.

Although the cost of tracing and using an incremental approach are still significant, Metronome's λ factor appears to address the fragmentation problem for its DM algorithm. This research therefore provides not only a real-time collector with tighter space and time overheads but, more importantly, a solution that can be applied to explicit fine grain models for use in more resource constrained environments. This could therefore fill the gap between the environments real-time collectors address and the those addressed by the static approach. The λ factor allows the developer to capture Johnstone's thesis that fragmentation in real-world systems is negligible. However, there are two problems with this approach: firstly, identifying the λ factor for an application is non trivial; secondly, the analysis does not capture the possible variance of this value during the application's lifetime. Therefore, the chosen λ factor will always be the smallest value during the entire lifetime of the application. This leads to a number of assumptions in the derivation of the worst-case space requirements for a given minimum mutator utilisation that inhibit a true calculation of the worst case space requirements. Crucially, the space-time relationship between heap size and utilisation is not well defined due to the interdependence of parameters leading to undesirable recursive functions. In particular, the amount of extra space required depends on the time needed for a collection cycle which in turn depends on the amount of heap space. The chosen or derived heap size is based on an "expansion factor" of the maximum amount of live memory that is chosen to be around 2.5 based on experimentation. In the absence of the pessimistic values that would be obtained from a recursive relationship, the results given in [1, 3] are essentially observed rather than analytical quantities.

4 The Case for Coarse Grain Memory Models

Whereas it could be argued that existing fine grain models that guarantee space and time bounds fulfil real-time requirements, the overheads of these approaches may make this prohibitive in resource-constrained environments. From the evaluation of fine grain approaches in Sections 2 and 3.3, it is immediately apparent that the most urgent information required is that which addresses fragmentation and the cost of tracing. The Metronome collector may fail in the former case only because most applications have fluctuating characteristics such as allocation and fragmentation rates that are not well captured as global information. It fails in the latter case because it does not allow the user to express where objects are known to have become garbage. Furthermore, the specification of the lifetime of objects by the developer at a coarse granularity becomes feasible where this is increasingly less practical in fine grain models. It thereby becomes possible to leverage the benefits of existing work on offline allocation strategies as well as to consider new strategies such as the scoped ordering proposed in the RTSJ. Therefore, the solution to the memory management problem for resource-constrained real-time systems could lie in an explicit model (thereby eliminating the need for tracing) and directing research at a more local characterisation of the application's information, particularly fragmentation. The Entropy Hypothesis argues the case for more information to be expressible and for this information to then be used by that model. The key problem is identifying what this information is and how it can be captured. Rather than arguing for similar global, application-specific information to be used in these models, a case for more localised information can be made. For example, global parameters could be made more localised in Metronome by being sensitive to the program's flow. Therefore, rather than there being just one integer λ factor, a number of values could be assigned that depend on the current execution trace. These values become points on the flow graph of the application and new analysis would be required to identify how the transition between these points changes the behaviour of the collector. The feasibility of such a characterisation is unclear, both in terms of identifying this function as well as how this fits into the scheduling model. Until such research is available, an alternative is available in the form of coarse grain memory models.

Explicit Coarse Grain Models as a Solution to Fragmentation and Tracing

The cause of fragmentation in DM algorithms that adopt online allocation policies is the irregular arrival pattern of allocation and deallocation requests [17]. The assumption of a random arrival pattern results in worst case space requirements that are often too large for constrained real-time and embedded systems [29, 30, 28]. However, Johnstone's research [16, 17] shows that regularities in arrival patterns are often implicit in most applications. Coarse grain models provide a mechanism to capture this phenomenon in such a way that deterministic guarantees can be provided on the space and time requirements of the application while taking advantage of reduced overheads demonstrated in the observed case. In particular, coarse grain models take advantage of the phenomenon that it is often possible to aggregate objects according to their lifetime based on some boundary criteria. For ex-

ample, objects that are created close together typically have similar lifetimes due to spatial locality of reference [6]. For a real-time environment, a more application-specific and local approach is required as accuracy and precision are necessary to provide the required guarantees with low pessimism.

The benefits of a coarse grain approach are leveraged when the aggregation of object lifetime can be captured without additional development complexity. Therefore, since coarse grain models involve first specifying the boundaries of aggregates, the identification of these boundaries must not be prohibitively complicated. For example, if a memory pool is used whereby every object is explicitly placed in a pool based only on knowledge of the exact lifetime of that object and without any discernable underlying pattern, then targeting this model provides little advantage in terms of development complexity over using an offline algorithm that requires the exact lifetime of individual objects to be specified.[6] This is an example of an application targeting the wrong memory model as discussed in Section 3.1.

Coarse grain models are not by definition explicit memory models as the deallocation point of regions could also be identified by automatic techniques. However, the reduced number of elements that need to be contended with in comparison to a fine grain model means that an explicit approach to specifying the lifetime of regions is a feasible option. In this case, tracing is therefore no longer required to identify objects that can be deallocated, thereby elimination the overheads of this operation. However, an explicit memory model has repercussions on the safety of an application in the event that the lifetime of an object specified at compile-time does not match that of the object at runtime. The garbage collection approach is to guarantee that this problem never occurs. The RTSJ's approach is to employ the scoped reference rules to ensure no reference can be created that could later lead to a dangling pointer. An alternative solution is to eliminate the runtime overheads of these two approaches and provide a failure mechanism in the event that an object is prematurely deallocated when the information specified on the lifetime of that object is discovered to be incorrect at runtime. A mechanism that achieves this in the RTSJ is described in [9].

Making Use of the Relative Lifetime of Regions
Information available at compile-time on object lifetimes is rarely taken advantage of in explicit, fine grain memory models. Indeed, most compilers go through great lengths to optimise for performance but few optimise for space [14, 13]. The results of research into polynomial offline algorithms are rarely carried over to real-world compile-time optimisation, firstly because, as noted by Johnstone, such optimisation is rarely necessary and secondly because specifying the lifetime of every object allocated in the application is complex in practice and undecidable in general by analysis techniques [12]. However, the results of this research could prove beneficial to real-time applications if capturing this information is made more practical. A coarse grain memory model can leverage these benefits as specifying the lifetime of coarse grain entities incurs a burden on the developer that is significantly smaller than in a fine grain model. This reverses the current trend away from explicit models and towards automatic ones that incur high time overheads.

[6] In practice, the choice of a memory pool model could also be based on cache considerations.

4.1 The RTSJ Memory Model: Criticism

The RTSJ adopts a novel approach to memory management with the introduction of scoped regions. This model is essentially a coarse grain model that aggregates object lifetime based on program flow. The main criticisms of this model are broadly as follows:

The model is complex to use:
The complexity of the RTSJ model could be partially argued to be a failure of developers to target the model. As argued in Section 3.1, developers must target a memory model rather than apply an orthogonal abstraction to the chosen model. Since the RTSJ defines object aggregates based on locality in the program flow, developers must express lifetime information around this abstraction. However, it is often the case that the real-world pattern of memory usage does not follow this approach. For example, applications that employ a producer/consumer pattern of memory usage are hard to describe in the RTSJ as the implicit information in this application is not well captured by the scoped memory abstraction.

Reference rules inhibit the expression of object lifetime:
A second source of the complexity in using the RTSJ's memory model comes from the model's reference rules. Despite an object's lifetime clearly belonging to some aggregate, these rules require a change in the lifetime of objects based on the reference graph. Restricting the flexibility of how aggregates are defined is an example of a memory model unnecessarily restricting the expression of known lifetime information.

The possibility of reusing code is limited:
The reuse problem of RTSJ code is caused by the embedding of memory concerns within application code. The absence of an interface that captures how the memory model is used in existing classes means that there is no way to export the lifetime of objects created in this code.

Lifetime information is poorly utilised:
Although the RTSJ specifies when the backing store of regions are allocated and freed in the runtime, no constraints on the underlying DM algorithm is specified. In particular, most implementations do not take advantage of the scoping

order. Also developers are often unaware of the implications of where aggregate boundaries are applied.

4.2 The RTSJ Memory Model: Solutions

The model is complex to use:
In order to address this problem, researchers have provided solutions that are orthogonal to the RTSJ scoped memory model only because the model fails to allow these patterns to be expressed. For example, Pizlo [26] proposes "wedge threads" that are used to keep a scoped region alive when its reference count would have otherwise dropped to zero. Although the introduction of these patterns highlights the complexity of using the scoped memory model, forcing a solution on top of the existing RTSJ model has earned them the term "anti-patterns". There are two possible conclusions that can be drawn from this: either the RTSJ needs to be extended to allow these patterns to be expressed as an integral part of the model or the way the model should be targeted is still not understood. The second possibility is improbable as the model's rationale is intuitive. Earlier work we have carried out [11, 10, 8] shows that in translating the same information present in an explicit coarse grain model to a scoped model results in a tradeoff space that can reduce space and time overheads is some cases but can lead to potentially unbounded space requirements in others. This conclusion is important as it makes a strong case for the RTSJ to provide alternative memory models in addition to scoped memory that allow these patterns of object lifetime to be expressed. The RTSJ scoped memory model can express some patterns of object lifetime better than other models and is therefore useful when these patterns are manifested in the real-world. When this is not the case, the RTSJ must provide other approaches that allow the expression of object lifetimes that the patterns such as those described in [26] address. Describing these patterns as an abstraction on top of the RTSJ scoped model is a poor approach. In conclusion therefore, the Entropy Hypothesis can be used to argue that the RTSJ model is complex only when it is used to express information that is poorly captured by its abstraction. The solution is therefore not an alternative model but an extended model that allow a wider range of patterns of memory usage to be expressed.

Reference rules inhibit the expression of object lifetime:
The problem of expressing some aggregate lifetime patterns in the RTSJ as described above is compounded by the RTSJ making it harder to define these aggregates. We have developed reference objects [9] in order to address this. Using a reference object rather than a normal reference achieves a compromise between maintaining the safety of objects and allowing this lifetime information to be expressed. Reference objects carry lifetime information that is looser than that specified by a regular reference. In the RTSJ, if an object A holds a reference to an object B then B *must* live as long as A. However, if A holds a reference object to an object B then B *can* live as long as A but the reference object must throw a caught exception if this is shown not to be the case at runtime. Reference objects are an example of allowing the developer to specify lifetime information in order to reduce space overheads.

The possibility of reusing code is limited:
This issue comes back to the question: "What is the best way of allowing known information to be expressed without placing unnecessary burden on developers?" If a memory model can allow an equivalent expression of this information externally to application code then code reuse is again possible. We have developed a solution to this problem as part of our work [11, 10, 8] that is based on extracting the cross-cutting memory management concern as a separate aspect. This is achieved by defining the boundaries and lifetime of aggregates on the program's control flow graph. An algorithm then finds the optimal scoping order and annotates the application to enter and exit regions when specified.

Lifetime information is poorly utilised:
The second role of the memory model, taking advantage of expressed memory usage, is also under-specified in the RTSJ. The underlying DM algorithm allocates and frees regions in a similar way to explicit allocation and deallocation, thereby resulting in similar fragmentation problems. In particular, the advantage of reduced fragmentation due to scoping is lost in multithreaded environments as the lifetimes of aggregates across different threads of control are unspecified. A solution to this problem is to have separate partitions for each branch of the scope stack when this can be statically determined to be possible. When this is not possible, the model experiences similar fragmentation to fine grain models if the variance in the sizes of regions is large. This therefore partly eliminates the rationale for a scoped approach. Again, the inability to define separate partitions for scope stacks is an example of how, unavailable information leads to higher overheads. In this case, a simple analysis of the variance of region sizes can be used to merge regions of similar lifetime so that fragmentation can be reduced.

5 Conclusion

The search for suitable memory models that address the requirements of complex embedded real-time systems continues to gain momentum. The choice of a suitable memory model for the RTSJ is viewed as a contentious issue by many, particularly where a choice between real-time GC and scoped memory must be made. The Entropy Hypothesis shows that an argument for some memory model is not absolute to a particular domain, whether that domain is defined in terms of allowable space and time overheads or development costs. Rather, a memory model is suitable for a given application only in the degree to which it can capture information of

memory usage in the application. The goal of future research must therefore lie in identifying this information and providing ways of allowing this information to be expressed in order for the underlying memory subsystem to make use of it. This is a significant shift from current research directions that deliver only marginal improvements due to the implicit assumption that expressing lifetime information implies unnecessary burdens on application developers.

References

[1] D. Bacon, P. Cheng, and V. T. Rajan. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, pages 285–298. ACM Press, January 2003.

[2] D. Bacon, P. Cheng, and V. T. Rajan. The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems. In R. Meersman and Z. Tari, editors, *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)*, volume LNCS 2889 of *Lecture Notes in Computer Science*, pages 466–478, 2003.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling Fragmentation and Space Consumption in the Metronome, a Real-Time Garbage Collector for Java. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems (LCTES 03)*, pages 81–92. ACM Press, June 2003.

[4] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[5] J. Barnes and J. G. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[6] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[7] H. J. Boehm. Memory Allocation Myths and Half-Truths. Available at: http://www.hpl.hp.com/personal/Hans_Boehm/gc/myths.ps.

[8] A. Borg. On The Development of Dynamic Real-Time Applications in the RTSJ - A Model for Expressing Dynamic Memory Requirements . Technical Report YCS-2004-379, University of York, June 2004.

[9] A. Borg and A. Wellings. Reference Objects for RTSJ Memory Areas. In R. Meersman and Z. Tari, editors, *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)*, volume LNCS 2889 of *Lecture Notes in Computer Science*, pages 397–410, 2003.

[10] A. Borg and A. Wellings. Towards an Understanding of the Expressive Power of the RTSJ Scoped Memory Model. In R. Meersman, Z. Tari, and A. Corsaro, editors, *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '04)*, volume LNCS 3292 of *Lecture Notes in Computer Science*, pages 315–332, 2004.

[11] A. Borg and A. Wellings. Scoped, Coarse-Grain Memory Management and the RTSJ Scoped Memory Model in the Development of Real-Time Applications. *To Appear: International Journal of Embedded Systems*, 2006.

[12] A. Deutsch. On the Complexity of Escape Analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 358–371. ACM Press, January 1997.

[13] J. Fabri. Automatic Storage Optimization. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction (SIGPLAN '79)*, pages 83–91, 1979.

[14] J. Gergov. Algorithms for Compile-Time Memory Optimization. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*, pages 907–908, January 1999.

[15] M. Hertz and E. Berger. Automatic vs. Explicit Memory Management: Settling the Performance Debate. Technical Report CS TR-04-17, University of Massachussetts, Department of Computer Science, 2004.

[16] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, December 1997.

[17] M. S. Johnstone and P. R. Wilson. The Memory Fragmentation Problem: Solved? *SIGPLAN Notices*, 34(3):26–36, 1999.

[18] B. Kalyanasundaram and K. Pruhs. Dynamic Spectrum Allocation: The Impotency of Duration Notification. In S. Kapoor and S. Prasad, editors, *Procedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science, (FST TCS '00)*, volume LNCS 1974 of *Lecture Notes in Computer Science*, pages 421–428, 2000.

[19] K. C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[20] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: a High-integrity Profile for Real-time Java. *Concurrency and Computation: Practice and Experience*, 17(5-6):681–713, April 2005.

[21] J. Lindblad. Reducing Memory Fragmentation. *Embedded Systems Engineering*, pages 26–28, April 2004.

[22] M. G. Luby, J. S. Naor, and A. Orda. Tight Bounds for Dynamic Storage Allocation. *SIAM Journal on Discrete Mathematics*, 9(1):155–166, 1996.

[23] T. Mann, M. Deters, R. LeGrand, and R. K. Cytron. Static Determination of Allocation Rates to Support Real-Time Garbage Collection. *SIGPLAN Notices*, 40(7):193–202, 2005.

[24] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *16th Euromicro Conference on Real-Time Systems (ECRTS 04)*, pages 79–88, 2004.

[25] M. S. Neely. An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation. Master's thesis, The University of Texas at Austin, May 1996.

[26] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-Time Java Scoped Memory: Design Patterns and Semantics. In *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 101–110. IEEE Computer Society, May 2004.

[27] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS'02)* , 2002.

[28] J. Robson. Worst Case Fragmentation of First-fit and Best-fit Storage Allocation Strategies. *The Computer Journal*, 20(3):242–244, 1977.

[29] J. M. Robson. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *Journal of the ACM*, 18(3):416–423, 1971.

[30] J. M. Robson. Bounds for Some Functions Concerning Dynamic Storage Allocation. *Journal of the ACM*, 21(3):491–499, 1974.

[31] C. E. Shannon and W. Weaver. The Mathematical Theory of Communication. University of Illinois Press, 1949.

[32] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management (IWMM '95)*, pages 1–116. Springer-Verlag, 1995.

[33] B. Zorn. Barrier Methods for Garbage Collection. Technical Report CU-CS-494-90, University of Colorado at Boulder, 1990.

0-7695-2619-5 /06 $20.00 © 2006 IEEE