Predicting Application Memory Allocation Behavior

Project Report

Christopher J. Hazard
North Carolina State University

April 21, 2005

1. Introduction

Computer architecture has always been rapidly changing. New architectures are invented, techniques are refined, and new problems need to be addressed. Memory latency has been one of the biggest hurdles recently in advancing the throughput of computer processors. Processor architecture technologies continue to advance, but the amount of CPU cycles wasted waiting on memory has been continuously climbing. Because of this, SMT (simultaneous multi-threading, also known as Hyper-threading), and CMP (chip multi-processing) are becoming popular, with both Intel and AMD recently releasing CMP processors. Both of these solutions add processor throughput, but all virtual processors share the same cache, increasing the performance degradation of cache contention. In general, computer architecture is looking in the direction of increasing multi-tasking for higher throughput.

Computer security has also been an increasing concern. As our dependency on computing infrastructure continues to grow, security exploits will have greater cost. Buffer overflows, one of the more common exploits, occur when erroneous code allows external data to be written out of its bounds to overwrite, damage, and possibly control the program. Many attacks against the program heap could be prevented if unused pages were write-protected.

The project we are working on addresses aspects in both of these issues. The primary goal of this project is to provide increased security without negatively impacting performance. This goal is achieved by creating a memory heap server that executes in parallel on the processor with kernel memory privileges. By allowing the memory allocations and frees to occur asynchronously, the processor's capacity is utilized without creating much cache contention.

Because the memory allocation and freeing is asynchronous, a communication delay is added in the overhead of each memory allocation. By predicting the size and times of memory block requests, the memory management system can hide the latency of memory allocation. The more accurately this thread can predict requests, the faster the program will be executed.

For this project, I am aiding Mazen Kharbutli, who is working under Professor Solihin in the ECE department. My role is to determine ways that memory allocation management actions can be predicted without prior knowledge about the programs and their execution. The work has not yet been published, so further implementation and implication details are preferred to not be disseminated. With regard to data mining for memory allocation prediction, such details are nevertheless irrelevant.

2. Background

Because of the increased latency for memory allocations, the memory manager needs to successfully predict memory allocations occurring in the near future. Some programs have been found to have high sustained memory allocation request rates which will be difficult for the memory manager to keep up with. Such cases will require the memory allocation prediction to be accurate in batch allocations to prevent major slowdown.

It should also be noted that the application side maintains a pool of 8 to 16 pre-allocated memory sizes in reserve. The number of pre-allocated chunks may be large;

however two problems arise if the number is too large. The primary problem is that if the application requests a new memory size that is not in the memory pool, one of the size pools must be evicted. The overhead of freeing these unused memory blocks consumes time and communication bandwidth from the memory manager. The second problem is the extra memory required. It is a waste of system resources for a small program to allocate significant amounts of memory that it will not use.

Freed memory blocks may be added back to the pre-allocated pools instead of being sent back to the memory manager to be asynchronously freed. It is the job of the predictor to make all of these choices. In this paper, I will report on my findings in using data mining techniques to find insights that will help us in the construction of this predictor.

3. Related Work

Surprisingly little work has been done on predicting memory allocation requests. Much work has been done in garbage collection, but very little of it is applicable to standard low-level memory practices. In practice, performance bound programs that have specific memory allocation patterns have created custom memory allocators. However, they often do not outperform the standard memory allocation algorithms [1].

Zorn et. al did large amounts of work investigating using and predicting different memory allocation usages in terms of the memory block's lifetime. By using the stack information as input to an analytic predictor for segregating heap objects by usage patterns, Seidel and Zorn were able to decrease page faults and slightly improve cache hits [2,3]. In [4], Barrett and Zorn investigate using the predicting lifespan of an allocated block to designate the memory allocation technique and location used to allocate the memory block. Using this technique, they improve memory management performance by predicting lifespan based on the function call chain. Grunwald's work with Zorn [5] used different memory allocation strategies during run-time to optimize execution speed for a particular application while keeping unused memory low, based off application profiles.

Caching initialized objects to reduce initialization overhead during memory allocations was investigated by Bonwick [6], but does not use any predictive modeling. Chang, et. al investigated the use of a hardware memory management system using a bitmapped memory allocator, which requires linear storage overhead for the bitmap with regard to the size of the memory [7,8]. Wuytack et. al used decision trees to improve the throughput and behavior of memory management on high-throughput embedded network devices, however, the decision trees were derived analytically from knowledge about the specific data structures used [9].

3.1 Selected Paper Review

In their paper entitled, "Predicting lifetimes in dynamically allocated memory", Cohn and Singh describe their approach of using decision trees to tune memory management systems [10]. By better predicting memory block lifetimes, a memory manager can decrease CPU usage required for memory allocations, decrease memory fragmentation, and improve program cache locality. Their related work section is a bit short, but to give the authors some credit, little work was being done on memory

management at the time and this paper was cited and extended in following years in other important works in memory management.

Following the introduction, the authors describe the features selected for use in their decision tree. Previous work has investigated using program flow analysis to predict memory lifetimes, but the authors argue that it is not just the program flow and functions, but rather the parameters; the same function called with different parameters may behave differently. To take this into account, their decision tree's classification is based on the top 20 machine words in the stack. The authors find that use of the registers in the decision tree is insignificant.

In distinguishing short-lived memory blocks from long term blocks from "permanent" blocks (lasting the duration of program execution), the memory manager can use different strategies optimized for the memory blocks' lifetime. However, false positives may be costly, as memory blocks allocated with a mispredicted lifetime may adversely affect many other allocations. The authors assumed false positives and false negatives to have the same weight for ease of evaluation.

The authors used the OC1 decision tree software. The tree evaluation was based on a cost complexity heuristic, using 90% of the samples to build the tree and 10% to prune it. Because the software did not provide the capability of using a custom heuristic, the authors used a boosting technique to include equal numbers of positive and negative examples by duplicating training examples. They did not use the word boosting, perhaps because the term was not widely used at that point.

The authors profiled a common set of benchmarks: Ghostscript, Espresso, Cfrac, Gawk, Perl, and GCC. These programs were recompiled and linked against a slightly modified version of the standard C library that recorded when specific segments of memory were allocated and freed, along with the 20 words of stack trace.

The authors then compare the results of OC1 false positive and false negative rates to that of the only other work done prior to their paper (by Barrett and Zorn). Cohn and Singh's worst results are comparable, but the best results are far better predictors (e.g. 25.2% false negatives compared with 1.7% false negatives). For predicting permanent memory allocations, their system had very good performance for 3 of the 4 metrics.

To test the actual performance gained by using the decision tree, the authors implemented a slightly modified version. If a specific memory allocation site only reserved memory for one type of use (short-term, long-term, permanent), the authors used the specific memory allocation algorithm. If the memory allocation site's uses were mixed, the authors used the decision tree to dynamically choose which algorithm it would use. For most of the results, efficiency was increased by a small amount. The Cfrac program, however, performed very poorly during long runs due to mispredictions.

In this paper, the authors were the first to apply machine learning or data mining to memory management. In the past 9 years, virtually no other groups have applied such methods to lower level memory management (several have extended the methods, but I have been unable to find any work on this problem outside of higher level garbage collection techniques). I speculate that it may be due to the fact that most programs spend only a small fraction of their execution time performing memory management. Though exceptions do exist, it makes a tough case to sell when many programs would

receive only a small percentage increase in execution speed from the added complexity of a decision tree-based memory manager.

On the other hand, this work does seem somewhat incomplete. The authors used only 6 programs for the initial benchmarking, and only used 3 of them for the performance analysis. It would have also been interesting to isolate the time spent in the malloc and free functions before and after using the decision tree, as that would be a good indicator as to how their method would work for programs that perform large amounts of memory allocation.

The authors were also severely limited by the available data mining algorithms and software. They mentioned that they were unable to use their own evaluation mechanisms in the creation of the decision tree that would take into account the weights of false positives and false negatives. Given the way many new programming languages and methods instantiate many objects and destroy them frequently, I think that such a memory allocation predictor may have greater impact. I would be interested in seeing this work being redone with today's data mining techniques in today's environments, to see if it would yield better results.

3.2 Data Mining Algorithms

The primary data mining algorithm used in this work is J48, a C4.5 algorithm based decision tree classifier implemented in Weka. The algorithm attempts to maximize entropy gain when making each split in the decision tree. C4.5 has been widely used, and due to its maturity, searching for papers on C4.5 reveals that most of the studies involving it are applying it to other domains. Though entropy is used in the C4.5 algorithm, there are a wide variety of successfully used interestingness measures to choose from [11]. Bagging, boosting, and randomization has been studied with C4.5, showing that boosting gives the best results, with randomization and bagging giving similar quality decision trees [12], but bagging is the best method for practical use due to boosting's sensitivity to noise. Though I had a large volume of data in this project, I used bagging by sampling the large data set with replacement. In terms of decision tree theory, Fiat and Pechyony have recently explored some theoretical underpinnings for optimality [13].

4. Obtaining the Data

4.1 Program Selection

To obtain useful memory allocation patterns, careful consideration went into deciding which programs to profile. The programs must be diverse, because profiling ten programs that fulfill the same role would likely only show us a small subset of allocation patterns. However, the scope of available programs is greatly limited by the processor simulator on which they will be required to run for certain performance evaluations. The simulator has a very limited implementation of POSIX, without support for networking, multithreading, or X-Windows. To be useful, programs must also have non-trivial memory allocation patterns. For example, a program which allocates a small number of blocks upon initialization with no further memory allocation would not be affected by memory allocation latency. The programs need to be open source as well, so that they can be later recompiled to run on the processor simulator.

The SPEC CPU 2000 benchmarks are a common standard for processor profiling. Despite the commonality, we only chose a couple of those measures, since many of them do not exhibit the aforementioned characteristics. The final list of programs chosen is as follows:

Name	Description
7zip	data compression utility that uses a variety of compression algorithms
ammp	SPEC CPU 2000 benchmark
bison	CFG parser generator, run with the specifications of a Java language CFG parser source
diffutils	finds differences between text files
espresso	logic minimization solver
gawk	GNU Awk interpreter
gcc	optimizing C/C++ compiler (SPEC CPU 2000 benchmark)
gzip	data compression utility (SPEC CPU 2000 benchmark)
netpbm	image format translation library, run to convert still images to a compressed movie file
perl	interpreter for the Perl programming language, run with a script to do various text parsing
splint	C/C++ static security source analysis tool

4.2 Data Collection

The data were collected by augmenting the standard C library malloc functions to log every call. The obvious data to log is whether the memory is being allocated or freed, as well as the size requested for an allocation. In addition, we logged the current time as measured in CPU clock ticks (zero representing the first memory allocation), allowing us to see memory allocation timing patterns. We recorded the amount of time taken to perform the memory management task for future performance comparisons. The address of each memory allocated or freed was logged as well. This allows us to find when a specific block of memory was both allocated and freed, which would be required for memory lifetime analysis.

We also recorded the calling address of each malloc and free, hoping to be able to associate particular allocation sizes with certain parts of the code. After inspecting the data for many of the programs, I found the calling address attribute to be an unreliable indicator, which became more obvious after looking through the high level source code. Many programs use their own small allocation functions that call malloc and perform safety checks, but many also do not. As common programming standards and languages continuously change and call stack analysis can become quite complex for our application, we decided that utilizing this measure would not be in our best interests for universal prediction.

Initially, we intended to use the simulator to gather cache activity to find access patterns. Our hope was that the allocator could use cache access patterns to predict future access patterns and allocate memory to increase cache hits. Other commitments on the part of the project leader forced this data collection out of the schedule (they also delayed

my data from being collected for several weeks). In hindsight, the cache access patterns would have added a great deal of complexity to the system and the analysis, and seems best suited for future work after a functional system is in order.

5. Analysis

5.1 Constraints on Memory Predictor

The memory system imposes several constraints that each affects prediction in a different manner. One constraint is that all memory requests less than 16 bytes will be rounded up to 16 bytes, and all larger requests are rounded up to the next multiple of 8 bytes. This limitation is required for memory allocation efficiency, but also aids data mining slightly by somewhat discretizing the allowed sizes. The system easily allows for the possibility of using larger memory blocks for smaller requests, so if the system predicts a larger size, they can be used even if wrong. The predictor must be careful not to do this too often, or with sizes too large, otherwise the program will consume more memory than is required, suffer more page misses, and potentially suffer more cache misses.

A second constraint is that the memory predictor must be careful to not converge; it must be ergodic. If the memory predictor applies any real-time machine learning techniques, it must be designed to disregard or unlearn old memory allocation patterns. Programs often have distinct phases. If the memory allocation predictor's training kept all of the results since the beginning of the program, it may continue to use patterns from the earlier phase in the later phases. Occasionally this may be beneficial, but other times it may cause unreasonable predictions, severely hindering the performance of the memory manager.

The last major constraint is timing. To be effective, the memory manager must not use large amounts of processing power and memory. Any prediction algorithms must be relatively simple. The memory manager can afford more complex calculations if it predicts with reasonable accuracy that it will not be receiving memory allocation requests for a longer duration.

5.2 Sequential Analysis Attempt

Initially, I sought to break the stream of memory operations into phases, cluster the phases, and then find patterns that would be best predictors for each given phase type. To do this, I wrote a Perl script to combine the malloc's and free's that operated on the same address into the same memory allocation event. Instead of having a separate malloc and free line, the allocation lines had the memory blocks' lifetimes. This was particularly helpful because the malloc and free contained very different information, and the free events did not contain the blocks' sizes.

I investigated phase analysis as applied to intrusion detection systems, child growth, and low level instruction phases for processors. Unfortunately, all of the phase recognition systems I investigated were very large and complex. The complexity worried me on two fronts: the amount of effort it would take to implement the algorithms and also the amount of processing the memory manager can reasonably in predicting memory allocations. Additionally, all the phase recognizers I encountered were able to utilize higher dimensional data than I had, so I wondered about the capability to recognize phases based only on memory allocation times and sizes. I spent a bit of time

qualitatively analyzing the data (staring at every graph I could come up with), and eventually decided that I should try another more promising approach.

5.3 Collapsing Sequential Data

Given that the memory management predictor must make decisions based on the current situation, I decided to investigate collapsing each of the points in the time series into an element that represents everything that the predictor is likely to know about its situation. By evaluating heuristics from the perspective of every point in the data, sequential dependencies can be removed, and each data point can be analyzed independent of the others.

The evaluation heuristics are best guesses at what types of aggregated measures that could be useful. All of the aggregated measures were applied to the memory sizes after rounding according to the rounding requirements specified in section 5.1. The complete list of metrics I created and evaluated is as follows:

Name	**Description (all times are in processor cycles)**
size	Size of the current memory allocation request
previous_size	Size of the previous memory allocation request
time_since_last_alloc	Time since last allocation of any size
time_since_last_eviction	In maintaining a least-recently-used eviction policy in a cache of the most recently used 8 sizes, this is the time since the last eviction was required.
time_size_last_seen	Time since a memory allocation request of this size was received
last_free_this_size	Time since a memory block of the given size was last freed (0 if has never been freed)
entropy_of_last128	The entropy of last 128 memory allocation requests, computed with respect to allocation size probabilities. $H(X) = -\sum p(x) \log_2 p(x)$, where p(x) is the probability of the given size out of the last 128 sizes.
entropy_of_last32	Same as entropy_of_last128, but with the previous 32 memory allocation requests
entropy_of_last8	Same as entropy_of_last128, but with the previous 8 memory allocation requests
count_of_last128	Number of memory allocation requests of the current size out of the last 128 requests
count_of_last32	Same as count_of_last128, but with the previous 32 memory allocation requests
count_of_last8	Same as count_of_last128, but with the previous 8 memory allocation requests
time_til_next	Time until the next memory allocation request will occur - Used for training and evaluating.
count_of_next128	If the frequency of the current size is 0 for the next 128 memory allocation requests, then ‘none’. If the frequency is <= 1/3 of the 128, then ‘low’.

	If the frequency is <= 2/3 of the 128, then 'medium'. If the frequency is > 2/3 of the 128, then 'high'. Used for training and evaluating.

These metrics were coded as part of a batch preprocessor that translated the raw data into both ARFF and CSV files for use in Weka and Excel. The batch processor processed each of the data sets in its entirety (one per profiled application), and wrote out files with data elements that were sequentially independent, containing the metrics mentioned above. Because the count_of_next128 and count_of_last128 metrics needed a span of 256 entries, several data sets were thrown out at this point because they contained too few elements (the data sets thrown out are not listed in section 4.1). Ignoring this data is not problematic for a universal predictor because applications with few memory allocation requests will not be noticeably impacted by the increased overhead.

The batch preprocessor also sampled the applications by randomly choosing 120 of the data points from each of them, totaling 1320 data points from the 11 profiled applications in a single file for experimentation. I wrote the batch processor in Perl for several reasons: Perl is excellent at text processing; Perl is good for rapid prototyping; and I am very experienced with the Perl language.

5.4 Initial Analysis Using Weka

Earlier in the semester, before I had obtained my data, I experimented with both SAS and Weka. I chose to use Weka for two primary reasons. First, SAS does not have recent versions of its software available for Apple's OS X. And second, Weka was very easy to learn and very intuitive. The only portion I for which I needed to read training materials was the KnowledgeFlow package. I worked through several training tutorials with SAS. Though SAS had more features than Weka, I found it comparatively clunky, difficult to use, and non-intuitive.

When first exploring the available algorithms, I wanted to choose algorithms that would reveal information to aid in the developing of a predictor. Neural networks and other functional learning methods can predict numeric data. However, many functional data mining methods do not easily illuminate relations within the data; they work for producing a result.

As mentioned earlier, it is possible for the predictor to utilize slightly larger memory blocks for smaller requests. Coupled with the benefits of readability of classification algorithms compared to clustering algorithms, most notably decision trees, it seemed attractive to bundle the size requests into a smaller nominal set of data. To discretize the sizes, I applied Weka's supervised discretize method on the sampled data (using both count_of_next128 as the nominal type). This yielded the results in the following table:

Size Range	Number of Samples
(-inf-26]	655
(26-308]	394
(308-404]	79
(404-1148]	21
(1148-4088]	139
(4088-inf)	32

Although this range is useful, it is not as practical for combining memory sizes, as they do not maximize efficiency as well as sums of larger powers of 2. With this list in mind, along with the potential of reasonably oversized memory blocks and class versus array data sizes, I constructed the following categories:

(0-24]	very-small
(24-64]	small
(64-192]	medium-small
(192-512]	medium-large
(512-1024]	large
(1024-4096]	extra-large
(4096-inf]	huge

When I began experimenting with various algorithms, I tried using the size discretized and numeric. To a small surprise, I found that discretized size to predict the future frequency of the size in allocation requests had no significant impact on the results. Nearly all the algorithms I tried (using parameters reasonable for our data set) had classification accuracy within a couple percentage points (most were in the neighborhood of 80-85% accuracy) when comparing discretized to numeric size values in predicting count_of_next128. Though this indicates that discretizing of sizes may not be useful, it is required to use most non-function-based classifiers to predict the size of the next memory allocation requests.

5.5 Evaluation of Algorithms

To choose which algorithm to base most of the decisions off, I wanted to try a variety. I compared the following 9 algorithms for accuracy based on predicting count_of_next128 (with time_til_next removed from the input) using 10-fold cross-validation:

Algorithm Evaluated	**Accuracy**	**Notable Parameters**
Rule Based: JRip	86.2879	Use 10 rules
Rule Based: Conjunctive Rule (single rule learner)	77.1212	
Tree Based: J48	86.5152	0.000001 confidence factor (high threshold, producing a small tree)
Tree Based: Random Forrest	88.3333	10 trees
Lazy: IB1	85.4545	
Lazy: LWL	79.1667	
Boosting: AdaBoost w/ J48 tree	87.5758	
Network Based: Multilayer Perceptron	86.0606	
Network Based: Bayes Network	83.9394	

All of the classification algorithms tested yielded similar results. The network-based algorithms, especially the multilayer perceptron network took the longest to compute. Because this prediction model must run in a limited computing environment, so simple-to-compute models are preferable. Human readability of the results is also very preferable, because it allows for a straight-forward implementation of the predictor, as well as observable justifications for the predictor's behavior.

Because the primary goal of this analysis is to illuminate memory allocation prediction strategies, it is obviously very useful to know how well the heuristics compare for initial classification. By using the measure of information gain, the heuristics are ranked as follows:

Prediction Target	**Information Gain**	**Heuristic**
count_of_next128, discretized size	0.976	count-of-last-128
	0.976	count-of-last-32
	0.976	count-of-last-8
	0.531	entropy-of-last-128
	0.49	entropy-of-last-32
	0.45	entropy-of-last-8
	0.337	time-since-eviction
	0.276	size
	0.249	time-last-seen
	0.229	time-since-free
	0.217	previous-size
	0.141	time-since-alloc
count_of_next128, numeric size	0.3418	count-of-last-128
	0.3418	count-of-last-32
	0.3418	count-of-last-8
	0.2529	entropy-of-last-32
	0.2223	entropy-of-last-8
	0.22	entropy-of-last-128
	0.173	size
	0.1419	time-since-eviction
	0.1248	time-since-free
	0.1134	time-last-seen
	0.0897	time-since-alloc
	0.0684	previous-size
time_til_next	0.526	time-since-alloc
	0.431	time-last-seen
	0.429	size
	0.375	previous-size
	0.222	time-since-eviction
	0.185	entropy-of-last-128
	0.184	time-since-free
	0.18	count-of-last-32
	0.18	count-of-last-8
	0.18	count-of-last-128
	0.165	entropy-of-last-32
	0.134	entropy-of-last-8

size	1.1674	previous-size
	0.6217	time-last-seen
	0.5634	time-since-alloc
	0.5306	count-of-last-32
	0.5306	count-of-last-128
	0.5306	count-of-last-8
	0.5204	time-since-eviction
	0.2932	entropy-of-last-128
	0.2654	time-since-free
	0.1383	entropy-of-last-32
	0.0992	entropy-of-last-8

From the initial analysis, it shows that the count_of_last128 is the best predictor of the frequency of the current size occurring in the next 128 allocations, as are counts of the previous 32 and 8. Discretization of size had very small affects on the ordering. It was also interesting that the best predictor of time_til_next was time_since_alloc, and size was previous_size. These results indicate the tendency for memory allocation patterns to follow the same patterns on the small scale, but provide interesting information as to what parameters are useful when the patterns are not as regular.

5.6 Results

When creating models for optimizing program execution, care must be taken not to over-fit the profiles. Optimizing for one or two particular programs is usually easy, but optimizing such that most programs will run faster is much more difficult. Coupling this with Occam's razor, I sought the simplest decision trees that would give me good results. I chose to use Weka's J48 algorithm decision tree classifier, which is based off the C4.5 decision tree algorithm because it is widely used and seemed to be very effective.

By trying different confidence factor thresholds in decision tree pruning, I found that decreasing the confidence threshold to .000001, the decision tree was very small, but still provided results within a few percentage points of large confidence thresholds.

For the count_of_next128 prediction using discretized sizes, Weka produced:

Decision Tree	count-of-last-128 <= 86 | count-of-last-128 <= 35: low (387.0/63.0) | count-of-last-128 > 35 | | entropy-of-last-32 <= 0.195909: high (17.0/1.0) | | entropy-of-last-32 > 0.195909 | | | time-since-free <= 240336208: medium (218.0/50.0) | | | time-since-free > 240336208: low (10.0/1.0) count-of-last-128 > 86: high (688.0/64.0)				
Accuracy	Correctly Classified Instances 1115 84.4697 % Incorrectly Classified Instances 205 15.5303 %				
Confusion Matrix	A	B	C	D	← Classified as
	2	35	0	0	A=none
	0	332	32	15	B=low
	0	25	147	60	C=medium
	1	12	25	634	D=high

This result shows that the best predictor is how frequently the size has occurred before, but that if the count is mid-range, then it depends more on how many different sizes are being allocated in the last 32 allocations.

For the count_of_next128 prediction using numeric sizes, Weka produced the following decision tree:

Decision Tree	count-of-last-32 <= 13 | count-of-last-128 <= 34: low (349.0/45.0) | count-of-last-128 > 34 | | count-of-last-8 <= 5 | | | size <= 36: medium (84.0/30.0) | | | size > 36: low (11.0) | | count-of-last-8 > 5: high (7.0/2.0) count-of-last-32 > 13 | count-of-last-128 <= 87 | | count-of-last-32 <= 25: medium (154.0/30.0) | | count-of-last-32 > 25: high (33.0/11.0) | count-of-last-128 > 87: high (682.0/73.0)				
Accuracy	Correctly Classified Instances 1096 83.0303 % Incorrectly Classified Instances 224 16.9697 %				
Confusion Matrix	A	B	C	D	← Classified as
	0	33	0	1	A=none
	3	317	37	22	B=low
	0	30	157	66	C=medium
	0	6	26	622	D=high

These results are interesting in how they differ from the results produced with discretized sizes. This tree uses all the ranges of counts, and uses sizes for some fine tuning.

For time_til_next prediction using numeric sizes (but descritizing the data set in order to classify time_til_next), Weka produced:

<table>
<tr><td>Decision Tree</td><td colspan="5">time-since-alloc = (-inf-4622]: (-inf-4598] (527.0/145.0)
time-since-alloc = (4622-27924]: (4598-28286] (510.0/183.0)
time-since-alloc = (27924-123326]
| entropy-of-last-32 = (-inf-0.481238]: (28286-134414] (78.0/29.0)
| entropy-of-last-32 = (0.481238-1.221548]: (4598-28286] (9.0/5.0)
| entropy-of-last-32 = (1.221548-1.570624]: (4598-28286] (6.0/3.0)
| entropy-of-last-32 = (1.570624-inf): (4598-28286] (36.0/10.0)
time-since-alloc = (123326-13316372]: (134414-inf) (146.0/30.0)
time-since-alloc = (13316372-inf): (28286-134414] (8.0/1.0)</td></tr>
<tr><td>Accuracy</td><td colspan="5">Correctly Classified Instances 910 68.9394 %
Incorrectly Classified Instances 410 31.0606 %</td></tr>
<tr><td rowspan="5">Confusion Matrix</td><td>A</td><td>B</td><td>C</td><td>D</td><td>← Classified as</td></tr>
<tr><td>382</td><td>152</td><td>5</td><td>1</td><td>A= (-inf-4598]</td></tr>
<tr><td>129</td><td>358</td><td>4</td><td>13</td><td>B= (4598-28286]</td></tr>
<tr><td>16</td><td>39</td><td>52</td><td>22</td><td>C= (28286-134414]</td></tr>
<tr><td>0</td><td>10</td><td>19</td><td>118</td><td>D= (134414-inf)</td></tr>
</table>

The accuracy is not as high for predicting time_til_next, but predicting the attribute is not as important as the others. The decision tree shows that programs tend to allocate memory at fairly constant rates. The only exception is if a program is having mid-range times between allocations, it depends more on the number of different sizes being allocated. This makes intuitive sense in that if a program is having some time between allocations, it may be doing different tasks, and the variety of those tasks would predict how long until the next allocation.

For size prediction (using a slightly smaller confidence factor of .0000001), Weka produced:

Decision Tree

```
previous-size <= 304
|  time-last-seen <= 46648
|  |  previous-size <= 24: very-small (523.0/31.0)
|  |  previous-size > 24
|  |  |  previous-size <= 64
|  |  |  |  time-since-free <= 6096
|  |  |  |  |  count-of-last-8 <= 0: medium-small (2.0)
|  |  |  |  |  count-of-last-8 > 0: small (31.0)
|  |  |  |  time-since-free > 6096
|  |  |  |  |  count-of-last-128 <= 39
|  |  |  |  |  |  time-since-free <= 65129880: small (59.0/11.0)
|  |  |  |  |  |  time-since-free > 65129880: very-small (4.0)
|  |  |  |  |  count-of-last-128 > 39: very-small (52.0/1.0)
|  |  |  previous-size > 64
|  |  |  |  time-last-seen <= 7664: medium-small (42.0/1.0)
|  |  |  |  time-last-seen > 7664
|  |  |  |  |  count-of-last-8 <= 7: very-small (36.0/8.0)
|  |  |  |  |  count-of-last-8 > 7: medium-small (10.0)
|  time-last-seen > 46648: small (281.0/115.0)
previous-size > 304
|  previous-size <= 448: medium-large (83.0)
|  previous-size > 448
|  |  count-of-last-128 <= 95
|  |  |  previous-size <= 1024
|  |  |  |  count-of-last-128 <= 31: large (19.0/8.0)
|  |  |  |  count-of-last-128 > 31: very-small (6.0)
|  |  |  previous-size > 1024: huge (36.0/10.0)
|  |  count-of-last-128 > 95
|  |  |  time-since-alloc <= 30204: extra-large (5.0/2.0)
|  |  |  time-since-alloc > 30204: extra-large (131.0)
```

Accuracy

Correctly Classified Instances	1098	83.1818 %
Incorrectly Classified Instances	222	16.8182 %

Confusion Matrix

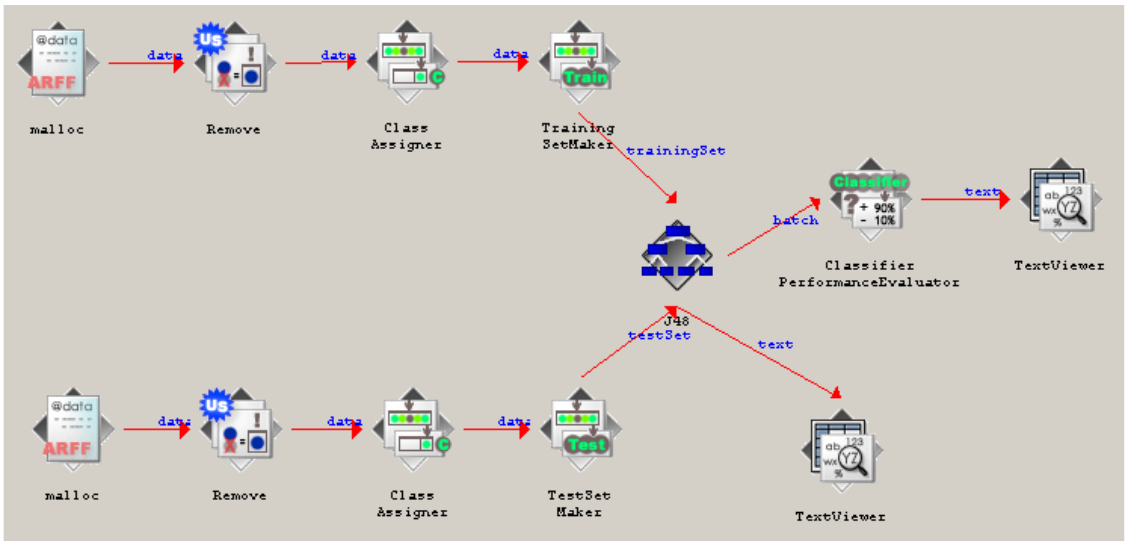
A	B	C	D	E	F	G	←Classified as
565	75	6	2	1	2	3	A=very small
38	233	0	1	1	1	1	B=small
8	38	52	2	2	0	1	C=medium-small
1	8	0	83	2	0	0	D=medium-large
2	6	0	0	9	2	0	E=large
2	5	1	0	3	131	6	F=extra-large
0	1	0	0	0	1	25	G=huge

Attempting to predict the size is a little more difficult than the other attributes, and thus produces a bigger decision tree. The dominant factor is the previous size, and then is decided by how many times it the particular size was seen (which would need to

be evaluated for each of the different sizes and compared for accuracy). Some of the lower nodes in the decision tree decide on timing, but those seem to be less significant.

5.7 Evaluating the Model

In order to automate the evaluation of the model against all of the sources independently, I needed to use Weka's KnowledgeFlow interface. This interface allows the user to develop a pipeline of data processing and manipulation. The following figure shows the model I developed for evaluating the accuracy of count_of_next128 with discretized size:



In this figure, the data flows from left-to-right. The top left ARFF loader loads the file containing the sampled memory allocations from all of the programs. The unnecessary fields are removed, source file and time_til_next, to make sure they are not used by the classifier. The Class Assigner module then picks count_of_next128 as the class differentiator. The data's type is converted to that of a training set by the TrainingSetMaker, and then sent to the J48 classification module. The bottom left path is almost identical to the top left path, except it loads the full test data for an individual program, and changes it to a test set. The immediate TextViewer coming out of the J48 classifier produces the decision tree. The results of the classification are sent to the ClassifierPerformanceEvaluator to determine accuracy and error measures, and the results sent to a TextViewer. The KnowledgeFlow models for evaluating the prediction of size and count_of_next128 with numeric data are almost identical. A testing system was not built for time_til_next. The main reason was that we deemed it as less important, since it is not as critical for performance, and it probably would not have been worth the effort (a meaningful discretization of time_til_next would have been required to use the J48 classifier). The results of the measures are in the following table:

Program Name	count_of_next128 prediction accuracy (discretized size)	size prediction accuracy	count_of_next128 prediction accuracy (numeric size)
7zip	76.4706	66.5991	76.9439
gawk	81.2883	80.184	80.9202
netpbm	69.7867	90.7673	68.9908
ammp	97.9884	99.9683	98.1943
gcc	88.6121	88.3514	88.8402
perl	89.5768	57.3246	89.4866
bison	86.5459	80.4585	85.8227
gzip	85.6368	80.3333	85.7282
splint	87.9127	87.123	87.3335
diffutils	100	100	100
espresso	85.4824	76.1948	84.3102

The count_of_next128 evaluations were fairly good across all the measures. The lowest accuracy was netpbm, which was still at 69.8%. Size prediction accuracy had slightly more disparity, the lowest being perl at 57% accuracy. To see if perl is easily predictable, even though it performs only moderately well on this model, I tried classifying it on its own. With the same low confidence factor of .0000005, Weka produced a large decision tree of 333 nodes, with accuracy of 87%. I could not decrease the threshold any lower without Weka failing to produce any decision tree at all. Perl may be predictable, but it is too complex and too niche (it is only one program) for us to consider such models.

5.8 Discussion

Before doing this investigation, we had several hypotheses. We thought that determining the common frequently allocated sizes would be a good predictor of future frequencies. This hypothesis was confirmed, most notably by the strength of the previous counts used in deciding the frequency of a particular size in the next 128 allocations. We also thought that by discretizing allocation size, we would be able to better our predicting. In this study, however, I found that this is only partly true. Discretization of sizes is required in order to use any easily understandable classifier model in order to predict future sizes, which conforms to our initial hypothesis. Comparing discretization to numeric treatment of sizes for predicting the future frequency of a given size does not seem to improve the results. Discretizing the size does change the structure of the decision tree and relative importance of different attributes, which is interesting. With numeric results, the decision tree took into account the variety of sizes being allocated (entropy), whereas the discretized decision tree mostly dealt with past occurrence frequencies. Perhaps it would be best to combine the two models. This will need to be studied further.

This study did confirm our belief that memory allocation sizes can be predicted with some accuracy. The fact that the prediction of future frequency is more accurate than the prediction of the next allocation size reinforces the initial design of having a pool of pre-allocated sizes waiting to be allocated.

Ultimately, the execution speed will be the final determining factor as to the confidence of these results. Even with our knowledge of importance of attributes in predicting memory allocation, the system will most likely require a good deal of fine parameter tuning, and evaluation on more different programs. We also need to watch for any programs that perform pathologically badly under our model, but obtaining any reasonable sample of such a large pool of used computer programs is virtually impossible.

These decision trees will be used in creating heuristics for our predictor.

6. Conclusion

This study of methods to predict memory allocation patterns had good results. It showed that memory allocation prediction is worthwhile and predictable, and also provided easily to understand heuristics from which to base our predictor. The decision tree classifications worked as well as any other type of predictor behavior. We will continue this work on memory allocation prediction with this information, and hopefully be able to hide the majority of the additional latency required by our memory allocation system.

References

[1] Berger, Emery D., Zorn, Benjamin G., McKinley, Kathryn S. Reconsidering Custom Memory Allocation. 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), pp 1-12, November 2002.

[2] Seidl, Matthew L., Zorn, Benjamin G. Segregating Heap Objects by Reference Behavior and Lifetime. Eight International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOSVIII), pp 12-23, San Jose, CA, October 1998.

[3] Seidl, Matthew L., Zorn, Benjamin G. Predicting References to Dynamically Allocated Objects. Department of Computer Science, University of Colorado at Boulder, Technical Report CU-CS-826-97, January 1997.

[4] Barrett, David A., Zorn, Benjamin G. Using Lifetime Predictors to Improve Memory Allocation Performance. ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, pp 177-186. Albuquerque, NM. June 1993.

[5] Grunwald, Dirk., Zorn, Benjamin G. CustoMalloc: Efficient Synthesized Memory Allocators. Technical Report CU-CS-602-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, July 1992.

[6] Bonwick, Jeff. The Slab Allocator: An Object-Caching Kernel Memory Allocator. Proceedings of the USENIX Summer Technical Conference, pp 87-98, June 1994.

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

[8] Daugherty, C. H., Chang, J. M. "Common List Method: A Simple, Efficient Allocator Implementation", Proceedings of Sixth Ann. High-Performance Computing Symposium, Boston, Massachusetts, Apr. 5-9, 1998. pp. 180-185.

[9] Wuytack, Sven., da Silva, Julio L. Jr., Catthoor, Francky. de Jong, Gjalt. Ykman-Couvreur, Chantal. Memory Management for Embedded Network Applications. Readings in Hardware/Software Co-design. Kluwer Academic Publishers. pp. 465-476. 2002.

[10] Cohn D, Singh S. Predicting lifetimes in dynamically allocated memory. Advances in Neural Information Processing Systems 9. 1996.

[11] Hilderman, Robert J., Hamilton, Howard, J. Knowledge Discovery and Interestingness Measures: A Survey. Technical Report CS 99-04, University of Regina, Regina, Saskatchewan, Canada, 1999.

[12] Dietterich, Thomas G. An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization. Machine Learning vol 40, pp 139-157. 2000.

[13] Fiat, Amos., Pechyony, Dmitry. Decision Trees: More Theoretical Justification for Practical Algorithms. Proceedings of the 15th International Conference Algorithmic Learning Theory. Pp 156-170. Padova, Italy. October 2004.