FRAMEWORK FOR EVALUATING DYNAMIC MEMORY ALLOCATORS
INCLUDING A NEW EQUIVALENCE CLASS BASED
CACHE-CONSCIOUS ALLOCATOR

Tomislav Janjusic

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

August 2013

APPROVED:

Krishna M. Kavi, Major Professor
Ron K. Cytron, Committee Member
Song Fu, Committee Member
Mahadevan Gomathisankaran, Committee Member
Paul Tarau, Committee Member
Barret R. Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering
Mark Wardell, Dean of the Toulouse Graduate School
Software applications’ performance is hindered by a variety of factors, but most notably by the well-known CPU-memory speed gap (often known as the memory wall). This results in the CPU sitting idle waiting for data to be brought from memory to processor caches. The addressing used by caches cause non-uniform accesses to various cache sets. The non-uniformity is due to several reasons, including how different objects are accessed by the code and how the data objects are located in memory. Memory allocators determine where dynamically created objects are placed, thus defining addresses and their mapping to cache locations. It is important to evaluate how different allocators behave with respect to the localities of the created objects. Most allocators use a single attribute, the size, of an object in making allocation decisions. Additional attributes such as the placement with respect to other objects, or specific cache area may lead to better use of cache memories.

In this dissertation, we proposed and implemented a framework that allows for the development and evaluation of new memory allocation techniques. At the root of the framework is a memory tracing tool called Gleipnir, which provides very detailed information about every memory access, and relates it back to source level objects. Using the traces from Gleipnir, we extended a commonly used cache simulator for generating detailed cache statistics: per function, per data object, per cache line, and identify specific data objects that are conflicting with each other. The utility of the
framework is demonstrated with a new memory allocator known as equivalence class allocator. The new allocator allows users to specify cache sets, in addition to object size, where the objects should be placed. We compare this new allocator with two well-known allocators, viz., Doug Lea and Pool allocators.
ACKNOWLEDGMENTS

I am very thankful and appreciative of numerous individuals for the support that I have received throughout my current academic career. First and foremost I would like to thank my mother, Anka Janjusic-Sratel and my father Petar Janjusic (may he rest in peace). Her dedication, support, and personal sacrifices have helped propel me in achieving the greatest accomplishment so far. Her guidance, wisdom, and love have helped me become a better, stronger, and more sophisticated individual. I can honestly say that few children are lucky to have a mother of her statue.

I would also like to thank my professor and friend Dr. Krishna M. Kavi. His utmost dedication to our work, resourcefulness, passion, and patience have made an everlasting impact and helped me become a better and more fulfilling academic. Our countless discussions and numerous meetings have sparked many intellectually rewarding debates. For his support as an academic and friend I am eternally grateful and I am most excited to future collaborations.

My thanks also go out to all my committee members, Dr. Mahadevan Gomathisankarn, Dr. Paul Tarau, Dr. Song Fu, and Dr. Ron Cytron; their advice and guidance were always very appreciated. I am also very grateful to all my friends and colleagues at the Computer Science and Engineering department who were a part of this exciting and rewarding journey. My thanks also go out to all the department administrators and to my former flatmates Tamara Schneider-Jimenez and Jose Jimenez.

I am also very appreciative and grateful to the Lantz family (Chris, Sheila, Ashley, Kyle, Kevin, and Brittney) for extending their kindness and love to me. Their support made an impact on me as an individual and their friend. Finally I would like to thank my girlfriend Ashley Lantz for standing by my side in moments of need. Her understanding, patience, and love have helped me accomplish a life-time achievement.
**CONTENTS**

ACKNOWLEDGMENTS iii

LIST OF TABLES vii

LIST OF FIGURES ix

CHAPTER 1. INTRODUCTION 1
  1.1. Motivation 1
  1.2. Processor Memory Hierarchy 2
  1.3. Software Memory Allocation 7
  1.4. Major Contributions 9

CHAPTER 2. SURVEY OF PROFILING TOOLS 12
  2.1. Introduction 12
  2.2. Instrumenting Tools 14
    2.2.1. Valgrind. 14
    2.2.2. DynamoRIO. 17
    2.2.3. Pin. 18
    2.2.4. DynInst. 19
  2.3. Event-driven and Sampling Tools 21
  2.4. Hardware Simulators 23
  2.5. Conclusions 24

CHAPTER 3. SURVEY OF MEMORY ALLOCATORS 25
  3.1. Introduction 25
    3.1.1. Allocator Design 26
    3.1.2. Fragmentation 28
    3.1.3. Locality 29
3.1.4. Analysis 30
3.1.5. Memory Usage Pattern 30
3.1.6. Allocator Mechanics 32
3.2. Classic Allocators 35
3.2.1. Sequential Fits 35
3.2.2. Segregated Lists 38
3.2.3. Buddy Systems 40
3.2.4. Indexed and Bitmapped Fits 42

CHAPTER 4. GLEIPNIR 44
4.1. Introduction 45
4.1.1. Fine-grained Memory Access Analysis 45
4.2. Implementation Overview 47
4.2.1. Valgrind’s Intermediate Representation 47
4.2.2. Tracing Instructions 48
4.2.3. Tracing Static, Global, and Dynamic Data 50
4.2.4. Multi-threading 53
4.2.5. Multi-process Capabilities 55
4.3. Analysis Environment 56
4.3.1. Analysis Cycle 56
4.3.2. Cache Behavior 57
4.3.3. Cache Simulation 58
4.3.4. Visualizing Data Layout 63
4.4. Multi-core and multi-process analysis 66
4.5. Future Work 68
4.5.1. Identifying Logical Structures 69
4.5.2. Trace-driven Data Structure Transformations 69
4.6. Conclusions 70
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Address cache index bits representation.</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Address (AD2000) bit representation (lower 32bits).</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Application profiling: instrumenting tools.</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Gleipnir’s basic trace line.</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Gleipnir’s advanced trace line for <em>stack</em> and <em>global</em> data.</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>Gleipnir’s advanced trace line for <em>dynamic</em> data.</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>Cache simulation results.</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>Cache simulation function results.</td>
<td>59</td>
</tr>
<tr>
<td>4.6</td>
<td>Simulation results for function <em>encode_mcu_AC_refine</em> and its accessed</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>variables and structures.</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Cache simulation’s cost-matrix for function <em>encode_mcu_AC_refine</em>.</td>
<td>67</td>
</tr>
<tr>
<td>4.8</td>
<td>Gleipnir trace with physical address tracing enabled.</td>
<td>68</td>
</tr>
<tr>
<td>5.1</td>
<td>Binary search stress-test benchmark’s simulation results for <em>dl_malloc</em>.</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>Binary search stress-test benchmark’s simulation results for <em>pool_malloc</em>.</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>Binary search stress-test benchmark’s simulation results for <em>eqmalloc</em>.</td>
<td>89</td>
</tr>
<tr>
<td>5.4</td>
<td>Binary search: stack, global, and heap segment overall L1 cache misses.</td>
<td>89</td>
</tr>
<tr>
<td>5.5</td>
<td>Linked-list search stress-test benchmark’s simulation results for <em>dl_malloc</em>.</td>
<td>93</td>
</tr>
<tr>
<td>5.6</td>
<td>Linked-list search stress-test benchmark’s simulation results for <em>poolmalloc</em>.</td>
<td>94</td>
</tr>
<tr>
<td>5.7</td>
<td>Linked-list search stress-test benchmark’s simulation results for <em>eqmalloc</em>.</td>
<td>94</td>
</tr>
<tr>
<td>5.8</td>
<td>Linked-list search: stack, global, and heap segment overall L1 cache misses.</td>
<td>94</td>
</tr>
<tr>
<td>5.9</td>
<td>Multi-list search stress-test benchmark’s simulation results for <em>dl_malloc</em>.</td>
<td>100</td>
</tr>
<tr>
<td>5.10</td>
<td>Multi-list search stress-test benchmark’s simulation results for <em>poolmalloc</em>.</td>
<td>100</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Memory hierarchy.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Physical and virtual memory concept view.</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Memory to cache concept view.</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Application’s data segments concept view.</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Application and hardware profiling categories.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Sampling tools and their dependencies and extensions.</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Allocator splitting and coalescing concept.</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>External memory fragmentation concept.</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Internal fragmentation concept.</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Dijkstra’s memory usage pattern (peak).</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Jpeg’s memory usage pattern (plateau).</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>Micro-benchmark’s memory usage pattern (ramp).</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>Fitting an 8byte block into a cache line.</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Buddy system memory splitting, concept.</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>SuperBlock flow chart.</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>Valgrind client state: without Valgrind (left) and with Valgrind (right).</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Forking a child process of a client.</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>Tracing and analysis cycle.</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Jpeg: load, store, or data modifies for every 100k references.</td>
<td>60</td>
</tr>
<tr>
<td>4.6</td>
<td>Jpeg: load, store, or data modify misses for every 100k references.</td>
<td>60</td>
</tr>
<tr>
<td>4.7</td>
<td>Jpeg: heap data references for every 100k references.</td>
<td>61</td>
</tr>
<tr>
<td>4.8</td>
<td>Jpeg: heap data misses for every 100k references.</td>
<td>61</td>
</tr>
</tbody>
</table>
4.9 CPU’s L1 cache state at 240,000 executed data references.

4.10 A cache-set layout visual representation of Jpeg function encode_mcu_AC_refine.

5.1 Virtual pages into a 32K cache concept.

5.2 Overview diagram of Eqalloc.

5.3 DLmalloc general block management.

5.4 Poolmalloc general block management.

5.5 A binary search stress-test benchmark.

5.6 Binary search: Stack, global, and heap references using dl_malloc.

5.7 Binary search: Stack, global, and heap misses using dl_malloc.

5.8 Binary search: heap node misses using dl_malloc.

5.9 Binary search: binary tree’s NODE_1k, NODE_2k, NODE_5k cache mapping using dl_malloc.

5.10 Binary search: equivalence allocator’s allocation strategy.

5.11 Binary search: binary tree’s NODE_1k, NODE_2k, and NODE_5k cache mapping using eq_malloc.

5.12 Binary search: binary tree’s NODE_1k, NODE_2k, NODE_5k cache mapping using pool_malloc.

5.13 Linked-list search stress-test benchmark

5.14 Linked-list: linked-list NODE_500, NODE_1k, NODE_1.5k, and NODE_2k misses using dl_malloc.

5.15 Linked-list: Stack, global, and heap cache mapping using dl_malloc.

5.16 Linked-list: Stack, global, and heap cache mapping using eq_malloc.

5.17 Linked-list search stress-test benchmark’s NODE_500, 1000, 1500, 2000 cache mapping using the equivalence allocator.

5.18 Linked-list: Stack, global, and heap cache mapping using pool_malloc.
5.19 Multi-list search stress-test benchmark.

5.20 Multi-list: *Stack, global, and heap* reference counts using *dl* _malloc_.

5.21 Multi-list: *Stack, global, and heap* miss counts using *dl* _malloc_.

5.22 Multi-list: Heap data references using *dl* _malloc_.

5.23 Multi-list: Heap data misses using *dl* _malloc_.

5.24 Multi-list: *Stack, global, and heap* cache mapping using *dl* _malloc_.

5.25 Multi-list: Heap object’s cache mapping using *dl* _malloc_.

5.26 Multi-list: *Stack, global, and heap* cache mapping using *eq* _malloc_.

5.27 Multi-list: *Stack, global, and heap* cache mapping using *pool* _malloc_.

5.28 Dijkstra’s node walk.

5.29 Dijkstra: *stack, global, and heap* references using *dl* _malloc_.

5.30 Dijkstra: *stack, global, and heap* misses using *dl* _malloc_.

5.31 Dijkstra: *stack, global, and heap* cache mapping using *dl* _malloc_.

5.32 Dijkstra: *stack, global, and heap* cache mapping using *pool* _malloc_.

5.33 Dijkstra’s *stack, global, and heap* cache mapping using *eq* _malloc_.

5.34 Jpeg: *Stack, global, and heap* cache mapping using *dl* _malloc_.

5.35 Jpeg: *very small* and *small block* cache mapping using *dl* _malloc_.

5.36 Jpeg: *medium block* cache mapping using *dl* _malloc_.

5.37 Jpeg: *large block* cache mapping using *dl* _malloc_.

5.38 Jpeg: *stack, global, and heap* cache mapping using *pool allocator*.

5.39 Jpeg: heap objects cache mapping using *pool allocator*.

5.40 Patricia’s node insertion.

5.41 Patricia: *stack, global, and heap* cache mapping using *dl* _malloc_.

5.42 Patricia: *stack, global, and heap* cache mapping using *pool* _malloc_.

5.43 Patricia: heap structures cache mapping using *pool* _malloc_.

xi
5.44 Patricia: *stack, global, and heap* cache mapping using *eq malloc.* 117
5.45 Patricia: structure cache mapping using *eq malloc.* 118
5.46 Virtual memory utilization when using *dl malloc.* 122
5.47 Multi-list: virtual memory utilization when using *dl malloc* (heap zoomed in). 122
5.48 Multi-list: virtual memory utilization (full-view) using pool allocator. 123
5.49 Multi-list: virtual memory utilization using pool allocator (heap zoomed in). 124
5.50 Multi-list: virtual memory utilization using the equivalence allocator. 125
5.51 Patricia: virtual memory utilization using the equivalence allocator. 125
5.52 Allocator misses cache summary. 127
CHAPTER 1

INTRODUCTION

In this chapter we provide a high level view of software and hardware interactions as they relate to an application's memory usage. We first describe the memory systems design from a hardware point of view, and then show how these designs impact the performance of software applications. Since most modern programming languages and software allocates memory for objects dynamically based on the need, the management of dynamic memory is critical to the performance of applications. The primary goal of our research is to improve various hardware and software techniques to optimize performance. We will elaborate on our motivation and achievements.

1.1. Motivation

Software application’s performance is hindered by a variety of factors, but most notably driven by the known CPU-Memory speed gap (also known as memory wall). The discrepancy lies in the CPU’s enduring speed increase at roughly 60% per year while memory’s speed increase is less than 10% per year. This results in the CPU sitting idle waiting for data to be brought from memory to processor caches. While caches are designed to alleviate the speed gap, the addressing used by caches causes non-uniform accesses to various cache sets: some sets are heavily accessed while other sets are rarely accessed; and the heavily accessed sets cause most conflict misses. The non-uniformity is due to several reasons, including how different objects are accessed by the code and how the data objects are located in memory. In principle one can explore changing the order of accesses (by code refactoring) or by changing where the data is located. In this thesis we will focus on the second solution - changing the data layout. In order to achieve this goal it is necessary to know how an application creates and accesses data during its execution. For this purpose we developed a tool called Gleipnir that can identify every memory location accessed by an application. Gleipnir is different from other instrumentation tools because it has the ability to relate the accesses back to program internal structures. This information can then be used to either
relocate the object or change how data objects are defined.

Memory allocators determine where dynamically created objects are placed, thus defining addresses and their mapping to cache locations. Thus it is important to evaluate how different allocators behave with respect the localities of the created objects. It may also be possible to develop new allocators for the sole purpose of improving localities. Most allocators use a single attribute, the size, of an object in making allocation decision. One of our goal is to provide the allocator with additional attributes so that the allocator may achieve better results in terms of localities. Our research developed one such allocator called *Equivalence Class based Cache Conscious Dynamic Memory Allocator*. This allocator uses the equivalence class specified with an allocation request to determine where to locate the object. The key idea is that objects in a given equivalence class fall to the same cache sets. Thus conflicts may be avoided by allocating objects to different equivalence classes.

1.2. Processor Memory Hierarchy

The study of how the memory layout and data placement affects the performance requires a detailed understanding of the underlying memory and processor architecture. Figure 1.1 shows a stylized view of memory hierarchy of modern processors - the sizes of the memories is reflected by the pyramid shape. The fastest storage units, which are CPU registers, are on top, and the slowest storage system, which is a magnetic disk drive, is at the bottom of the pyramid.

![Memory hierarchy](image)

**Figure 1.1.** Memory hierarchy.

In most load/store architectures, the CPU uses registers for its computations, since they are the fastest memory in the hierarchy. By keeping most of the data in registers during
a computation, the processor can achieve very high clock speeds. However since there are only a limited number of registers, the processor needs to bring data from the next level of memory, L-1 cache to registers (and move data from registers to cache). It should be noted that most processors rely on stored program paradigm, meaning that even the instructions that a processor executes must come from memory. In this work we will not consider the memory system as it pertains to instructions, but focus only data memory. L-1 cache (or lower level caches) is the next fastest memory in the system, but once again, to reduce the cost of the system, L-1 caches are also limited in size. We then introduce additional levels of caches (L-2, L-3) and keep more recently used data in lower levels of caches. If the data is not found in caches a cache miss results and the missing data is requested from the next higher level of memory; and if the data is not found in any of cache levels, we then request the data from main memory, which is typically built using DRAM technology. Finally if the data is not found there, the application is context switched out since it takes a very long time to move data from the disk to main memory.

The concept of memory hierarchy, whereby most commonly and frequently used data is kept in faster memory, is based on localities exhibited by applications. There are actually two types of localities, spatial and temporal localities. Temporal locality implies that, once a location is referenced, there is a high probability that it will be referenced again soon, and less likely to do so as time passes; spatial locality implies that when a datum is accessed it is very likely that nearby data will be accessed soon. An example of temporal locality results from accesses to loop indices, while access to elements of an array exhibit spatial localities. Since cache stores recently used segments of information, the property of locality implies that needed information is also likely to be found in the cache.

Since a cache miss causes processor delays, we should strive to minimize the number of cache misses encountered by the application. Before we discuss cache misses, it is necessary to understand how caches use data addresses, including concepts like virtual and physical addresses, address binding, and cache indexing.

There are two types of data addresses that a CPU’s recognizes: virtual and physical.
Virtual addresses are addresses that are generated by the application’s instructions, while physical addresses refer to the actual addresses in main memory where the application’s data resides. The main or physical memory of a system (sometimes referred to as RAM) is managed by the operating system of a computer. When an application is loaded into the RAM, a set of logical memory regions known as pages are given to the application. The application’s data resides in these pages, and the system provides mechanism for translating the application’s virtual address to these physical pages. Pages are fixed sized entities, and in most current systems, they contain 4096 (or 4KB) bytes.

Figure 1.2 shows a conceptual view of an O.S. and its virtual and physical page mapping. The location of the physical pages allocated to an application will impact where the data is mapped in a cache, if the cache is physically indexed. Most current processors utilize physical indexes, although some L-1 caches are designed to use virtual indexing which are based on virtual addresses instead of physical addresses. For the purpose of understanding how caches work, it is not necessary to distinguish between physically addressed and virtually addressed caches.

When a CPU requests data from memory it uses cache indexing technique to place the
data in an appropriate block in the cache. A CPU’s cache is indexed using a data’s memory address. Assume we have a 32K byte sized cache with 8 ways and 64 bytes per cache line for a total of 64 sets. Also, suppose we have a 4096 byte sized page mapped to some address \(X\); for example \(X = 0xAD2000\). Address bits are divided into \(TAG, INDEX, \) and \(OFFSET\) bits. For a cache of 64 unique sets and 64 bytes per line we require 6 bits to locate (or index) the set and 6 bits to determine the offset of the specific byte requested; the remaining bits are used as \(TAG\) to match the currently residing data with correct address. The index bits required are given by \(B = log_2(sets)\). The rest of the address is used to tag the data therefore for a 64 bit address we need \(T\) tag bits; where \(T\) tag bits \(= 64 - (set\) bits \() - (offset\) bits) (table 1.1).

\[
\begin{array}{c|c|c}
\text{TAG} & \text{INDEX} & \text{OFFSET} \\
\hline
6\text{bits} & \log_2(\text{Sets}) & 6\text{bits}
\end{array}
\]

Table 1.1. Address cache index bits representation.

For address: \(X = 0xAD2000\) lower 32 bit representation is shown in table 1.2.

<table>
<thead>
<tr>
<th>0000</th>
<th>0000</th>
<th>1010</th>
<th>1101</th>
<th>0010</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
</tr>
</thead>
</table>

Table 1.2. Address (AD2000) bit representation (lower 32bits).

Six rightmost bits determine the cache line’s byte offset. The next 6 bits determine the cache set index. In this example they are 0, therefore address \(0xAD2000\) maps to byte 0 in cache set 0. Figure 1.3 shows a conceptual view of cache indexing. It is important to note that any data fetch is always serviced at cache line granularity. In todays processors a cache line is likely to be 64 bytes. This means that any request for data (e.g. an 8 byte integer) will fetch 64 bytes into the cache. Figure 1.3 illustrates this concept. When a cache request cannot be serviced, also known as a cache miss, the system must fetch the data from the main memory. Cache miss is expensive both in terms of the delays and energy consumed by transferring of data between caches and main memory. Cache misses are categories such as \(\text{Compulsory, Capacity or Conflict}\) misses.

- Compulsory misses are also known as cold start misses and are typically misses that result from the first access to new data. Pre-fetching sometimes reduce the number
of cold start misses by fetching data into the cache before they are needed.

- Capacity misses are misses that cannot be avoided because they are misses caused by the limited size of caches.
- Conflict misses result when two or more data items map to the same location in a cache.

![Figure 1.3. Memory to cache concept view.](image)

As can be observed from Figure 1.3 all addresses with the same index value map to the same location. A new data will evict the current resident of that set. Because (a portion) the address of a data item determines its location in cache, we can consider assigning different addresses to conflicting data item and eliminate some of these conflicts. We will describe some techniques that can be used for data placement (and address assignment) to minimize some conflict misses. Hardware techniques based on using different address bits for indexing, or relocating data from heavily conflicting sets to underused cache sets have been proposed [29], but we will not include them in our study.

We will now define the necessary conditions for a cache conflict. An instruction trace
(T) is a sequence of one or more memory accesses caused by instructions (load, store, or modify), and an instruction window (IW) is any sequence of load or stores defined over a range of instructions T. We can define a cache conflict (CC) as any occurrence of data reference mappings into the same cache set over an IW. As an example consider 3 data accesses in the following order: \( D_a, D_b, D_a \) where \( D_a, D_b \) map to the same cache set, i.e. their index bits are identical. A cache conflict occurs if the following conditions are met:

1. The conflict involves 3 or more data references.
2. The conflict involves 2 or more data elements, \( D_i \) and \( D_j \) where \( i \neq j \).
3. \( D_i \) and \( D_j \) must map into the same cache set, that is to say, their INDEX bits are identical.

Although the last condition appears to be sufficient to cause a conflict, the conflict may not occur within the instruction window, or only one of the conflicting data items are actually accessed by the program.

1.3. Software Memory Allocation

Most modern programs rely on allocating memory for objects only when needed. The system memory manager receives requests form the application and allocates a chunk of memory from the applications heap space. In addition to the heap, the application’s address space contain other areas including: the code segment, data segment, uninitialized data segment, an application’s stack, and environment variables. Figure 1.4 shows a conceptual view of the various areas of an application and how they relate when mapped to physical memory. Note that an application always views a linear address space (or virtual address space) divided into these various segments. However pages of these virtual address spaces are mapped to physical memory pages by the OS, and the virtual segments may not be mapped to consecutive physical pages. The Code segment is the area of memory where an application’s instructions are stored. The data segment stores an application’s initialized global data, and the .BSS (Block Started by Symbol) segment is the static uninitialized memory. During runtime an application may call various functions or routines. Arguments
and other bookkeeping data associated with function calls are stored on the stack. In order to honor nested calls, stack is viewed as a LIFO structure. Thus the size of the stack area grows and shrinks as functions are called and the called functions complete their execution. As stated above, requests for dynamic memory allocations are serviced from the heap segment of user memory. The heap region is managed by the software memory manager. It is responsible to efficiently keep track of currently used and free blocks of memory.

Most memory allocators try to optimize two metrics, memory fragmentation and allocation speed. Fragmentation refers to memory areas that are wasted and cannot be used to meet applications memory needs. This can result either because the allocator grants a larger than requested memory object, or some of the available memory blocks are not sufficient to satisfy application’s requests. The time that it takes the allocator to find a block of memory in response to a request is known as allocation speed. Since this time is not contributing directly the computations of the application, we should try to minimize this time. Mini-
mizing fragmentation and minimizing allocation time are often conflicting goals. Therefore for applications that request many allocations a very fast allocator is preferred because slow allocation adds to overall application execution time. Likewise, allocators have to be very efficient when choosing blocks of memory as this may cause inefficient use of available memory. Thus for application’s that request various block sizes often an allocation scheme that reduces fragmentation is preferred. Chapter 3 covers memory allocators in greater detail. In addition to these two primary goals, we feel that it is important to understand how the cache localities are affected by where dynamically allocated objects are located in the address space. As described previously, the address of an object determines where the object will be located in the cache; and cache conflicts are caused if multiple objects map to the same cache set. This observation provides opportunities to explore allocation techniques that place objects in such a way as to minimize conflicts.

1.4. Major Contributions

To understand the issue of object placement and explore solutions, it is necessary to develop tools that can track memory accesses of program objects and their mapping to cache locations. Our research led to the conclusion that none of the existing tools met our requirements so we developed a new tracing and profiling tool called Gleipnir for tracing memory accesses and a cache simulator GL-cSim that maps the accesses to cache sets. Gleipnir provides very detailed information on every memory access and relates it back to source code objects. This framework allows researchers to better understand object placement, to refactor code or to refactor data. Gleipnir’s tracing capabilities and GLcSim’s cache simulation and object tracking capability are suitable for other optimizations, for example, we can use the data for trace-driven data structure transformation. In addition the framework supports user-interface client calls to track user-defined memory regions for application’s that use manual memory management. Similarly, global and stack variables and structures can be tracked as well.

In combination with our simulator GLcSim the framework is capable of delivering fine grained cache behavior information at multiple CPU levels for various execution intervals.
For example, we can track detailed application cache behavior that relates an application’s run-time stack, heap, and global data segment’s cache hits and misses as demonstrated in Chapter 5. This allows users to focus their optimization efforts only on relevant application phases as well as focus on segments that are most responsible for cache misses. Because of Gleipnir’s traces each segment can be further expanded into data structures allocated in each segment, for example global structures, local (stack) structures, or heap structures. Since a data structure’s memory placement and layout determines its cache mapping users need to relate object’s memory placement and its effects on the cache. Therefore, the framework supports a data-centric cache memory view by tracking every object and its corresponding hits and misses on each cache set.

The usefulness of traces can also be applied to various other research areas as well. For example, trace-driven semi-automatic data-structure transformations was demonstrated in [24]. Similarly, detailed data-structure trace information can be used to study reordering of data-structure elements to optimize cache line utilization. The trace, in combination with slight modifications of our cache simulator, can be used to study the impact of various hardware implemented cache indexing mechanisms. Additional capabilities include annotation of memory values on every load, store, modify which was used in related research that evaluated the amount of unnecessary writebacks in a memory hierarchy.

Memory allocation is a ubiquitous process in modern computer programs, thus the need for cache-efficient data placement demands a re-examination of memory allocation strategies. Since the introduction, dynamic memory allocation has not changed much. At present it uses a size parameter as the driving parameter for allocations. Similarly, the tools used by programmers focus on providing general metric for evaluating cache performance. In this dissertation, Framework for Evaluating Dynamic Memory Allocators including a new Equivalence Class based Cache-Conscious Allocator, we propose a new allocation algorithm for user-driven cache-conscious data placement. The basis of our work is that standard tools focus on providing general cache related information that fail to relate cache performance bottlenecks and block placement. Furthermore, we argue that allocation metrics such as
memory size requests are not enough when allocating complex structures, but that other parameters must be considered as well to build the next generation of cache-conscious memory allocators. Thus, we evaluate several allocator strategies with respect to these metrics.
CHAPTER 2

SURVEY OF PROFILING TOOLS

This chapter describes a few instrumentation and analysis tools that are most relevant to our research. A more comprehensive survey of commonly used tools to profile applications and tune performance of application can be found in [25].

2.1. Introduction

To our knowledge, the first paper on profiling was published in early 1980s, gprof: a call graph execution profiler[18]. The need for gprof arose out of the necessity to adequately trace the time spent on specific procedures or subroutines. At that time profilers were fairly simple and the tools only reported very limited information such as how many times a procedure was invoked. Gprof, a compiler assisted profiler, extended this functionality by collecting program timing information. Compiler assisted profiling tools insert instrumentation functions during the compilation process. Gprof’s development launched the research area of program profiling. Similar tools were developed soon after (e.g. Parasight[3] and Quartz[2]) targeting parallel applications. Another form of application instrumentation is binary translation. Binary translation can be either static binary translation or dynamic binary translation. Static translators are usually faster, but less accurate than dynamic translators. Binary translators are also known as optimization frameworks, e.g. DIXIE[17] and UQBT[13], because of their ability to translate and modify compiled code. Binary instrumentation tool research accelerated after the development of the Atom tool[45]. As the name suggests binary instrumentation operates on a binary file, unlike compilers which operate on the application’s source code. These type of tools gather various performance metrics during the binary instrumentation process using an interpreter or synthesizing the binary (machine) code into an intermediate representation. Binary instrumentation tools are different from other profiling tools because their main approach lies in injecting program executables with additional code. The inserted code is passed to plug-in tools for analyses. Similarly to binary translation, binary instrumentation can be either static, dynamic or
a combination of both. The trade-offs are also related to performance and accuracy. For static binary instrumentation the performance and accuracy trade-off is due to the inability to predict application code paths and analyze code statically. Conversely, dynamic binary instrumentation is slower because it needs to manage code at run-time, but more accurate because it benefits from application’s run-time information. For clarity purposes we can categorize hybrid tools that incorporate dynamic binary translation and binary instrumentation into hybrids or runtime code manipulation (RCM) tools. Runtime code manipulation involves an external tool (sometimes also referred to as the core-tool) which supervises the instrumented application (also referred to as the client). The instrumented code may be annotated or otherwise transformed into an intermediate representation (IR) and passed onto plug-in analysis tools. This also implies that the capability, accuracy, and efficiency of plug-in tools is limited by the framework. The benefits of runtime instrumentation, particularly dynamic binary instrumentation, lies in the level of detail of the binary code that plug-in tools can utilize. Example frameworks in this area are Pin[30], Valgrind[38], DynamoRIO[7], DynInst[8] and others. Figure 2.1 shows a diagram of general tool categories.\(^1\)

\(^1\)In this chapter we will concentrate only on instrumenting tools.
2.2. Instrumenting Tools

Instrumentation is a technique that injects analysis routines into the application code to either analyze or deliver the necessary meta-data to other analysis tools. Instrumentation can be applied during various application development cycles. During the early development cycles instrumentation comes in the form of various print statements, this is known as manual instrumentation. For tuning and optimization purposes manual instrumentation may invoke underlying hardware performance counters or operating system events. *Compiler assisted* instrumentation utilizes the compiler infrastructure to insert analysis routines, e.g. instrumentation of function boundaries, to instrument the application’s function call behavior. *Binary translation* tools are a set of tools that reverse compile an application’s binary into intermediate representation suitable for program analysis. The binary code is translated, usually at a basic block granularity, interpreted, and executed. The translated code may simply be augmented with code that measures desired properties and resynthesized (or recompiled) for execution. Notice that binary translation does not necessarily include any instrumentation to collect program statistics. The instrumentation in this sense refers to the necessity to control the client application by redirecting code back to the translator (i.e. every basic block of client application must be brought back under the translator’s control). Instrumentation at the lowest levels is applied on the application’s executable binaries. Application’s binary file is dissected block by block or instruction by instruction. The instruction stream is analyzed and passed to plug-in tools or interpreters for additional analysis. *Hybrids* are tools that are also known as *runtime code manipulation* tools. Hybrid tools apply binary translation and binary instrumentation. The translation happens in the framework’s core and the instrumentation is left to the plug-in tools (e.g. Valgrind[38]). Table ?? shows a summary of application instrumenting tools and tools found in each subcategory.

2.2.1. Valgrind.

Valgrind is a dynamic binary instrumentation framework that was initially designed for identifying memory leaks. Valgrind and other tools in this realm are also known as shadow value tools. That means that they shadow every register with another descriptive
Instrumenting tools

<table>
<thead>
<tr>
<th></th>
<th>compiler assisted</th>
<th>binary translation</th>
<th>binary instrumentation</th>
<th>hybrids/runtime code manipulation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Etch[42]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EEL[33]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1. Application profiling: instrumenting tools.

value. Valgrind is also known as a complex or heavyweight analysis tool both in terms of its capabilities and its complexities. In our taxonomy Valgrind is an instrumenting profiler that utilizes a combination of binary translation and binary instrumentation. The basic Valgrind structure consists of a core-tool (the framework) and plug-in tools (tools). The core tool is responsible for disassembling the application’s (client’s) binary image into an intermediate representation (IR) specific to Valgrind. The client’s code is partitioned into superblocks (SBs). An SB, consists of one or more basic-blocks, which is a stream of approximately 50 instructions. The block is translated into an IR and passed to the instrumentation tool. The instrumentation tool then analyzes every SB statement and inserts appropriate instrumented calls. When the tool is finished operating on the SB it will return the instrumented SB back to the core-tool. The core-tool recompiles the instrumented SB into machine code and executes the SB on a synthetic (simulated) CPU. This means that the client never directly runs on the host processor. Because of this design Valgrind is bound to specific CPU’s and operating systems. Several combinations of CPUs and OS systems are currently supported by Valgrind including AMD64, x86, ARM, PowerPC 32/64 running predominately Linux/Unix systems. Valgrind is open source software published under the GNU GPL² license.

Several widely used instrumentation tools come with Valgrind while others are designed by researchers and users of Valgrind.

- Memcheck: Valgrind’s default tool memcheck enables the user to detect memory

²GNU GPL is an acronym for: GNU's Not Unix General Public License.
leaks during execution. Memcheck detects several common C and C++ errors. For example it can detect accesses to restricted memory such as areas of heap which were deallocated, using undefined values, incorrectly freed memory blocks or a mismatched number of allocation and free calls.

- **Cachegrind:** Cachegrind is Valgrind’s default cache simulator. It can simulate a two level cache hierarchy and an optional branch predictor. If the host machine has a three level cache hierarchy Cachegrind will simulate the first and third cache level. The Cachegrind tool comes with a 3rd party annotation tool, that will annotate cache hit/miss statistics per source code line. It is a good tool for users who want to find potential memory performance bottlenecks in their programs.

- **Callgrind:** Callgrind is a profiling tool that records an application’s function call history. It collects data relevant to the number of executed instructions and their relation to the called functions. Optionally Callgrind can also simulate the cache behavior and branch prediction and relate that information to function call profile. Callgrind also comes with a 3rd party graphical visualization tool that helps visualize Callgrind’s output.

- **Helgrind:** Helgrind is a thread error detection tool for applications written in C, C++, and Fortran. It supports POSIX pthread primitives. Helgrind is capable of detecting several classes of error that are typically encountered in multithreaded programs. It can detect errors relating to the misuse of the POSIX API that can potentially lead to various undefined program behavior such as unlocking invalid mutexes, unlocking a unlocked mutex, thread exits still holding a lock, destructions of uninitialized or still waited upon barriers etc. It can also detect error pertaining to an inconsistent lock ordering. This allows it to detect any potential deadlocks.

- **Massif:** Massif is a heap profiler tool that measures an application’s heap memory usage. Profiling an application’s heap may help reduce its dynamic memory footprint. As a result reducing an application’s memory footprint may help avoid exhausting a machine’s swap space.
• DHAT: DHAT is a dynamic heap analysis tool similar to Massif. It helps identify memory leaks, analyze application allocation routines which allocate large amounts of memory but are not active for very long, allocation routines which allocate only short lived blocks, or allocations that are not used or used incompletely.

• Lackey: A Valgrind tool that performs various kinds of basic program measurements. Lackey can also produce very rudimentary traces that identify the instruction and memory load/store operations. These traces can then be used in a cache simulator (e.g. Cachegrind operates on a similar principle).

2.2.2. DynamoRIO.

DynamoRIO\[7\] is a dynamic optimization and modification framework built as a revised version of Dynamo. It operates on a basic-block granularity and is suitable for various research areas: code modification, intrusion detection, profiling, statistical gathering, sandboxing, etc. It was originally developed for Windows Os but has been ported to a variety of Linux platforms. The key advantage of DynamoRIO is that it is fast and it is designed for runtime code manipulation and instrumentation. Similar to Valgrind DynamoRIO is classified as a code manipulation framework, thus falls in the hybrid category in Figure 2.1 Unlike other instrumentation tools Dynamo does not emulate the incoming instruction stream of a client application but rather caches the instructions and executes them on the native target. Because it operates on basic block granularity DynamoRIO intercepts control transfers after every basic block. Performance is gained through various code block stitching techniques, for example basic blocks that are accessed through a direct branch are stitched together so that no context-switch, or other control transfer, needs to occur. Multiple code blocks are cached into a trace for faster execution. The framework employs an API for building DynamoRIO plug-in tools. Because DynamoRIO is a code optimization framework it allows the client to access the cached code and perform client driven optimizations. In dynamic optimization frameworks instruction representation is key to achieving fast execution performance. DynamoRIO represents instructions at several levels of granularity. At the lowest level the instruction holds the instruction bytes and at the highest level the instruction is fully de-
coded at machine representation level. The level of detail is determined by the routine’s API used by the plug-in tool. The levels of details can be automatically and dynamically adjusted depending on later instrumentation and optimization needs. The client tools operate through hooks which offer the ability to manipulate either basic-blocks or traces. In DynamoRIO’s terminology a trace is a collection of basic blocks. Most plug-in tools operate on repeated executions of basic blocks also known as hot-code. This makes sense because the potential optimization savings are likely to improve those regions of code. In addition, DynamoRIO supports adaptive optimization techniques. This means that the plug-in tools are able to re-optimize code instructions that were placed in the code-cache and ready for execution. Dynamic optimization frameworks such as DynamoRIO are designed to improve and optimize applications. As was demonstrated in [7] the framework improves on existing high-level compiler optimizations. The following tools are built on top of the DynamoRIO framework:

- Dr. Memory: Dr.Memory[6] is a memory profiling tool similar to Valgrind’s memcheck. It can detect memory related errors such as accesses to uninitialized memory, accesses to freed memory, improper allocation and free ordering. Dr. Memory is available for both Windows and Linux operating systems.
- Adept: Adept[51] is a dynamic execution profiling tool build on top of the DynamoRIO platform. It profiles user-level code paths and records them. The goal is to capture the complete dynamic control flow, data dependencies and memory references of the entire running program.

2.2.3. Pin.

Pin[30] is a framework for dynamic binary program instrumentation that follows the model of the popular ATOM tool (which was designed for DEC Alpha based systems, running DEC Unix), allowing the programmer to analyze programs at instruction level. Pin’s model allows code injection into client’s executable code. The difference between ATOM and Pin is that Pin dynamically inserts the code while the application is running,
whereas ATOM required the application and the instrumentation code to be statically linked. This key feature of Pin allows Pin to attach itself to already running process, hence the name Pin. In terms of taxonomy Pin is an instrumenting profiler that utilizes dynamic binary instrumentation. It is in many ways similar to Valgrind and other dynamic binary instrumentation tools; however, Pin does not use an intermediate form to represent the instrumented instructions. The primary motivation of Pin is to have an easy to use, transparent, and efficient tool building system. Unlike Valgrind, Pin uses a copy and annotate intermediate representation, implying that every instruction is copied and annotated with meta-data. This offers several benefits as well as drawbacks. The key components of a Pin system are the Pin virtual machine (VM) with just-in-time (JIT) compiler, the pintools, and the code cache. Similar to other frameworks a pintool shares a client’s address space, resulting some skewing of address space; application addresses may be different when running with Pin compared to running without Pin. The code cache stores compiled code waiting to be launched by the dispatcher. Pin uses several code optimizations to make it more efficient. For a set of plug-in tools an almost necessary feature is its access to the compiler generated client’s symbol table (i.e. its debug information). Unlike Valgrind, Pin’s debug granularity ends at the function level. This means that tracing plug-in tools such as Gleipnir can map instructions only to the function level. To obtain data level symbols a user must rely on debug parsers built into the plug-in tool. Pin uses several instrumentation optimization techniques that improve the instrumentation speed. It is reported in [30] and [38] that pin outperforms other similar tools for basic instrumentation. Pin’s rich API is well documented and thus attractive to users interested in building Pin based dynamic instrumentation. Pin comes with many example pintools can provide data on basic blocks, instruction and memory traces and cache statistics.

2.2.4. DynInst.

DynInst[22] is a runtime instrumentation tool designed for code patching and program performance measurement. It expands on the design of ATOM, EEL, and ETCH by allowing the instrumentation code to be inserted at runtime. This contrasts with the earlier static
instrumentation tools that inserted the code statically at postcompile time. Dyninst provides a machine independent API designed as part of the Paradyn Parallel Performance Tools project. The benefit of DynInst is that instrumentation can be performed at arbitrary points without the need to predefined these points or to predefined the analysis code at these points. The ability to defer instrumentation until runtime and the ability to insert arbitrary analyses routines makes Dyninst good for instrumenting large scale scientific programs. The dynamic instrumentation interface is designed to be primarily used by higher-level visualization tools. The DynInst approach consists of two manager classes that control instrumentation points and the collection of program performance data. DynInst uses a combination of tracing and sampling techniques. An internal agent, the metric manager, controls the collection of relevant performance metrics. The structures are periodically sampled and reported to higher level tools. It also provides a template for a potential instrumentation perturbation cost. All instrumented applications incur performance perturbation because of the added code or intervention by the instrumentation tool. This means that performance gathering tools need to account for their overhead and adjust performance data accordingly. The second agent, an instrumentation manager, identifies relevant points in the application to be instrumented. The instrumentation manager is responsible for the inserted analyses routines. The code fragments that are inserted are called trampolines. There are two kind of trampolines: base and mini trampolines. A base trampoline facilitates the calling of mini trampolines and there is one base trampoline active per instrumentation point. Trampolines are instruction sequences that are inserted at instrumentation points (e.g. beginning and end of function calls) that save and restore registers after the analyses codes complete data collection. DynInst comes with an application programming interface that enables tool developers to build other analyses routines or new performance measurement tools built on top of the DynInst platform. There are several tools built around, on top of, or utilizing parts of the DynInst instrumentation framework:

- TAU: TAU[44] is a comprehensive profiling and tracing tool for analyzing parallel programs. By utilizing a combination of instrumentation and profiling techniques
Tau can report fine-grained application performance data. Applications can be profiled using various techniques using Tau’s API. For example, users can use timing, event, and hardware counters in combination with application dynamic instrumentation. Tau comes with visualization tools for understanding and interpreting large amounts of data collected.

- **Open SpeedShop**: Open SpeedShop[43] is a Linux based performance tool for evaluating performance of applications running on single node and large scale multi-node systems. Open SpeedShop incorporates several performance gathering methodologies including sampling, call-stack analysis, hardware performance counters, profiling MPI libraries and I/O libraries and floating point exception analysis. The tool is supplemented by a graphical user interface for visual data inspection.

- **Cobi**: Cobi is a DynInst based tool for static binary instrumentation. It leverages several static analysis techniques to reduce instrumentation overheads and metric dilation at the expense of instrumentation detail for parallel performance analysis.

### 2.3. Event-driven and Sampling Tools

Sampling based tools gather performance or other program metrics by collecting data at specified intervals. One can be fairly conservative with our categories of sampling based tools as most of them rely on other types of libraries or instrumentation frameworks to operate. Sampling based approaches generally involve interrupting running programs periodically and examining the program’s state, retrieving hardware performance counter data, or executing instrumented analysis routines. The goal of sampling based tools is to capture enough performance data at a reasonable number statistically meaningful intervals so that the resulting performance data distribution will resemble the client’s full execution. Sampling based approaches are sometimes known as statistical methods when referring to the data collected.

The basic components of sampling tools include the host architecture, software/hardware interfaces, and visualization tools. Most sampling tools use hardware performance counters and operating system interfaces. Sampling based tools acquire their performance data based on three sampling approaches: timer based, event based, and instruction based. Diagram in
Figure 2.2 shows the relationships of sampling based tools.

Timer based and timing based approaches are generally the basic form of application profiling, where the sampling is based on built-in timers. Tools that use timers are able to obtain a general picture of execution times spent within an application. The amount of time spent by the application in each function may be derived from the sampled data. This allows the user to drill down into the specific program’s function and eliminate possible bottlenecks. Event based measurements sample information when predetermined events occur. Events can be either software or hardware events, for example a user may be interested in the number of page faults encountered or the number of specific system calls. These events, are trapped and counted by the underlying O.S. library primitives thereby providing useful information back to the tool and ultimately the user. Mechanisms that enable event based profiling are generally the building blocks of many sampling based tools. Arguably the most accurate
profiling representation are tools that use instruction based sampling (IBS) approach. For example AMD CodeAnalyst\cite{14} uses the IBS method to interrupt a running program after a specified number of instructions and examine the state of hardware counters. The values obtained from the hardware counters can be used to reason about the program performance. The accuracy of instruction sampling depends on the sampling rate.

2.4. Hardware Simulators

Computer architecture simulators are tools built to evaluate architectural trade-offs of different systems or system components. The simulation accuracy depends on the level of simulated detail, complexity of the simulation process, and the complexity of the simulated benchmark. Architectural simulators are generally categorized into single component simulators, multi-component simulators, or full-system simulators. We also need to mention that additional subcategories include design specific simulators aimed at evaluating network interconnects and power estimation tools. Although single component simulators are less complex compared to full-system simulators, they may require a simulation of all the component complexities to produce accurate simulation results. For example trace-driven simulators receive an input in a single file (e.g. a trace of instructions) and they can simulate the component behavior for the provided input (for example if a given memory address causes a cache hit or miss, or if an instruction requires a specific functional unit). A most common example of such simulators are memory system simulators including those that simulate main memory systems (e.g., \cite{47} to study RAM (Random Access Memory) behavior) or CPU caches (e.g., DineroIV \cite{23}).

\textit{DineroIV} is a trace-driven uni-processor cache simulator \cite{23}. The availability of the source code makes it easy to modify and customize the simulator to model different cache configurations, albeit for a uniprocessor environment. DineroIV accepts address traces representing the addresses of instructions and data accessed when a program executed and models if the referenced addresses can be found in (multi-level) cache or cause a miss. DineroIV permits experimentation with different cache organizations including different block sizes, associativities, replacement policies. Trace driven simulation is an attractive method to
test architectural sub-components because experiments for different configurations of the component can be evaluated without having to re-execute the application through a full system simulator. Variations to DineroIV are available that extend the simulator to model multi-core systems, however, many of these variations are either unmaintained or difficult to use.

2.5. Conclusions

Currently available tools did not meet our requirements in terms of providing fine-grained memory access information. Therefore, we developed our own tool *Gleipnir* which is based on Valgrind. We selected Valgrind because it is an open source tool and it is actively supported by a large user community. Also, the benefit of using single component simulators is that they can be easily modified to serve as third party components of more advance software. Therefore, we extensively modified DineroIV cache simulator in order to provide cache statistics at a greater detail. We will describe Gleipnir and the extended simulation environment (called *GL_cSim*) in Chapter 4.
CHAPTER 3

SURVEY OF MEMORY ALLOCATORS

In this chapter a number of commonly used dynamic memory allocation mechanisms will be described, with respect to their design and implementation. Memory allocators remain an essential part of any modern computer system and of interest to researchers in search of better allocation techniques. Because dynamic memory management is key to performance of applications, we will first define common metrics to compare memory management techniques; focussing on allocation and not on garbage collection. Two key metrics used to compare memory allocators are, fragmentation and locality of objects\(^1\). Yet another metric refers to the speed of the allocator itself. With changing landscape of processor architectures and memory hierarchies, the locality of objects is becoming more critical. In this chapter we will detail how dynamic memory allocation works, how current allocators differ in their approach and how they compare in terms of their performance.

3.1. Introduction

In order to understand the role memory allocators play in software and hardware systems we need to define what an allocator is, and what its role is. The basic job of any allocator is to keep track of memory that is used and memory that is free. And when a request for additional memory comes, the allocator must carve space out of the free memory to satisfy the request. Allocators only deal with memory requests created by an application dynamically, which is the preferred way in modern programming languages (using such constructs as `malloc`, or `new`). Because memory and accesses to data stored in memory are critical to the performance, a good memory allocator must optimize competing performance metrics. It must reduce the average time encountered by the application in accessing the allocated memory, and it must reduce the amount of memory wasted and cannot be used for useful allocations. And the allocator itself should be efficient in terms of the memory it needs and the time it takes to complete its job.

\(^{1}\)We refer to an object as data block of memory, not as an object in the traditional object oriented sense.
As should be obvious, allocators do not have the luxury of static analysis, (which is available to compilers that can optimize statically allocated variables), since allocators are only invoked while an application is executing. Also while garbage collection may be a critical companion of memory allocation, since garbage collection involves identifying memory spaces that are no longer used (or reachable) and compacting memory that is currently in use to free larger areas of memory for future request, in this research we will not analyze garbage collection methods.

The dynamic nature of memory requests make allocation algorithms complex. Applications may generate many requests for many different sizes and at different times in their execution; it is very difficult to predict the request patterns. As memory requests are satisfied, the original contiguous memory space available to the application becomes carved up, often leaving free areas of memory that are too small to meet future allocation requests. This phenomenon of small free areas is known as fragmentation. In subsequent sections we will expand on memory fragmentation and explain why it is potentially detrimental to performance. To our knowledge there no allocators that can be called optimum in terms of eliminating fragmentation and completing allocation requests in a reasonable amount of time. Wilson [49], demonstrated that for any memory allocation strategy, there exists an application that will adversely impact the allocator’s fragmentation efficiency or otherwise defeat its placement strategy; likewise one can find applications that cause the allocator to take unacceptably long to satisfy the request. Thus the goal of most allocators is exhibit good or acceptable behavior on the average.

3.1.1. Allocator Design

The main decision that affects an allocator’s behavior is the finding space for the allocated objects. That is, when an object of X sized bytes is requested the allocator must choose which memory location should be given to the application (i.e. where to place the object). Given that the application can free objects at arbitrary times object placement is critical because it determines available space for future allocations, otherwise very small chunks will be left which cannot satisfy most requests, causing fragmentation. One way
to reduce fragmentation is to combine small chunks when possible or *coalescing*, leaving bigger chunks of memory available for allocation; and *splitting* the bigger chunks to satisfy requests. In most systems, when the memory under the allocator control can no longer satisfy requests for memory, the allocator will request additional memory from the system\(^2\); but such operations can be very expensive in terms of execution times.

For the purpose of this chapter we will characterize some terms as they apply to allocator algorithms and implementations. A *strategy* may involve understanding or predicting the nature of requests in order to improve allocators performance. For example an application may regularly request smaller chunks, and a strategy may be to pre-allocate a number of smaller chunks (or carve up memory into small chunks) to satisfy these requests. A *policy* refers to rules that the allocator follows. For example a policy may be to the placement of objects including the placement of an object relative to another object. Policies often allow alternative choices when the primary rule cannot be met. Finally the *mechanism* specifies a set of algorithms that implement *policies*.

We can think of the strategy as a guideline for a set of policies describing an allocation mechanisms. A policy is described as a decision making mechanisms, and an allocators mechanisms is the engine (a set of algorithms and data-structures) enforcing its policies.

*Figure 3.1. Allocator splitting and coalescing concept.*

---

\(^2\)Provided that such memory is available, otherwise it will fail and the application will likely abort.
Virtually every allocator utilizes three techniques to support a range of placement policies. An allocator has three options to satisfy a given request: split, merge, or request more memory. When we cannot find adequate sized free chunk for an allocation request, a larger chunk is split into two chunks, one to satisfy the request, and the second (now smaller chunk) for possible future allocations. If the request cannot be satisfied with existing free chunks, it may be possible to combine or coalesce smaller chunks into one large free memory to satisfy the request. Figure 3.1 shows the diagram of these mechanisms. The coalescing can either occur when previously allocated objects are freed or deferred until a need arises. It is important to note that coalescing or merging of adjacent blocks may incur significant cost in terms of execution performance, hence most allocators delay the coalescing until needed.

3.1.2. Fragmentation

We must thus define what fragmentation really is and how it occurs in practice. In its basic form fragmentation are pockets of unusable memory which are the result of memory splits and memory merges. In other words, memory fragmentation can occur when $n$ objects are allocated in contiguous space, but only $n - i$; where $i > 1$ are freed such that the deallocation will carve an empty block in the contiguous space. Fragmentation can also come about due to the time-varying changes in object allocation. For example an application may request and free several smaller objects at varying times and then request larger objects. The request for larger blocks is unlikely to be satisfied in previously freed smaller blocks.\(^3\)

There are two types of fragmentations: internal fragmentation is the excess of memory allocated beyond the amount that is actually requested, and external fragmentation refers to (small) free chunks of available memory that do not satisfy any requests(see Figure 3.2). External fragmentation can be reduced using splitting and coalescing and in some cases by compacting memory currently in use. Internal fragmentation primarily results because of how the allocator selects chunks for allocation - sometimes the policy may require alignment of address on certain boundaries, and sometimes the policy may require a minimum size for available memory chunks. In some cases, the allocator may carve up the memory into

\(^3\)Unless, of course, the space can be coalesced into a satisfying large block.
chunks of different (but fixed) sizes; say 8 byte chunks, 16 byte chunks, etc. And allocation request are satisfied by selecting the smallest chunk that is adequate to meet the request. For example when a request for 14 bytes is received, the allocation will allocate a 16 byte chunk, wasting or causing a 2 byte internal fragmentation. Figure 3.3 shows this concept.

![Figure 3.2. External memory fragmentation concept.](image)

![Figure 3.3. Internal fragmentation concept.](image)

3.1.3. Locality

Most modern processors rely on the concept of *localities* to improve memory access times. Using fast, but smaller cache memories in the memory hierarchy serves this goal. Most recently and most frequently accessed data items are placed in cache memories. However, caches are designed in such a way that a given memory address (and associated object) can only be placed in a fixed set of cache memory locations. Thus what data resides in caches depends on the addresses of the objects. Thus proper location of dynamically allocated objects impacts the localities that can be achieved. In fact there are two types of localities: *spatial* localities refer to data items that exhibit spatial properties meaning that if data at address $X$ is used then nearby data will likely be requested in the near future, and *temporal* localities that refer to data that will be requested again in the near future. An example of data that exhibits temporal data are array elements. For example if we request an array
element $A_i$ we are likely to request $A_{i+1,2,...,n}$ in the near future. Similarly examples of spatial data are loop indices such as $i, j, k$. In chapter 5 we will demonstrate that allocators performance in terms of allocating objects to improve localities is very critical to the cache performance of the application.

3.1.4. Analysis

Allocators are evaluated using benchmark programs as kernels to stress test so that their performance can be measured. It may also be possible to generate probabilistic models for allocation requests, and analyze the performance of an allocator based on these probability distributions of allocation request. However [49] notes that the complexity and randomness of allocation requests generated by real applications make such models unreliable. Another option is to create synthetic benchmarks to generate allocation requests that are representative of common applications. One should be very careful in interpreting the results of these analyses to understand allocators’ performance. In general no single allocator uniformly performs well for all applications; the performance may depend on nature and frequency of requests generated by the applications. For this reason, a comprehensive evaluation should select benchmarks carefully in order to evaluate allocators under different stress conditions.

3.1.5. Memory Usage Pattern

Understanding memory allocators necessitates an overview of dynamic memory usage patterns of real application. In this subsection we will cover three common memory usage patterns originally identified by [49]. The memory in use may vary over the execution of an application, and may have peaks at different times, maintain a steady distribution or plateau throughout the lifetime, or incrementally increase or ramp up with time. Note that these three common usage patterns are well-known examples also discussed in [49]. Some application’s behavior may not exactly fit these patterns, but it is generally accepted that these patterns are sufficient to classify most applications memory usage.

Figure 3.4 is a classic example of peaks memory pattern. The amount of memory in
use varies substantially, and the peak amount of memory in use occurs at different intervals. Dijkstra is one of the benchmarks in Mibench suite [20] and implements the well known Dijkstra’s shortest path algorithm to find shortest paths in a graph. The graph is represented by an adjacent matrix representing weights with arcs connecting nodes. As the algorithm proceeds to find shortest path, it maintains list to track current shortest paths and paths already traversed. These lists require dynamic memory allocation (and deletion). This is the reason for the memory usage pattern reflected in Figure 3.4.

A second pattern quite common is a steady or plateau memory usage pattern. This behavior is observed with another Mibench benchmark, Jpeg, as shown in Figure 3.5. The benchmark implements the Jpeg image compression and decompression algorithm. The input is an image, either a compressed Jpeg image or decompressed image. The application allocates space for the input and output at the very beginning of the execution, and requires no additional memory allocations.

The third, and in our experience a less frequently occurring pattern, is the incremental memory utilization or ramp pattern. Figure 3.6 is an example depicting this pattern. We could not readily find an actual benchmark that generates such an usage pattern. So we chose to create benchmark kernels (stress-test benchmarks) creating such patterns. This can
be achieved by allocating new objects over the entire life of the application, but not release any allocated objects.

3.1.6. Allocator Mechanics

Here we describe some common structures used by allocators.

*Header fields.* In most cases allocation requests contain a single parameter, the size of the object. This information is used by the allocator to search through the available memory, find suitable space, and return the address of the space to the application. When an
object is deleted, the allocator can use the address of the object to infer other informations since most allocators use headers with allocated objects. A header filed can be several words in size.\(^4\) Depending on the system a header field may contain such information as the size of the object, status fields, pointers to other memory chunks, etc. It should be noted that the header field is included as part of the allocated object. The header fields aid in managing memory by the allocator and for locating the next available chunk of free memory. However, including headers with objects causes internal fragmentation; and can cause excessive memory overhead when applications request mostly smaller blocks of memory.

\textit{Boundary Tags.} Many allocators that support general coalescing will utilize an additional filed, often appended to the object and thus can be viewed as a footer field.\(^5\) Footer fields are useful in checking if two free chunks can be coalesced to create larger blocks of available memory. But, as with headers, footers add to memory overhead. Consider a case where, in a 64-bit architecture, 8 bytes each are used for header and footer. This means that the smallest chunk memory that can be allocated in such a system is 24 bytes long, in order to align the objects on word (64-bit) boundaries; and the only 8 bytes of this chunk can be used for the object. Now assume that this object is brought into a cache, only a third of the cache line is useful to the application.\(^6\) Figure 3.7 shows an 8 byte object surrounded by 8 byte header and footer. Because of the headers and footers only 24 bytes of a 64-byte cache line is actually used by the of application data.

\textit{Link fields within blocks.} In order to manage the list of free chunk, the allocator needs pointers connecting available chunks (in some cases, link allocated blocks of memory). These pointers can stored in the object itself, particularly when the object is not allocated. In many implementations, allocators may use doubly-linked lists to connect the available chunks. However, each chunk must be large enough to contain the pointer information (in addition to header and footer fields). There actual implementation that uses these pointers

\(^4\)On a 64-bit system the allocator found in \textit{stdlib.h} includes a 16-byte field.

\(^5\)The allocator found in \textit{stdlib.h} is an implementation of an allocator that utilizes header and footer fields.

\(^6\)Refer to chapter 1 for an explanation on cache line utilization.
to traverse the list of available memory may vary. A doubly linked list implementation makes traversing the lists very fast. In some cases, the pointers are used to create tree-like structures of available memory; an example of such implementation is known as address ordered trees. In a sense, the minimum size of a chunk of memory that can be allocated depends on the allocator, but as shown in Figure 3.7 additional fields that are not used to store the actual data for the object adversely impact the utilization of cache memories.

**Lookup tables.** Another way to manage chunks of available memory is to maintain a separate table of such blocks. Usually a lookup table is an array that is indexed using a value (eg. size). This technique relies on grouping blocks of the same size and an index to the first object of the group. A different approach uses bitmaps to an area for marking allocated (and free) chunks. But this may require additional structure to find the size of the free checks listed in the bitmaps. We will discuss this approach more in subsequent sections.

**Allocating small objects.** It was reported in [19] that most applications allocate a large number of small blocks, with very short life times; in other words the allocated objects become free (inaccessible soon after their allocation). The reason for this behavior is that an application requests memory for a number of small objects, perform some computations on these object and then discard them to save memory. An example of such behavior can be observed for Dijkstra benchmark, shown in Figure 3.4. In our experiment this application allocated about 6,000 24 byte objects at peak intervals, but retained only 568 objects during off peak intervals. This poses a challenges to the allocator in terms finding a good strategy.
for tracking available memory. Since the requests for smaller object is common in this application, an allocator may keep objects grouped by size, and keep the smaller chunks organized for fast access.

The end block of the heap. In section 1.3 we introduced the concepts related virtual and physical addresses. Although an application may have a large virtual address space, and an allocator can allocate objects out of the virtual address space, in most systems, allocator is limited to currently allocated physical address space, marked by the end of heap memory. If the allocator needs to allocate outside of this address, it must ask the system to extend the physical heap space, using system functions such as sbrk(), brk(), or mmap() system calls.

3.2. Classic Allocators

Since dynamic memory allocation has a long history, most textbooks first describe the following allocators [49]. We will call them classic allocators. Later we will describe more recent allocators and custom allocators.

- Sequential Fits (first fit, next fit, best fit, worst fit)
- Segregated Free Lists (simple segregated storage, segregated fits)
- Buddy Systems (binary, fibonacci, weighted, double)
- Indexed Fits (structured indexes for implementing fit policies)
- Bitmapped Fits (a particular type of Indexed Fits)

3.2.1. Sequential Fits

The sequential fits refers to a group of techniques that use linear linked list for available memory chunks; and this list is searched sequentially until a chunk of adequate size can be found. Sequential fit allocators are normally implemented using boundary tags to aid in splitting and coalescing chunks. The performance of these allocators becomes unacceptable when the linked list becomes very large, causing excessive traversal times. In some variations the large list is divided into several smaller (or segregated) lists, or list tracking objects
with specific property (i.e. size of objects in that list). Most sequential fit allocators pay particular attention to their block placement policies.

**Best fit.** In this approach the linked list of available chunks of memory is traversed until the smallest chunk that is sufficient to meet the allocation is found. This is often called best-fit since allocator tries to find the "best" possible free chunk. Best fit allocators minimize internal fragmentation but can leave many very small chunks which cannot be used in the future. Moreover the search time can be very high in the worst case [49], although the worst case scenario is rarely found in practice.

**First fit.** In the first fit approach, the allocator finds the first available chunk that is large enough to satisfy the request. This technique may also lead to many small checks left. But the search in this method is very fast. Fragmentation can be minimized by organizing the linked list into trees or other more complex data structures. Another important technique used to improve the speed of allocation, it may be useful to keep recently deallocated objects in a separate list (either in LIFO or FIFO form). The idea behind this technique stems from observations that future requests are likely for objects whose sizes are similar to those objects that are recently deleted; this is the phenomena that we described earlier; many small object with very short life times. It is possible to consider different data structures to further improve the performance of such allocation techniques.

**Next fit.** In this method, the allocator remembers the position in the linked list where the allocation for the last request came from; the allocator moves forward from that point in finding a chunk for the next allocation request. In order to implement this technique, the linked list of available chunks must be viewed as a circular list, so that when the end of the list is reach while using the next-fit traversal, the allocator needs to continue the search from the beginning of the list. While this technique appears to score high in terms of allocation performance, it suffers in other ways. For example in most applications, requests for the same sized objects are clustered. Thus at different points in the execution, request may be for different sized objects. The next-fit will perform poorly in those situations since it may have to travel around the linked list more often. Such an allocation may also suffer in terms
the locality of allocated objects. These methods can be improved both for their performance and localities with more complex structures tracking the available chunks.

To summarize, sequential fit algorithms differ in terms of their implementations, strategies and policies for allocating objects. The implementations use different data structures (singly linked, doubly linked, tree structures, LIFO lists, etc). They may also differ on how splitting and coalescing are handled. For example an allocator may use a threshold value while splitting large object; find a free chunk such that it is large enough not only to meet the current request, but the remaining chunk is adequate for future requests (or it is not too small to be useful in future allocations). However, as found in [49], using threshold values during splitting may be not very useful in many applications, since you either allocate an about without splitting (causing internal fragmentation), or spend excessive time in finding a suitable chunk to split. There are other, although less frequently cited, policies possible, such as the worst fit policy. Here, unlike the best-fit, the allocator finds the largest chunk of free memory from which a request is satisfied. The justification is that splitting large chunks leaves sufficiently large blocks for future allocations. This is somewhat similar to threshold technique.

Sequential search allocators are not very scalable, since the search time depends on the size of the linked list. And the size of the list depends both on the allocation request and also the size of memory that is used by the application. Consider an application that requests a large number of smaller blocks. One benchmark with such a behavior is Patricia from the Mibench suite [20]. In our experiments we recorded well over 180,000 individual allocation for very small blocks, during its life time; and the memory in use varies widely during the execution time (somewhat similar to Dijkstra application). This means that the list of available chunks varies in length, sometimes becoming very long for efficient traversal. Note that Mibench benchmarks are actually very small applications, representing applications from embedded systems domain. If we consider more complex benchmarks, such as those in the SPEC2006 suite[21], the number of objects allocated (and freed) over the life time of an application can exceed a million.
Lastly we need to consider the implications regarding the end block, i.e. the last block that is virtually separating heap from non-physically-backed memory. This block is also known as the wilderness-chunk \cite{49} and several researchers have studied the implications regarding the allocation or usage of the wilderness-chunk as wilderness-chunk preservation. Consider what happens when more memory is requested. The end block is now a new block available for application usage. The allocator may consider this block in several ways. It can either treat it as an old large block and insert it at the free list as a larger chunk, or it can carve out smaller blocks (depending on the request) and insert them at the beginning of the list for immediate use.\footnote{Queueing of the newly carved blocks is policy dependent we are naively assuming a LIFO queuing policy.} It turns out that treating the last chunk can have profound fragmentation implications.

3.2.2. Segregated Lists

Segregated lists are an allocation technique that uses an array of free lists. Each list consists of common block sizes. Requests for object blocks is searched from the array and usually the first block from the list is returned back to the application. In this subsection we will describe a variety of more common segregated list allocators such as \textit{simple segregated storage} and \textit{segregated fits}. In segregated list allocators usually the blocks are grouped by a easy to index size such as a power of two. This mechanism makes searching the array extremely fast. When a block is freed it is simply pushed back on top of the appropriate list. Note that segregated list allocators do not physically segregate blocks based on their size. Blocks are virtually segregated by the array, and interleaved in physical memory.

\textit{Simple segregated storage.} Simple segregated storage usually performs no splitting or coalescing. When the request for a block cannot be satisfied because the list is empty more memory is requested from the system using a \texttt{sbrk()} system call. The request usually involves a multiple of page sized (eg. 4096 bytes) requests. The additional memory is split into the requesting sized block and pushed on the free list for allocation. Note that while this allocation policy ensure high locality for applications that utilize pools of memory it also has a potential to cause heavy external fragmentation. There is a potential for saving
page sized memory blocks for additional sized classes by combining a memory utilization
mechanisms to track if any live-objects are still using a page. In general, however, the simple
segregated storage allocator is a very fast allocator. The obvious worst-case example is an
application that allocated large amount of different pools of memory causing high external
fragmentation.

*Segregated fits.* Like simple segregated storage method, in this technique the allocator
only needs to find the first available chink from the appropriate list based on the size of the
request. However, if there are no free chunks in that list, some allocators find larger free
chunks, and split them. The splitting of larger chunks may result in several smaller sized
chunks that can be used to meet future request. Techniques differ on which chunk to split.
We can consider techniques similar to best-fit, next-fit or worst-fit policies described with
sequential lists. Similar to splitting, it may also be possible to create larger chunks by
combining several smaller chunks; in other words, moving several elements from smaller
sized lists to create a few elements in larger sized lists. The performance of this technique
can be improved with some implementation tweaks.

The policy of fitting requests can be divided into three categories. *Exact lists* maintain
separate lists for all possible sizes of objects. However, maintaining large number of different
lists can be costly both in terms of the memory needed for these lists and the time to traverse
them, particularly if the allocator attempts to find an exact fit for every allocation request.
One variation is to maintain individual segregated lists for a few smaller sizes, but a single
combined list for all large sized chunks, since most applications tend to request a few different
sized objects, most of them tend to be smaller sizes. In *strict size classes with rounding*,
the allocator allocates objects that are rounded up to higher sizes allowing it to maintain
chunks of some fixed sizes. Since allocated objects tend to be larger than the actual requests,
this method incurs internal fragmentation, and the amount of fragmentation depends on the
sizes of objects in various lists. Similarly, in *size classes with range lists* chunks of only a
few different sizes are maintained. Often the sizes of objects in these lists differ by fixed
amount. For example one can consider lists for 8 byte, 16 byte, 24 byte, 32 byte objects. An
allocation request will be rounded up to match one of these chunk sizes and the first object from the appropriate list is returned to the application. This is one of most commonly used method in commercial systems, including the standard Gnu library.

3.2.3. Buddy Systems

Buddy systems are a special case of segregated lists that impose space and location allocation restrictions. In buddy systems [31] [32], the size of any memory chunk (live, free or garbage) is $2^k$ for some $k$. Two chunks of the same size that are next to each other in terms of their memory addresses are known as buddies. If a newly freed chunk finds its buddy among the free chunks, the two buddies can be combined into a larger chunk of size $2^{k+1}$. During allocation, larger chunks are split into equal sized buddies, until a chunk that is at least as large as the request is created. Figure 3.8 shows the splitting concept of a buddy system. At level 3 only two buddies are created, with the first buddy returned to the system. Even though the second buddy is adjacent to the newly created chunks, they cannot be merged to create a larger chunk, since buddies whose starting addresses differ in one bit can me merged. This restriction allows for very fast coalescing and splitting and minimizes the number of lists needed to track the chunks. This technique lends itself well for hardware implementations [27]. Large internal fragmentation is the main disadvantage of buddy allocators. It has been shown that as much as 25% of memory is wasted due to fragmentation in buddy systems [28]. An alternate implementation, Double Buddy [28] which create buddies of equal size, but does not require the sizes to be powers of 2, is shown to reduce the fragmentation by half [40] [50]. Binary buddies are probably the simplest and best known buddy system allocation mechanism. Each block size is a power of two, and each size has two equal parts. This makes computing address offsets very easy because every block is aligned on a power of two address from the start of the heap address, thus every block address offset bit represents the memory hierarchy level of one block. The drawback to binary buddy system is the relatively high internal fragmentation. Consider an allocation of a 32byte block plus the header field (usually 8bytes) the entire block will be rounded up to 64bytes. That’s a $\sim 47\%$ overhead; in fact, it was noted in [32] that an average expected
overhead for binary buddy systems is roughly 28%.

**Figure 3.8.** Buddy system memory splitting, concept.

*Fibonacci buddies* is another approach for reducing internal fragmentation; the sizes of the different chunks are based on fibonacci series. Yet another variation that tries to reduce internal fragmentation is the *weighted buddies*. The weighted buddy system takes advantage of the fact that between a binary series, another series *3 times power of two* can be found. For example a regular within the binary series of 2, 4, 8, 16, 32, ... we can define another series that differ by multiples of 3, eg. 3, 6, 12, 24.... Using this observation this technique maintains two levels of lists: at the first level we use lists similar to binary buddies, and within each, we maintain chunks that are multiples of 3. This greatly reduces internal fragmentation, since allocation requests can be rounded up to a smaller value than the next power of 2; however, this is less efficient in terms of coalescing and searching through the lists. *Double buddies.*

The double buddy system is an extension of the weighted buddy system. It utilizes the same class series as the weighted buddies except that the splitting policy is different. In a double buddy system a block can only be split in half. This means that any block that belongs to the binary series will be split into two subblocks from that series. Any block that is part of the *3 times power of two* series will be split into subblocks of that series. During an allocation request the size is rounded up to the nearest size of either class series and accommodated only from that series. The aim with this allocation is to reduce the internal fragmentation
of buddy systems. The implication is that space used from one size can only be merged with blocks from the same series. Several drawbacks are immediately apparent, blocks of memory that were defined for a particular size series will remain defined for that series for the duration of the application. An optimization of that approach is to keep track of live objects for a larger area of memory and when all blocks are freed the allocator can choose to redefine the series. *Estranged buddy system* is an adaption of the buddy system that delays block coalescing until larger storage blocks are needed. The idea of estranged buddy systems is that smaller blocks are short lived and will need to be reallocated more often and thus it may be beneficial to delay block coalescing. A newly freed block is viewed as *estranged* from his buddy and therefore coalescing is postponed. This system was proposed as means to tackle the growing need for fast lightweight allocators in Java application for embedded systems [15].

3.2.4. Indexed and Bitmapped Fits

*Indexed fits* technique is based on better indexing schemes for locating a chunk in a linear list used for sequential fit allocators as described in 3.2.1 section. Indexing often combines size of the chunk and the starting address of the chunk to efficiently traverse the list. An example implementation of this approach is described in [19]. Another good example of indexed fits is the Stephenson’s *fast fits* allocator which uses Cartesian trees sorted by size and address, or the segregated list allocator which implements an exact fit policy. *Bitmapped fits* technique uses bitmaps in place of indexes for available memory chunks. An example use will involve allocating fixed sized objects from large chunk of memory, and track which objects are allocated or available using a bit map. However, Wilson [49] state that they are not common in commercial system. However, some BSD systems use pages from which smaller objects (of a given size, say 8 bytes) are allocated, and a bit map is kept as in the header portion of the page [35]. Bitmap fits are not generally used in common allocators, but they are more common in garbage collectors. The reason is that there is a perception that bit-mapped allocators are slow because the bitmap must be traversed to find a suitable length of allocatable blocks. Another problem is that a bit-mapped fit allocator must be accompanied
by a secondary structure that will keep block size information. Further optimization may, in fact improve the bitmap speed by utilizing a combination of page sized arena allocation and allocate single sized chunk from these respective arenas. The other advantage is for application that allocate a number of very small objects because the bit-mapped allocators incurs very little overhead. Fragmentation may also be alleviated due to the ability to search for free blocks based on addresses.

*Custom Allocators* In an attempt to improve the performance of specific applications, several custom allocators that rely on the behavior of the application in terms allocation requests and accesses to these objects. While they may perform well for the specific application, in most cases their performance for other applications can be very poor [16].
Embedded and high performance applications often require fine-tuning to improve their performance. This is achieved using analysis tools that provide insights into the application’s behavior. A common approach is to instrument the application code and observe its behavior during an actual execution on a target system. Refer to Chapter 2 for more detail about analysis tools.

In this chapter we describe a program profiling and tracing tool called Gleipnir. Gleipnir is built as a plug-in tool to a widely used binary instrumentation framework called Valgrind. Gleipnir can be used to trace memory accesses and associate each access with a specific program internal structure such as threads, functions, local, global, and dynamic data structures, and scalar variables. This ability makes Gleipnir a good candidate for advanced memory performance tuning. The data provided by Gleipnir may be used by trace-driven simulators, such as cache simulators to analyze accesses to data structure elements so that programmers can understand how the memory access patterns are impacting the execution time of the application. The programmer may then be able to change data layouts or reorder code to change the access patterns and eliminate performance bottlenecks. The goal of Gleipnir is to give information rich traces that can be used by any number of advanced memory analysis tools, particularly cache simulators. Our claim is that despite advances in allocation techniques and data reordering, detailed dynamic and static memory behavior of applications is often not readily available or available only in terms of statistical average accesses and cache miss rates.

Optimizing cache performance at all levels is very important to improving the performance of applications running on single-core and multi-core processors. In this chapter we will describe Gleipnir and provide examples on how the output of Gleipnir can be used by cache simulators to understand the impact different data organization on cache statistics. The overall goal of this research is to develop techniques that can be used by application
developers as well as compilers and runtime systems to improve performance of applications running on single and multi-core processors.

4.1. Introduction

Processor-Memory speed gap is still a major hurdle to performance. Software developers need to analyze application’s memory access behavior to maximize the execution performance. For this purpose software developers often rely on instrumentation and profiling tools. These tools are used in combination with other performance optimization, tuning, and analysis software. For example, today’s processors are shipped with hardware performance counters. There are architecture specific counters that count various hardware related events. For example cache miss counters may count the number of cache misses occurring during a program execution. The programmer can access the hardware counters directly from the application by setting specific register values and executing a system interrupt. The data gathered by the counters is collected and analyzed off line. Because the process of setting up hardware counters is cumbersome, designers developed tools, e.g. performance counter libraries such as the performance application programming interface (PAPI) [36], that provide application programing interfaces (API) for easier hardware counter manipulation.

To differentiate the different terminologies in this chapter we will refer to the profiling tool, or simply tool, as the instrumented code inserted by the framework. We refer to the framework as the underlying mechanism, or core tool, that enables the profiling tool to insert instrumented code. We refer to the client as the application that is instrumented.

4.1.1. Fine-grained Memory Access Analysis

There are many frameworks that enable the development of memory access analysis tools. Plug-in tools are limited by the ability of the framework to expose application related information. For example, exposing only address traces limits the tools in performing more advanced analysis. When using basic traces, i.e. access type, address, and size, a cache simulator is limited to simulate only basic information such as overall behavior, number of
memory read and writes due to cache misses, number of capacity, compulsory, and conflict
misses, etc. For advanced cache analysis we need tools that offer greater detail about an
application’s cache behavior. Valgrind’s framework comes with tools such as Cachegrind [39]
and Callgrind [48] that analyze an application’s cache behavior. These tools can classify the
collected data per source code line, function, etc. However, they fail to relate the source
of the cache misses back to the root cause, such as which data items are conflicting and in
which order.

Similarly, Dprof [46] is a tool that identifies cache critical data structures by tracking
instructions that resulted in most misses. Dprof uses hardware performance counters and
correlates this information by translating instructions that resulted in a cache miss back
to source code. This is done by tracing the instruction address and using a debug parser.
However, the tool does not identify conflicting data structures.

In order to understand and simulate this behavior we must correlate a data access to
its source code level data structure name, and we must keep track of all evictions that persist
throughout a program’s runtime. Moreover, to enable visual representations we must also
keep track of specific cache sets and corresponding subsets. Providing this information can
help programmers and application developers visualize conflicts of program data structures.
Therefore, when we set out to develop Gleipnir we aimed at exposing as much information
as possible to cache and memory analyzers. We achieved this using a binary instrumenta-
tion framework that comes with the necessary infrastructure to enable fine grained memory
tracing. We opted for Valgrind predominately because of the ability to easily track low level
debug information and the availability of source code which can be modified for our purpose.

Our goal is to provide detailed tracing information for every memory access and relate
the access back to a source code data elements. In our earlier Gleipnir version [26] we were
able to trace accesses and relate them to stack and global variables. The current version
can trace accesses to stack, global, and dynamically allocated objects and relate them back
to source code variable and structure names. We have also greatly enhanced our simulation
environment to trace fine-grained cache usage. Using Gleipnir’s traces cache simulators can
analyze all data structure related cache misses and relate misses to their root cause. Our approach is different from existing tools because Gleipnir itself does not offer trace analysis, but instead offers information rich enough with traces for use by analysis tools. To illustrate how Gleipnir traces can be used, we modified DineroIV [23], a trace-driven cache simulator. In this chapter we outline some analyses that can be performed using the traces generated by our Gleipnir tool. However we feel that many more analyses can be developed using the very rich information provided by Gleipnir, and it is fairly easy to modify Gleipnir to generate additional information with traces.

4.2. Implementation Overview

4.2.1. Valgrind’s Intermediate Representation

Valgrind’s framework consists of a core tool and one or more instrumentation tools. The core tool operates on sections of code blocks known as *super blocks* (SB), a collection of up to 50 instructions with a single entry and multiple exits. Super blocks are disassembled and converted to an intermediate representation (IR). The IR is used by the instrumentation tools for inserting code to collect relevant information. The core tool recompiles (or resynthesizes) the instrumented IR block. The recompiled code with instrumentation is then executed on a simulated processor. The execution of the application and the instrumentation code provides the desired information about the application. Figure [4.1] shows a diagram describing this process.

![SuperBlock flow chart.](image)

Valgrind and client application operate in the same user-space which means that there are several implications. Every instruction is simulated on a synthetic CPU, thus
there is an inherent slowdown (of at least 4 times) because Valgrind transforms and executes the transformed instruction stream on a simulated CPU. Figure 4.2 shows the conceptual difference of executing applications with and without Valgrind.¹

**Figure 4.2.** Valgrind client state: without Valgrind (left) and with Valgrind (right).

### 4.2.2. Tracing Instructions

Valgrind’s native tools operate on essentially the same principle that is used to trace instructions. Similarly, Gleipnir instruments instruction events identified by Valgrind’s IR. However, to map instruction with proper source code symbols it also utilizes debug symbol parsing and dynamic memory allocation wrapper functions to identify and track source-code level dynamic structures. Every event in the instrumented code is either an *instruction read* (Ir), *data read* (Dr), *data write* (Dw), or *data modify* (Dm). When the instrumented code is executed an instruction or data address is passed to Valgrind’s debug parser. The debug parser searches the symbol table and returns relevant source code data structure information back to Gleipnir. We modified Valgrind’s debug parser functions for our purpose to easier trace debug symbol information. Symbol table look-up enables Gleipnir to deliver fine grained debug information for each data write, read, or modify. Table 4.1, 4.2, and 4.3 show the format of typical trace-lines generated by Gleipnir. In Table 4.1 the first field indicates the access type which can be a data *load, store, modify, or (other) instructions*. The second field is the *virtual address*² of the data being accessed followed by its *size* and the executing thread (*thread id*). Table 4.2 are debug generated instruction descriptions. They relate the instruction or data access to the originating *segment* (stack or global), followed

¹We refer to applications simple as clients because we service them through the framework.

²Gleipnir is also capable of producing physical addresses as shown in subsection 4.2.4 for shared memory analysis.
by function name of the executed instruction. If the instruction contains any debug symbol information then the trace line will be annotated with the element’s scope, local (L) or Global (G), and the element’s type variable (V) or structure (S). An access to a dynamically allocated structure will be annotated by a dynamic specific trace-line as shown in Table 4.3. The trace-line is similar to the standard trace-line except that the scope is heap (H) followed by the allocation number. The enumerated number is used to identify the number of uniquely allocated objects of the same structure. The last field is the name of the allocated structure. Note that unlike static structures, dynamic structures are supplemented with the byte offset of the identified structure rather than the element’s name.

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Address</th>
<th>Size</th>
<th>Thread Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load (L)</td>
<td>Virt. Address</td>
<td>size</td>
<td>tid</td>
</tr>
<tr>
<td>Store (S)</td>
<td>Phys. Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify (M)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. (I)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.1.** Gleipnir’s basic trace line.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Function</th>
<th>Scope</th>
<th>Variable or Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack (S)</td>
<td>Func. Name</td>
<td>Local Variable (LV)</td>
<td>Variable name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Global Variable (GV)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Local Structure (LS)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Global Structure (GS)</td>
<td></td>
</tr>
<tr>
<td>Global (G)</td>
<td>Func. Name</td>
<td>Local Variable (LV)</td>
<td>Variable name</td>
</tr>
</tbody>
</table>

**Table 4.2.** Gleipnir’s advanced trace line for stack and global data.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Function</th>
<th>Scope-No.</th>
<th>&lt;Dynamic name&gt;.byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap (H)</td>
<td>Func. Name</td>
<td>H-(no.)</td>
<td>&lt;dynamic name&gt;.offset</td>
</tr>
</tbody>
</table>

**Table 4.3.** Gleipnir’s advanced trace line for dynamic data.

We need to mention that Gleipnir can also insert special trace-lines annotated by an X access type that serve as keywords intended to supplement simulation or analysis software. For example special keywords may indicate when an allocation occurs, a thread spawns or joins, or a client executed a fork() or other system calls. In addition Gleipnir can insert
special, X instructions executed, keyword that can help multi-core software order instruction execution. Gleipnir relies on Valgrind’s internal debug parser; therefore, application analysis that need local or global structure and variable trace information must be compiled with the compiler’s -g flag.

4.2.3. Tracing Static, Global, and Dynamic Data

Because local and global data names are known at compile time the compiler will use symbol tables to store information about these variables. However; debug information is not available for dynamically allocated structures. This requires that we instrument and trace every allocation routine using wrappers. There are two ways that Gleipnir is able to deliver dynamic structure information: we can automatically track every allocation and trace the calling function back to the source code line that invoked the allocation and extract the name of the structure directly from the source code if the name is available. We can also use user-generated macros to insert appropriate names before every allocation. This will trigger tool request mechanisms and provide Gleipnir with the necessary information.

Listing 4.1 shows an example source code that references only local and global structures. Listing 4.2 shows a portion of the trace generated by Gleipnir for the source code in Listing 4.1. The listings shows how Gleipnir relates the source level information with the program’s memory accesses. This information can be useful for analysis tools that aim to improve application’s memory performance by observing the accesses to various program variables. The source code in Listing 4.1 and 4.2 include only stack and global variables. This example is included so that the reader can observe the level of detail provided by Gleipnir. The source code’s main function starts with a Gleipnir specific client request macro that turns the instrumentation on. With the macros GL\_START\_INSTR and GL\_STOP\_INSTR the user can chose to instrument only the relevant parts of the application. The function functionA is defined in lines 8 – 15. The main function declares a structure and a scalar variable in lines 17 – 30. The structures and scalar elements are accessed in lines 22 – 28. Lastly, functionA is called in line 21 and the corresponding code is executed in lines 8 – 16. The program’s execution is observable from the corresponding trace in Listing 4.2. Each trace
line represents the data elements accessed during the program execution. The trace starts with the main function storing a value to the local variable \textit{main\_local} (see trace-line 3 – 4) which corresponds to the source code line 20 on the left.

Listing 4.1. Example source 1

<table>
<thead>
<tr>
<th>int my_gl, my_glArray [10];</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct type{</td>
</tr>
<tr>
<td>int A;</td>
</tr>
<tr>
<td>int B[10];</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>struct type GStruct [2];</td>
</tr>
<tr>
<td>void functionA ( int param1 )</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>int fnc_local;</td>
</tr>
<tr>
<td>param1 = 12345;</td>
</tr>
<tr>
<td>fnc_local = 1234;</td>
</tr>
<tr>
<td>my_gl = 543;</td>
</tr>
<tr>
<td>my_glArray [2] = 123;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>int main ( void )</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>GLEIPNIR_START_INSTRUMENTATION;</td>
</tr>
<tr>
<td>int main_local = 100;</td>
</tr>
<tr>
<td>functionA ( main_local );</td>
</tr>
<tr>
<td>my_gl = 234;</td>
</tr>
<tr>
<td>my_glArray [2] = 123;</td>
</tr>
<tr>
<td>GStruct [1].A = 5;</td>
</tr>
<tr>
<td>GStruct[0].B[3] = 123;</td>
</tr>
<tr>
<td>GLEIPNIR_STOP_INSTRUMENTATION;</td>
</tr>
<tr>
<td>return 0;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

FunctionA is traced in lines 6 – 13 and the trace lines show the data stores to the functions parameters \textit{param1} followed by several local and global data stores. At each line the trace identifies the segment of the data (stack or global), the thread id, the executing function, the scope of the accessed variable \textit{LV, LS, GV, or GS}, and the name of the variable, structure, or structure's element including the array index. Because debug information is not available for dynamically allocated objects, Gleipnir captures the necessary information using wrappers around memory allocation functions (e.g. malloc, calloc, realloc, etc.). When a \texttt{malloc}() is executed Gleipnir will intercept the call and record the allocated structure's
size and the base address. If the user provided a client request (e.g. `GL_RECORD_MSTRUCT()`) Gleipnir will record the object with the supplied name; otherwise, it will attempt to use the source code line of the callee to extract the structure’s name and enumerate the instance of the allocated object. This allows Gleipnir to track any subsequent access to dynamic memory regions. Thus when an access to a dynamic region is encountered the address of the accessed data is used to locate the object’s name and its enumerated instance. Therefore, a dynamic trace line is different from a stack or global trace line in terms of the related debug information. Dynamic debug information is replaced by the structure name if applicable and the byte offset of the object’s element.

```
typedef struct _str1 {
    int val_int;
    char val_char;
    double val_double;
    struct list_el *next;
} S1;

typedef struct _str2 {
    int val_int;
    char val_char;
    double val_double;
    struct list_el *next;
} S2;

int main (void)
{
    S1 * ptr;
    S2 * ptr2;
    S2 * ptr3;
    ptr = malloc (sizeof (S1));
    GL_START_INSTR;
    ptr->val_int = 123;
    ptr->val_char = 'G';
    ptr->val_double = 23.24;
    GL_STOP_INSTR;
    GL_RECORD_MSTRUCT("CRIT_S2");
    ptr2 = malloc (sizeof (S2));
    GL_RECORD_MSTRUCT("MY_S2");
    ptr3 = malloc (sizeof (S2));
    GL_START_INSTR;
    ptr2->val_int = 123;
    ptr2->val_char = 'G';
    ptr2->val_double = 23.24;
    GL_STOP_INSTR;
    return 0;
}
```

Listing 4.3. Example source 2

```
START PID 11835
X THREAD_CREATE 0:1
X 1 MALLOC 004225010 24 _(_S1)_.0
S 7ff0004d0 8 1 S main
L 7ff0004d8 8 1 S main LV ptr
S 004225010 4 1 H main H-O _(_S1)_.0
L 7ff0004d8 8 1 S main LV ptr
S 004225014 1 1 H main H-O _(_S1)_.4
L 7ff0004d8 8 1 S main LV ptr
S 004225018 8 1 H main H-O _(_S1)_.8
S 7ff000490 8 1 S main LS _zzq_args [0]
S 7ff000498 8 1 S main LS _zzq_args [1]
S 7ff00049c 8 1 S main LS _zzq_args [2]
S 7ff0004a0 8 1 S main LS _zzq_args [3]
S 7ff0004a8 8 1 S main LS _zzq_args [4]
S 7ff0004b0 8 1 S main LS _zzq_args [5]
S 7ff0004b8 8 1 S main LS _zzq_args [6]
S 7ff0004c0 8 1 S main LS _zzq_args [7]
S 7ff0004c8 8 1 S main LS _zzq_args [8]
S 7ff0004d0 8 1 S main

X 1 MALLOC 004225030 24 CRIT_S2 0
X 1 MALLOC 004225050 24 MY_S2 0

S 004225030 4 1 H main H-O CRIT_S2 0
L 7ff0004e0 8 1 S main LV ptr2
S 004225034 1 1 H main H-O CRIT_S2 4
L 7ff0004e0 8 1 S main LV ptr2
S 004225038 8 1 H main H-O CRIT_S2 8
S 7ff0004a0 8 1 S main LS _zzq_args [0]
S 7ff0004a8 8 1 S main LS _zzq_args [1]
S 7ff0004b0 8 1 S main LS _zzq_args [2]
S 7ff0004b8 8 1 S main LS _zzq_args [3]
S 7ff0004c0 8 1 S main LS _zzq_args [4]
S 7ff0004c8 8 1 S main LS _zzq_args [5]
S 7ff0004d0 8 1 S main

X INST 40
```

Listing 4.4. Example trace 2
Listing 4.3 is an example program that allocates several structures and accesses their elements. Similarly, Listing 4.4 shows the corresponding trace when the program in Listing 4.3 is executed. The code defines simple structures with several elements in lines 1−12. The structure is allocated in lines 21, 29, and 31. Its elements are accessed via pointers in lines 23–25 and again in lines 34–37. Listing 4.4 shows the generated trace. Notice that we stopped the instrumentation prior to allocating the structures. While this is not necessary it helps keep the trace concise. Note that Gleipnir is fully capable of tracing program library variables and structures provided that the debug information is available. We can observe the executed code in Listing 4.4. The first instruction is a special keyword that indicates an allocation. The access type X indicates keywords that are identified by the simulator or other analyses tools. The MALLOC keyword indicates that a memory allocation was recorded. It shows the objects return address (0x42250), size (24), name (_S1_), and enumerated instance (0). We can observe the object accesses in lines 5–10. Lines 19–20 show the effects of recording the same structure under a different name. This helps track specific object allocations particularly when a program allocates many instances of the same object.

4.2.4. Multi-threading

Gleipnir is a Valgrind plug-in instrumentation tool which means that its tracing ability is constrained by the limitations of Valgrind. Valgrind fully supports multi-threading and many of Valgrind’s tools are used for tracing POSIX and OpenMP style programs. However, Valgrind is not a multi-threaded platform. This means that thread execution under Valgrind is serialized and thread scheduling is left to the operating system or the user can (using Valgrind specific flags) enforce a fair scheduling mechanisms. Therefore, tracing multi-threaded applications will result in traces with interleaved thread instructions. This implies that analysis tools and cache simulators will have to rely on instruction interleaving models (e.g., interleaving on every execution cycle).

The source code in Listings 4.5 and 4.6 shows an example program that uses multiple threads. Listing 4.6 shows the program’s main function spawning multiple threads in a
for loop, lines 8 – 14. After spawning all threads the main thread will wait for a lock to be released and exit (see lines 15 – 19). This example demonstrates the trace instruction interleaving when multiple threads are executed. Listing 4.5 shows the thread’s source code. Threads call a sumnumber() function and release the lock before exiting.

Listing 4.5. Example source 3a

```
#define NUM_THREADS 5

int lock = NUM_THREADS;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void sumnumber(int sum){
    int i;
    int _sum = 0;
    for (i = 0; i < sum; i++){
        _sum += i;
    }
    return;
}

void * thread(void * threadid) {
    GL_START_instr;
    long tid = (long) threadid;
    int i;
    sumnumber(tid + 5);
    GL_STOP_INSTR;
pthread_mutex_lock(&mutex);
    GL_START_INSTR;
    lock--; /* release lock */
    GL_STOP_INSTR;
pthread_mutex_unlock(&mutex);
pthread_exit(NULL);
}
```

Listing 4.6. Example source 3b

```
int main(int argc, char *argv[1]) {
    int i = 0, rc;
    long t = 0;
    pthread_t thread_ids[NUM_THREADS];

    /* create threads */
    for (t = 0; t < NUM_THREADS; t++){
        GL_STOP_INSTR;
        rc = pthread_create(&thread_ids[t],
                            NULL, thread,
                            (void*) t);
        GL_START_INSTR;
    }

    while (lock > 0){
        GL_STOP_INSTR;
pthread_mutex_unlock(&mutex);
pthread_mutex_lock(&mutex);
        GL_START_INSTR;
    }
    GL_STOP_INSTR;
pthread_exit(NULL);
    return 0;
}
```

Listings 4.7 and 4.8 show Gleipnir generated traces for the code in Listings 4.5 and 4.6. Notice the keyword indicating a newly created thread in Listing 4.7, when a thread is created Gleipnir tracks the event with a X THREAD_CREATE 1:2 keyword which indicates that thread 1 created thread 2 that established a parent – child ordering. These instructions aid in modeling a multi-threaded cache behavior. Depending on the thread mechanism the newly created threads will compete over the CPU time slice. In our example we have enabled a fair scheduling mechanism that enqueues newly created threads in an execution queue. We can
observe the thread execution in trace lines 14 – 30 where the trace-lines thread id changes as threads are being executed. Again, for clarity purposes we omitted the POSIX pthread library calls.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>START PID 12068</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X THREAD_CREATE 0:1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X 1 MALLOC 004225010 272 <em>sysres</em> 0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X 1 CALLOC 004225010 272 <em>sysres</em> 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>X THREAD_CREATE 1:2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>X FORK 12069</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>S 7ff0004d8 8 1 S main LV _zzq_result</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L 7ff0004d8 8 1 S main</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>M 7ff0004e0 8 1 S main LV t</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>L 7ff0004e0 8 1 S main LV t</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>S 7ff000480 8 1 S main LS _zzq_args[0]</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>S 7ff000488 8 1 S main LS _zzq_args[1]</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>S 7ff000490 8 1 S main LS _zzq_args[2]</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>S 7ff000498 8 1 S main LS _zzq_args[3]</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>S 7ff0004a0 8 1 S main LS _zzq_args[4]</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>S 7ff0004a8 8 1 S main LS _zzq_args[5]</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>S 7ff000478 4 1 S main</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>S 7ff000478 4 1 S main</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>M 7ff0004e0 8 1 S main LV t</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>L 7ff0004e0 8 1 S main LV t</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>S 7ff000480 8 1 S main LS _zzq_args[0]</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>S 7ff000488 8 1 S main LS _zzq_args[1]</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>S 7ff000490 8 1 S main LS _zzq_args[2]</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>S 7ff000498 8 1 S main LS _zzq_args[3]</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>S 7ff0004a0 8 1 S main LS _zzq_args[4]</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>S 7ff0004a8 8 1 S main LS _zzq_args[5]</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>S 7ff000478 4 1 S main</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>S 7ff000478 4 1 S main</td>
<td></td>
</tr>
</tbody>
</table>

**Listing 4.7. Example trace 3a**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X THREAD_CREATE 1:3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X FORK 12070</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S 000c06e0 8 2 S thread</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L 000c06e8 8 2 S thread LV threadid</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S 000c06e8 8 2 S thread LV tid</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>L 000c06e8 8 2 S thread LV tid</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>S 000c06e8 8 2 S thread</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S 000c06e0 8 2 S sumnumber</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S 000c06e0 8 2 S sumnumber</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S 000c06e0 8 2 S sumnumber</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>S 000c06e0 8 2 S sumnumber</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>S 000c06e0 8 2 S sumnumber</td>
<td></td>
</tr>
</tbody>
</table>

**Listing 4.8. Example trace 3b**

### 4.2.5. Multi-process Capabilities

Multiprocess capabilities are also fully supported. Applications that create multiple processes using system calls, e.g. `fork()`, can be traced by Gleipnir (to the extent permitted by Valgrind). Valgrind will fork its own instrumenting image so that each process can be separately instrumented. Gleipnir includes a mechanisms to detect when a child process is created. This will result in a new trace being created for each child process along with traces for the parent process. The parent trace will be annotated with a keyword `X FORK PID`, where `PID` is the process’ id number as illustrated in Figure 4.3. This allows cache simulators to
model multi-process applications.

4.3. Analysis Environment

The traces produced by Gleipnir can be used by other analysis tools. In our research we are interested in the cache memory performance and we use a modified version of DineroIV for our purpose. Our modifications include mechanisms to accept Gleipnir’s traces thereby extending its simulation capabilities and track any relevant events provided by Gleipnir. In this section we will illustrate our analyses: because Gleipnir provides very fine-grained access information, we can analyze the cache hits and misses per function or even per program variable and understand cache conflicts among variables. Such analyses can then be used to change data layouts, change the disposition of elements of a structure or re-factor code to change access patterns.

4.3.1. Analysis Cycle

A typical analysis procedure involves three steps as outlined in Figure 4.4. A user runs the application through Gleipnir. This generates the trace file which can be analyzed by appropriate analysis tools. At present we provide the modified Dinero IV along with Gleipnir. Other analysis tools can be used with Gleipnir, provided the analysis tool accepts traces as
generated. The cache simulation results can be plotted with various plotting tools (e.g. GnuPlot). Plotting the graphs is supplemented through scripts that parse cache simulation results.

![Diagram](image.png)

**Figure 4.4.** Tracing and analysis cycle.

### 4.3.2. Cache Behavior

To understand the usefulness of Gleipnir’s traces, we must touch on the subject of processor cache indexing. A processor’s cache is a small intermediate memory unit indexed using some of the address bits of the accessed object, the index bits normally rely on modulo arithmetic. In our previous work [37] we have shown that the cache accesses of most applications are non-uniform, causing some cache lines (or sets) to be underutilized while other cache lines are overutilized. The heavily accessed cache lines cause most of the cache conflicts and cache misses. Cache misses can be minimized if the cache accesses are distributed more uniformly. This can be achieved either by changing how cache indexes work (requiring hardware modifications [29]), or by changing the data layout of conflicting data structures. Using Gleipnir generated traces we can explore both research areas.
4.3.3. Cache Simulation

Observing how an application uses cache memory requires cache simulators\(^3\). Trace-driven simulators take address traces as input, map each trace to a cache access, and identify if the access is a hit or a miss. The accuracy of the statistics regarding hits and misses and the level of detail about hits and misses provided depend on the capabilities of the cache simulators.

For our purpose, we modified a widely-used, albeit simple, cache simulator. We extended the simulator to accept Gleipnir traces and track cache hits and misses in greater detail. For example, we can track hits and misses per function, or each program variable including dynamically allocated structures. We report the cache statistics hierarchically: starting with summary results for the entire program at the highest level, to per program variable at the lowest level.

--- Simulation begins.  
2000000, 4000000, 6000000, 8000000  
10000000, 12000000, 14000000  
--- Simulation complete.  

<table>
<thead>
<tr>
<th>l1-dcache</th>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Demand Fetches</td>
<td>14542192</td>
<td>14542192</td>
<td>11062461</td>
<td>3479731</td>
</tr>
<tr>
<td></td>
<td>Fraction of total</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.7607</td>
<td>0.2393</td>
</tr>
<tr>
<td></td>
<td>Demand Misses</td>
<td>109144</td>
<td>109144</td>
<td>68824</td>
<td>40320</td>
</tr>
<tr>
<td></td>
<td>Demand miss rate</td>
<td>0.0075</td>
<td>0.0075</td>
<td>0.0062</td>
<td>0.0116</td>
</tr>
</tbody>
</table>

Table 4.4. Cache simulation results.

We illustrate our analysis using the Jpeg MiBench benchmark. We focus on cache accesses caused by the benchmark’s stack, heap, and global address spaces. When the simulation completes the first output shown is the overall cache statistic for every cache level as shown in Table 4.4. The user can then observe how well the cache is utilized. The user can then identify the functions of the application that are causing most cache accesses and misses. This data for Jpeg is shown in Table 4.5; sorted by the number of accesses and misses. In addition to showing the function names, the data shows the number of accessed variables within these functions.

\(^3\)Other analysis techniques involve models, static analysis, and hardware performance counters.
<table>
<thead>
<tr>
<th>Accesses</th>
<th>Hits</th>
<th>Misses</th>
<th>Miss %</th>
<th>Function / Segment Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>2993864</td>
<td>2909921</td>
<td>83943</td>
<td>2.80</td>
<td><em>HEAP</em> 0</td>
</tr>
<tr>
<td>3358960</td>
<td>3315472</td>
<td>43488</td>
<td>1.29</td>
<td>encode_mcu_AC_refine 13</td>
</tr>
<tr>
<td>10878411</td>
<td>10859373</td>
<td>19038</td>
<td>0.18</td>
<td><em>STACK</em> 0</td>
</tr>
<tr>
<td>1148436</td>
<td>1129715</td>
<td>18721</td>
<td>1.63</td>
<td>encode_mcu_AC_first 9</td>
</tr>
<tr>
<td>1520128</td>
<td>1507179</td>
<td>12949</td>
<td>0.85</td>
<td>rgb_ycc_convert 13</td>
</tr>
<tr>
<td>641088</td>
<td>633216</td>
<td>7872</td>
<td>1.23</td>
<td>forward_DCT 14</td>
</tr>
<tr>
<td>116556</td>
<td>113546</td>
<td>3010</td>
<td>2.58</td>
<td>encode_mcu_DC_first 10</td>
</tr>
<tr>
<td>539206</td>
<td>536581</td>
<td>2625</td>
<td>0.49</td>
<td>emit_symbol 6</td>
</tr>
<tr>
<td>4480</td>
<td>2658</td>
<td>1822</td>
<td>40.67</td>
<td>_IO_file_xsgetn 2</td>
</tr>
<tr>
<td>1230919</td>
<td>1229140</td>
<td>1779</td>
<td>0.14</td>
<td>emit_bits 6</td>
</tr>
<tr>
<td>736464</td>
<td>734890</td>
<td>1574</td>
<td>0.21</td>
<td>compress_output 15</td>
</tr>
<tr>
<td>30464</td>
<td>28915</td>
<td>1549</td>
<td>5.08</td>
<td>encode_mcu_DC_refine 8</td>
</tr>
<tr>
<td>52256</td>
<td>50728</td>
<td>1528</td>
<td>2.92</td>
<td>__GI_memcpy 2</td>
</tr>
<tr>
<td>19560</td>
<td>18169</td>
<td>1391</td>
<td>7.11</td>
<td>pre_process_data 13</td>
</tr>
<tr>
<td>949296</td>
<td>948190</td>
<td>1106</td>
<td>0.12</td>
<td>jpeg_gen_optimal_table 14</td>
</tr>
<tr>
<td>669916</td>
<td>668950</td>
<td>966</td>
<td>0.14</td>
<td><em>GLOBAL</em> 0</td>
</tr>
<tr>
<td>6656</td>
<td>5767</td>
<td>889</td>
<td>13.36</td>
<td>fread 3</td>
</tr>
<tr>
<td>9728</td>
<td>8985</td>
<td>743</td>
<td>7.64</td>
<td>jpeg_write_scanlines 7</td>
</tr>
<tr>
<td>7937</td>
<td>7209</td>
<td>728</td>
<td>9.17</td>
<td>process_data_simple_main 7</td>
</tr>
<tr>
<td>9984</td>
<td>9264</td>
<td>720</td>
<td>7.21</td>
<td>__memcpy_sse3_back 3</td>
</tr>
<tr>
<td>4401</td>
<td>3780</td>
<td>621</td>
<td>14.11</td>
<td>start_pass_phuff 9</td>
</tr>
<tr>
<td>8722</td>
<td>8196</td>
<td>526</td>
<td>6.03</td>
<td>rgb_ycc_start 6</td>
</tr>
<tr>
<td>11904</td>
<td>11390</td>
<td>514</td>
<td>4.32</td>
<td>sep_downsample 12</td>
</tr>
<tr>
<td>8106</td>
<td>7672</td>
<td>434</td>
<td>5.35</td>
<td>__GI_memcpyp 3</td>
</tr>
<tr>
<td>2886</td>
<td>2500</td>
<td>386</td>
<td>13.37</td>
<td>main 13</td>
</tr>
<tr>
<td>274176</td>
<td>273804</td>
<td>372</td>
<td>0.14</td>
<td>h2v2_downsample 10</td>
</tr>
<tr>
<td>4096</td>
<td>3740</td>
<td>356</td>
<td>8.69</td>
<td>get_raw_row 5</td>
</tr>
<tr>
<td>8665</td>
<td>8420</td>
<td>245</td>
<td>2.83</td>
<td>jpeg_make_c_derived_tbl 14</td>
</tr>
<tr>
<td>17744</td>
<td>17505</td>
<td>239</td>
<td>1.35</td>
<td>access_virt_barray 9</td>
</tr>
</tbody>
</table>

Table 4.5. Cache simulation function results.

From Table 4.5 we can see that three functions, *encode_mcu_AC_refine*, *encode_mcu_AC_first*, and *rgb_ycc_convert*, cause a large number of cache misses and they access 13 to 15 program variables. This information can be seen in lines 3, 5, and 6 of Table 4.5. We can also see that accesses to *heap* elements, line 2, are responsible for nearly 80,000 misses and 3 million accesses. Accesses to *stack* elements, line 4, are considerably fewer with about 20,000 misses even though there are more than 10 million accesses to the stack area. Our cache simulator can be configured to provide statistics for selected execution intervals (for example for every *n* executed instructions or *m* cache misses). This type of information can reveal the dynamics of accesses to different address spaces and program variables during the program execution.

Figure 4.5 and Figure 4.6 show the number of accesses and misses during a program’s execution. The *X* axis is the time slice, i.e. the interval where data was accumulated (here
for every 300,000 load, store or modify accesses). The Y axis is the percent of the 300,000 accesses to stack, heap of global address spaces. The figure allows us to observe how the accesses to these different address regions change over the life of a program execution. In this example it can also be observed that most of the accesses to stack are hits (very few misses), while most of the access to heap addresses are misses. One can then explore the causes of these accesses and misses.

Figure 4.5. Jpeg: load, store, or data modifies for every 100k references.

Figure 4.6. Jpeg: load, store, or data modify misses for every 100k references.

The simulator’s data is also capable of tracking the percent of total accesses and misses during the execution intervals to individual program variables or structures. We can
observe the application’s L1 cache miss pattern in Figure 4.6. The majority of misses are *heap* misses despite the large large number of *stack* references observable in Figure 4.5. The simulator’s ability to track individual structures allows us to further investigate the behavior of individual structure objects by expanding the simulation analysis into the *heap* segment (see Figure 4.7 and Figure 4.8).

![Figure 4.7](image1)  
**Figure 4.7.** Jpeg: heap data references for every 100k references.

![Figure 4.8](image2)  
**Figure 4.8.** Jpeg: heap data misses for every 100k references.

We can observe that the majority of misses occur from approximately 240,000 – 270,000 executed data instructions. Also during the application’s phase from 350,000 –
910,000 instructions majority of misses are misses caused by a single block (LARGE_BLOCK-2). If we want to investigate the data structure cache layout (driven by its memory mapping) we can create a data structure layout plot for the duration of the application’s execution where each plot represent the CPU’s cache state for every $X$ instructions. For example, assume we are interested to see the cache state at 240,000 instructions. The reference plot (Figure 4.7) tells us that majority of references are due to MEDIUM_BLOCK-1, but we observe that the majority of misses are due to LARGE_BLOCK-2. We can examine the plot for the snapshot that corresponds to 240,000 accumulated data references. Figure 4.9 shows the CPU’s L1 cache state at 240,000 executed data references. We can reason about the cause of the incurred misses of object LARGE_BLOCK-2 by observing its access and miss pattern from Figure 4.9. The sea-saw like pattern indicates that the object is accessing every other set. This could indicate that a progression through memory is taking place (likely new data-address) and thus incurring misses. The majority of references from object MEDIUM_BLOCK-1 are visible as the light green spike occurring on set 33. This indicates that these accesses are to data with high temporal locality which means they are reused often and thus incur virtually no misses. Note that we can generate a cache state plot for every snapshot and examine the cache state changes in detail.

![Figure 4.9.](cpu-l1-cache-state.png)
Similarly, for every function displayed in Table 4.5, the simulator keeps track of individual program elements (variables or structures) as well as their access pattern for every cache set. This information is later displayed in detailed per cache set hit and miss tables. Such information can be used to identify cache sets that are heavily accessed (and cache sets that are underutilized).

For example assume that we are interested in function $encode_{mcu\_AC\_refine}$. The simulator generates a table (see Table 4.6) that shows the number of hits and misses for each cache set that the function touched during its execution. It also identifies the number of accessed variables or structures during that function call. In the first column we can see the detailed output per cache set for the function $encode_{mcu\_AC\_refine}$. This shows the total number of all accesses, hits, and misses for all identified variables and structures in that function. If we scroll through the table (see middle and right column) we can see other variables accessed associated with function $encode_{mcu\_AC\_refine}$. For illustration purposes we had to omit several cache sets and majority of variables. As an example; however, we can identify the variable $MEDIUM\_BLOCK$ which is in fact a dynamic object identified by scope level heap ($H$). The 3rd column shows data for some other variables, $Se$, $absvalues$, $jpeg\_natural\_order$, $block$, etc. Notice that every variable shown in our tables is annotated by its scope. For example, $MEDIUM\_BLOCK$ is identified by an $H$ indicating that this variable is allocated on the heap, $absvalues$ by $LS$ indicating that the variable is a local structure, or $jpeg\_natural\_order$ annotated with $GS$ indicating that the variable is a global structure.

### 4.3.4. Visualizing Data Layout

The data provided by the simulator is formatted so the data can be easily plotted. We have configured a set of Perl scripts to provide an easier interface to graphically visualize and analyze the simulator’s data. The scripts can generate a set of Gnuplot friendly output scripts or compile the data into Pdf documents. Thus, given the data in Table 4.6 we can generate graphical plots to visualize the behavior as depicted in Figure 4.10.

In Figure 4.10 the $X$ axis is the cache set number, and the $Y$ axis is the number of references (top graph), and misses (bottom graph) for every cache set. We can reason
that cache sets with heavy data contention and a relatively large number of references are likely to exhibit most misses, note that this behavior is usually offset by an increased cache associativity. In this example the simulation used a 32K bytes L1 cache with an 8-way associativity and 64 bytes per cache line, for a total of 64 cache sets. The graph in Figure 4.10 shows the function’s most misses are attributed to the dynamic object LARGE_BLOCK. We can also observe that there is a significant amount accesses to stack elements and other stack-based variables; however, their misses are minimal and thus negligible.

There can be any number of reasons for cache misses and the goal of the Gleipnir framework is to allow the user to observe memory access patterns and memory to cache mappings and observe cache effects in greater detail. The per cache set statistics (shown in Figure 4.10) can be used to develop a cache cost model to identify structures that cause the most evictions.

Table 4.6. Simulation results for function \textit{encode\_mcu\_AC\_refine} and its accessed variables and structures.
Figure 4.10. A cache-set layout visual representation of Jpeg function `encode_mcu_AC_refine`.

Graphs and plots can help the user to see a data structure’s memory to cache layout and observe the cache effects; however, to get an exact measure of the most conflicting variables the simulator can collect user-enabled cost matrices. A cost matrix is a simulator generated table that shows all cache line conflicts between various variables. Enabling this type of analysis degrades simulation performance due to tracking large number of data items that may reside on a cache line for every cache set. Table 4.7 shows an example cost matrix for function `encode_mcu_AC_refine`. Graphs in Figures 4.10, 4.7, and 4.8 suggest that the LARGE_BLOCK object is incurring most misses. However; to account for items that are evicting the object and thus are responsible for the misses we can generate its cost matrix. For example in Table 4.7 we can see that for function `encode_mcu_AC_refine`’s accesses to LARGE_BLOCK most evictions and thus misses (9276) are caused by the same function’s accesses to LARGE_BLOCK. This means that due to its size, LARGE_BLOCK
is incurring capacity misses. Also a significant amount of evictions (1460) is incurred due to accesses to function `encode_mcu_DC_refine`, and 1457 of those are incurred due to accesses to `LARGE_BLOCK` object. At a glance, this implies the this benchmark’s cache access pattern consists of frequent accesses to very large objects and their misses are largely due to capacity misses.

4.4. Multi-core and multi-process analysis

In chapter 1 we have introduced the concept of cache indexing as it pertains to private and shared level caches. Private level caches (L1 cache) is virtually indexed, this means that a data’s virtual address is sufficient to place an element into the cache. However, last level caches (LLC) or shared caches (a fundamental component of multi-core chips), are physically indexed. This implies that cache simulators must use a physical address to index last level caches in order to achieve a reasonable cache simulation accuracy. In turn, this implies that trace-generation tools are required to generate physical addresses. Gleipnir’s traces are annotated with relevant information to enable multi-core and multi-process analyses; however, this is not yet possible due to the limitations of the cache simulator used - DineroIV simulate single core caches only.

The physical data address is assigned by the operating system when a virtual page is mapped to a physical page and thus unavailable to most instrumentation frameworks and tools. However, Gleipnir includes a mechanism that takes advantage of operating system kernel’s `maps` and `pagemap` files to extract the physical addresses.⁴

Physical address tracking can be enabled in Gleipnir by setting the `-map-phys=yes` flag. Enabling that flag tells Gleipnir to track allocated virtual and physical pages. The trace in Listing 4.8 shows an example trace of the code in Listing 4.1 when physical address tracking is enabled. The trace looks similar to the basic Gleipnir trace line except that a physical address is added in additional to the virtual address.

⁴We are not aware of any other instrumentation frameworks or plug-in tracing tools that are capable of mapping physical and virtual addresses.
Table 4.7. Cache simulation’s cost-matrix for function encode_mcu_AC_refine.
Furthermore, to achieve a more accurate shared cache simulation we have to account for cycle information so that accesses from multiple processes or threads can be interleaved. The current cache simulator is trace-driven cycle-unaware but a future cache simulation model can rely on instruction cycle information provided by the tool. Fetching instruction types is trivial using the Valgrind framework. Annotating Gleipnir’s traces with instruction cycle models and physical addresses we can simulate a shared cache and multi-process environment.

### 4.5. Future Work

Our goal in developing Gleipnir is to make program analyses and profiling techniques easy to use by the average programmer, and by combining Gleipnir traces and cache simulation we aim to expose the underlying data conflicts that may occur between various program data structures. The ability to account for common cache performance bottlenecks is not an easy problem, and it is even more difficult to provide an efficient method for data placement analyses and optimizations. Future work aims to address the current shortcomings with the

<table>
<thead>
<tr>
<th>START PID 4481</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 7ff0004b0 86000000002c0eef 8 1 S main</td>
<td>2</td>
</tr>
<tr>
<td>S 7ff0004bc 86000000002c0efb 4 1 S main LV main_local</td>
<td>3</td>
</tr>
<tr>
<td>L 7ff0004bc 86000000002c0efb 4 1 S main LV main_local</td>
<td>4</td>
</tr>
<tr>
<td>S 7ff000470 86000000002c0eaf 8 1 S main</td>
<td>5</td>
</tr>
<tr>
<td>S 7ff000468 86000000002c0ea7 8 1 S functionA</td>
<td>6</td>
</tr>
<tr>
<td>S 7ff000454 86000000002c0e93 4 1 S functionA LV param1</td>
<td>7</td>
</tr>
<tr>
<td>S 7ff000460 86000000002c0e9f 4 1 S functionA</td>
<td>8</td>
</tr>
<tr>
<td>S 7ff000464 86000000002c0eaf4 4 1 S functionA LV fnc_local</td>
<td>9</td>
</tr>
<tr>
<td>S 00060109 86000000002d6b4f 4 1 G functionA GV my_gl</td>
<td>10</td>
</tr>
<tr>
<td>S 0006010a 86000000002d6b5f 4 1 G functionA GS my_glArray [2]</td>
<td>11</td>
</tr>
<tr>
<td>L 7ff000468 86000000002c0eaf7 8 1 S functionA</td>
<td>12</td>
</tr>
<tr>
<td>L 7ff000470 86000000002c0eaf 8 1 S functionA</td>
<td>13</td>
</tr>
<tr>
<td>S 00060109 86000000002d6b4f 4 1 G main GV my_gl</td>
<td>14</td>
</tr>
<tr>
<td>S 0006010a 86000000002d6b5f 4 1 G main GS my_glArray [2]</td>
<td>15</td>
</tr>
<tr>
<td>S 00060106 86000000002d6b23 4 1 G main GS GStruct [1]. A</td>
<td>16</td>
</tr>
<tr>
<td>S 0006010a 86000000002d6b07 4 1 G main GS GStruct [0]. B [3]</td>
<td>17</td>
</tr>
<tr>
<td>S 7ff000480 86000000002c0efb 8 1 S main LS _zzq_args [0]</td>
<td>18</td>
</tr>
<tr>
<td>S 7ff000488 86000000002c0e7 8 1 S main LS _zzq_args [1]</td>
<td>19</td>
</tr>
<tr>
<td>S 7ff000490 86000000002c0ecf 8 1 S main LS _zzq_args [2]</td>
<td>20</td>
</tr>
<tr>
<td>S 7ff000498 86000000002c0ed7 8 1 S main LS _zzq_args [3]</td>
<td>21</td>
</tr>
<tr>
<td>S 7ff0004a0 86000000002c0edf 8 1 S main LS _zzq_args [4]</td>
<td>22</td>
</tr>
<tr>
<td>S 7ff0004a8 86000000002c0e7 8 1 S main LS _zzq_args [5]</td>
<td>23</td>
</tr>
<tr>
<td>S 7ff0004c7 86000000002c0eab 4 1 S main</td>
<td>24</td>
</tr>
<tr>
<td>L 7ff00047c 86000000002c0eab 4 1 S main</td>
<td>25</td>
</tr>
</tbody>
</table>

**Table 4.8.** Gleipnir trace with physical address tracing enabled.
4.5.1. Identifying Logical Structures

Currently Gleipnir will track single structure allocations and automatically group them by the parsed or user supplied structure name. For example multiple allocations to structure XYZ are enumerated as $XYZ_{1,2,3...etc}$. We use the term logical structures to identify dynamic abstract data types such as linked lists or trees. The goal is to identify logical structures using the framework and group them in specific categories. This would enable the cache simulator to group cache conflicts with unique logical structures and thus give the programmer the ability to evaluate any potential benefits for utilizing different structures (e.g. linked lists vs. trees).

We can achieve this by tracking pointer values used to connect when an allocation to the next block occurs. By utilizing the user input to identify structure pointer types for allocated structures we can track and combine multiple structure allocations into unique logical structures. For example several allocations to structure XYZ can be linked into a logical structure $XYZ_1$. Enabling the \texttt{--track-values=yes} flag directs Gleipnir to identify and annotate the trace instructions with data values at address locations. Listing 4.10 is an example of a generated trace with annotated values for the source code in Listing 4.9.

4.5.2. Trace-driven Data Structure Transformations

Another potential use for Gleipnir as demonstrated in [24] is to utilize the traces for semi-automatic trace-driven data structure transformations. Observing various structure transformations is important to reason about potential cache effects different structure layouts may have. By utilizing Gleipnir’s traces we implemented a trace-driven cache simulation module that semi-automatically transforms data structures and analyzes the cache effects. It is semi-automatic because it relies on the user to supply transformations rules and automatic because the module can identify the transformation rules and automatically apply them on the trace.
4.6. Conclusions

In this chapter we described a tracing and profiling tool called Gleipnir. Gleipnir is built using a well-known dynamic binary instrumentation tool, Valgrind. We elaborated on how Gleipnir works in combination with our cache simulator. We have illustrated how Gleipnir traces programs and how to interpret the output generated by Gleipnir. Gleipnir provides very detailed information on every memory access including the name of the variable accessed, if the variable is a stack, global or dynamically allocated element, the function that contained the access and the thread that executed the access. For dynamic objects (or heap objects), Gleipnir tracks multiple allocations with the same structure name, and the element within the structure that was accessed.

We demonstrated the ability to produce detailed application cache analysis as well
as cost matrices (see Table 4.4, 4.5, 4.6, and 4.7) identifying the source of cache misses. We have shown how to track which variables are evicting a particular variable. Such information is valuable when exploring different data layouts, code and data refactoring techniques (such as tiling, fusing etc.).
CHAPTER 5

EQUIVALENCE CLASS BASED MEMORY ALLOCATION

In Chapter 3 we have described the traditional metrics when evaluating allocators, fragmentation and locality. We have explored the effects of several allocator policies on fragmentation and explained why bad fragmentation will ultimately result in performance degradation. However, the complexity of software systems and the complexity of memory hierarchies have significantly increased since the early days of dynamic memory allocation. Therefore, we must also explore other metrics that may ultimately affect overall system performance. Particularly we must look into block placement strategies when evaluating memory allocator behavior in addition to fragmentation and locality since placement is critical to minimizing cache conflicts. In this Chapter we use the Gleipnir framework to measure and analyze the performance of various allocators. In addition we propose an equivalence allocator, eq_malloc, a user-driven cache conscious allocator whose aim is to utilize equivalence classes to partition the memory based on object cache placement.

5.1. Introduction

Consider what happens during an application’s life. Memory objects (dynamic blocks of memory) of various sizes are allocated from different memory regions. Because cache mapping mechanism impacts overall cache performance, objects that are allocated from regions that map to same cache blocks will deteriorate cache performance. The reason for that is that blocks that occupy regions of memory which map into the same cache set will continuously evict each other from the cache. Of course there are hardware cache mechanisms to alleviate this behavior [29]. Similarly smaller short-lived objects that exhibit very short reuse distance are likely to evict larger long-lived data. An object’s reuse distance is the number of other distinct memory references between two accesses to the same object. In this Chapter we will describe a new allocator mechanism that utilizes equivalence classes to partition the memory into regions of cache conscious blocks, that is to say memory regions that are specifically chosen to fit into user-specific cache locations. In order to explain the
allocators mechanisms we need to define equivalence relations and equivalence classes.

An relation $R$ on a set $A$ is an equivalence relation ($\sim$) if and only if the following properties are true:

1. $R$ is reflexive. For all $x \in S$, $x \sim x$.
2. $R$ is symmetric. For all $x,y \in S$, $x \sim y$ implies $y \sim x$.
3. $R$ is transitive. For all $x,y,z \in S$, $x \sim y$ and $y \sim z$ implies $x \sim z$.

For example the congruent modulo $m$ ($\text{mod } m$) relation on a set of integers $<a, b>$ $|a \equiv b \text{ (mod m)}$, where $m$ is a positive integer greater than 1, is an equivalence relation. An equivalence class is derived from elements of a set $X$ and an equivalence relation $R$ on that set $X$. Thus an equivalence class of an element $a$ in $X$ is the subset of all elements in $X$ which are related to $a$: $[a] = \{x \in X | xRa\}$. Note that equivalence classes are disjoint. It should also be noted that cache memory addressing is based on modulo arithmetic and thus an applications addresses form equivalence classes – all addresses that map to a cache line (or set) is an equivalence class. Therefore, we can now partition the memory space $M = \{i\}, 0 \leq i \leq 2^{64}$ with equivalence relations $R_i$ where any partition of $M$ is an equivalence class $C_i$ of partitions $M$ by an attribute $A_i$. Or we can partition the memory space into sets of equivalence classes $C_i$ with a relation $R$ on a set of address spaces. We can further partition the memory space by any number of attributes $A_i$ that may result in recursive and simultaneous partitions.

5.1.1. Using Equivalence Classes in Memory Allocators

Let an instruction window ($IW$) be a number of instructions within which we can define conflicts. Thus we can set the window arbitrarily or restrict it to a set of rules. Let $V = (v_1, v_2, ..., v_j)$ be a set of unique variables accessed in the window $IW$. We can define the set of cache lines available as $S$. Now we will divide the set of variables $V$ into $S$ equivalence classes $C = (C_0, C_1, ..., C_j, C_{S-1})$. Cache indexing operates using modulo arithmetic, thus cache sets form an equivalence class with a ($\text{mod } m$) relation $R$. The goal is to minimize cache conflicts by using the reuse distance\(^1\), number of other distinct memory references

\(^1\)We can also use other physical parameters that define a structure or variable.
between two accesses to the same memory location, of variables with an equivalence class. Consider a set or equivalence class \( C_j = (v^j_i) \) which is the set of variables of \( V \) mapping to the cache set \( C_j \). We can minimize the conflicts by isolating those variables with short reuse distances. The goal is to keep the variables with short reuse distances in those sets. We can follow these steps to achieve this:

1. Sort variables in the set \( V \) based on their reuse distances.
2. Assign the first \( S \) variables with small reuse distances to the different cache sets.
3. Map remaining variables of \( V \) to the sets.

The second mapping may use a different heuristic.

Consider the two windows \( IW_k \) and \( IW_{k+1} \), consider \( V_k \) and \( V_{k+1} \) be the set of variables accessed in the consecutive instruction windows. If a variable \( v_i \) appears in both sets \( V_k \) and \( V_{k+1} \) then we need to minimize the probability of \( v_i \) being evicted between those windows. Thus by identifying all such variables accessed in consecutive windows, we can minimize the number of other variables in equivalent classes containing these variables. If a set \( C_j \) contains a variable \( v_j \) that is accessed in two consecutive windows \( IW_k \) and \( IW_{k+1} \) we will avoid placing other variables from the set of variables \( V_k \) into \( C_j \).

We can define \( IW \) using various properties but we will concentrate on physical parameters such as cache miss statistics. In conclusion the goal is to minimize cache conflicts by maximizing variable’s live-time. This in turn is achieved by limiting the variables mapping into the same cache sets using an allocator that can request blocks that map to user desired sets. Other \( IW \) properties to consider may be, average reuse distance over all variables or \textit{Equi-live} property proposed in [10].

5.1.2. Allocation Policies

Based on previous definitions we can describe our allocation policy as \textit{cache-conscious priority placement} policy. Unlike the general allocators described in Chapter 3 we can categorize our allocation policy as a user-driven \textit{critical data structure} placement policy. \textit{Cache-conscious data placement} is a software technique to improve data cache performance by
relocating variables in the virtual memory space. The goals are to improve overall cache performance through improved locality and reduced cache conflicts. *Cache-conscious data placement* was previously discussed in [12], [9], and [41]. Calder et al. described an automated algorithm approach for cache-conscious data placement in [9]. Their tool uses a profiler to establish an access and conflict pattern between allocated nodes to synthesize an automatic allocator and guide compiler based block placement. Chilimbi et al. describe several techniques for *cache-conscious* data placement in [12]. Their work discusses several programmer techniques for structure organization and couple semi-automatic and automatic tools to aid in data-placement.

A *critical data structure* is any user identified data structure that is observed to have detrimental behavior on the CPU’s cache. In other words a critical data structure could either be a list of head nodes of a linked-list, or a root node of a tree-like structure. We say user defined because we rely on the user to place the appropriate allocation predicates when allocating memory. This process may be automated at a later stage, however, at the moment any automation is beyond the scope of this study and not included in this implementation. This allocation necessitates the use of tools to automate the analysis and give feedback to the user so that the user may use the analysis information for appropriate allocation strategy. In other words, similar to custom allocation the user input is critical when implementing an application that will utilize this allocator. Note that optimizing allocators based on application profiles is a desired software engineering feat [49].

5.2. Implementation

Let us rehash the definition of an equivalence class with respect to a processor’s cache. An equivalence class is derived as a relation on elements in a set. In our implementation our equivalence class is a cache set. And for most practical purposes our allocator can accommodate S equivalence classes; where \( S = \frac{\text{cache size}}{\text{bytes per block} \times \text{ways}} \) or \( C = \frac{2^k}{2^b \times 2^w} \); where \( k = \log_2(\text{cache size}) \) bits, \( b = \log_2(\text{block size}) \) bits, and \( w = \log_2(\text{ways}) \) bits, that stores a cache-line size bytes from memory. For all practical purposes this policy revolves around the utilization of *offset, index, and tag* bits as described in section 1.2. We described in Chapter
1 how a memory location is indexed using a cache indexing mechanism. Cache indexing defines a modulo arithmetic to place data items from higher level memory hierarchies into cache sets. This is important to understand because effectively every memory location that is aligned on a cache line (typically 64 bytes) boundary will fall into one of the cache sets.

For the rest of this section we will assume a cache design of 32K bytes, with 64 bytes per block and an 8-way associativity. This cache configuration is a de facto standard for L1 CPU caches in today's desktop processors. Consider what happens when an application requests a block of memory that is aligned on a page size boundary. As shown in Chapter 1 Figure 1.3, the first address in the page will be indexed into the first set of the cache. The next 64 bytes will be indexed into the next set. This is true for every address aligned to 64 bytes within that page. When the next page is loaded the next address will wrap around the cache and indexed to set 0. Therefore a single cache way in the cache is exactly 4096 bytes in size (64 sets × 64 bytes per block). The entire cache is 32K bytes in size, this means that a cache can hold 8 pages of memory. Figure 5.1 shows this concept. We designed the equivalence class based allocator around this implementation constraint.

![Figure 5.1. Virtual pages into a 32K cache concept.](image)

**Basic structures.** Every allocator will implement a basic control mechanism that dictates overall allocator behavior. These are known as internal allocator structures. Because an allocator is essentially a module compiled into an application, these structures can either
reside on the application’s global data segment or they can be specially allocated when the
allocator is initialized. Our eqalloc consists of three basic control structures: main arena,
equivalence blocks, and regular blocks. Equivalence blocks are further split into an 8Kbyte
control section and a 32Kbyte data section. There is a reason for these sizes that will be
discussed shortly. Figure 5.2 shows a diagram of the basic eqalloc structures. The main
arena is a 4Kbyte (page aligned) structure that consists of several fields. The first field is an
eight byte storage field that indicates the number of allocated arenas (we use the term arena
to indicate a region of memory for equivalence allocations). The second field is an 8 byte
pointer to a list of allocated large blocks. Any block that is larger than 32K bytes is defined
as a large block and will not be given special treatment (these blocks are automatically
allocated from the regular list). The reason for this is obvious, blocks larger than 32K bytes
occupy more than the cache size and cannot be satisfied using equivalence allocations. The
next field is an array of 508 equivalence arena pointers. The size of this array is simply the
remainder of unused area in the main arena. Finally the last two fields are pointers to next
and previous main arenas. When an application requests more than the available number of
equivalence arenas it will need to allocate another main arena to manage them. The main
arenas are connected through a doubly linked-list. This makes traversing main arenas faster.

![Figure 5.2. Overview diagram of Eqalloc.](image)
5.3. Equivalence Allocator Mechanics

Controls. An equivalence arena as shown in Figure 5.2 consist of a control block (header) and the data section. We deliberately kept the header away from the actual blocks to increase compaction of smaller blocks. This makes sense for blocks that are smaller; however, for blocks that are larger this is a drawback because several header fields will go unused. The other reasoning is that using a carefully aligned header to partition memory into equivalence classes is straightforward. Consider that an entire page can fit into a 64 set, 64 bytes per line cache. This means that exactly 64 pointers are necessary to point to the maximum number of available classes. An element in the control is a structure of three elements for a total of 16 bytes. The first field is the number of free blocks within that set, the second field is the size of the block in the respective set, and the third field is the pointer to the actual set in memory. At 16 bytes per control and 64 sets we need 1024 bytes to represent a 4096 byte page, that is a 25% overhead. We will discuss this in further detail in subsequent sections.

Block allocation. When a block is requested the call supplies two parameters to the allocation. A size parameter, and a set parameter. The aditional parameter is a descriptive parameter for the requested block. The allocation algorithm is straightforward to implement using these two parameters. The algorithm is described in Algorithm 1. Every allocation requests an arena to work in. The algorithm loops through all the controls in an arena which means that a worst case scenario will loop through a maximum of 8 controls per arena before either satisfying the request or requesting more memory. For every control there are two possibilities. The set already contains the sized block and the request is a simple return of the head pointer, or the set is not initialized and an init() action is performed. Initializing the set means that the size is defined and the control head pointer is placed at the beginning of the set minus the returned block.

Block freeing. A block free is different from the traditional allocators. Finding the set to which an address belong is relatively easy using bit arithmetic; however, finding

\footnote{It is also possible to use a threshold, such as a number of neighboring sets if the requested set is not available.}
Algorithm 1 Algorithm for finding an available set block

while (1) do
    get_main_arena();
    for $i = 0$ to $CONTROL\_PER\_ARENA$ do
        if ($control - > size == 0$) then
            init_set();
            $found = TRUE$; return
        end if
        if ($control - > size == reqsize AND ptr \neq NULL$) then
            request();
            $found = TRUE$; return
        end if
    end for
    if ($found == FALSE$) then
        get_new_arena();
    end if
end while

the appropriate arena is difficult. The difficulty is due to memory mapping equivalence arenas using `mmap()` system call which can allocate equivalence arenas at arbitrary memory addresses. Therefore, freeing objects from an arena requires that we first find the base address of the arena by traversing every allocated arena. In the worst case scenario this may be expensive, especially when the number of arenas is relatively large; however, a simple caching mechanisms may improve performance.

*Splitting and Coalescing.* The *equivalence class* allocator does not support splitting and coalescing in the traditional sense but there are a few key characteristics that we must elaborate on. When a new equivalence arena is created, the control is initialized to point to every 64 byte block in the arena. This will partition the entire arena to every cache set. An undefined set is simply a set which was not allocated, the *size* field indicates if a block is defined, occupied, or free. A request to an undefined set will always result in splitting a block, unless the block is 64 or more bytes. This is because every set in the large 32K block is initially undefined, i.e. it is not claimed by any object. When a block is allocated
every set is always split into subblocks of smaller blocks. The largest supported single block is 32K bytes, thus splitting is always performed when a set is claimed because of the control structure. Coalescing is implemented as redefining control sets of free blocks. Thus we support coalescing on various block granularities without additional overhead. Smallest blocks, i.e. blocks smaller than a cache line are never coalesced. In previous studies it was pointed out that smaller blocks are short lived but reused often. Therefore, when a small block claims a set it will remain in that set for the duration of the application. Similarly a medium sized block, a block that is larger than a cache line but smaller than a page ([128bytes, 4096 bytes]), are also not subject to a complete redefinition. Large blocks, blocks that are larger than a page are always fully freed, meaning that their sets will be reinitialized so that they can be reused by another large request. Our philosophy here seems counterintuitive because reinitializing large blocks will be costly, but we believe that requests to such large blocks are rare, and even when they do occur they are long lived so we do not need worry about it happening too often.

5.4. Comparison of Allocators

In this section we will compare three different memory allocators with respect to their object placement and cache performance. In Chapter 3 we have described that fragmentation and locality were always considered to be the standard metrics for evaluating allocators. Here, we explore how allocator decisions affect cache performance. To evaluate this we will use the previously described Gleipnir framework (see Chapter 4). The Gleipnir framework is unique in that it allows us to trace every allocated block and record its cache placement and overall cache behavior. This allows us to relate object placement and other program structures (eg. stack, global, and more importantly, other dynamically allocated objects) to cache performance. To evaluate how well the allocator behaves we will run three different allocators against a series of stress-benchmarks as well as several real benchmark applications. The goal of this study is to explore the impact of cache-conscious memory allocations.

3This behavior is different if the block is larger than a cache line.
5.4.1. Summary of Allocators

**Standard Library’s C Malloc.** As our control allocator we will use the standard C library’s `malloc` (*dlmalloc*) allocator because in our experience the majority of benchmark use *dlmalloc*. This is a general purpose allocator designed to satisfy a variety of allocation patterns without incurring extra overhead. The implementation is derived from the Doug Lea’s segregated list implementation [34]. It uses boundary tags and segregates blocks into lists of same sized objects. Blocks smaller than a threshold size are managed using `sbrk()` system calls and blocks larger than the threshold are accommodated using the `mmap()` system call. Figure 5.3 illustrates the general block management. Free chunks are doubly linked and kept in same-sized lists. The benefit of this allocator is the speed of allocation and deletion. Allocation can be accommodated with a few instructions in constant time using an index into an array of same-size chunks to lookup the pointer to the next available chunk. Similarly block splitting and coalescing is fast using boundary tags. The drawback is that due to alignment, minimum required size, and block overhead the allocator is unable to utilize a full cache line for smaller blocks, thus large portions of each cache line will be unused, or cause internal fragmentation.

![Figure 5.3. DLmalloc general block management.](image)

**Pool allocator.** The pool allocator (*poolmalloc*) used in this study is a custom allocator designed for fast allocation. Pool allocation was proposed before in [4]. The allocator’s strategy is to reserve memory pools for same sized objects. This constraint may be restricted to allocate specific data objects from specific memory pools. In other words objects that are
of the same size but are likely to behave differently can be allocated from unique memory pools. In this implementation; however, the allocator will only accept the size parameter as the primary allocation argument. This also means that pool allocation will not differentiate objects based on their access pattern. The goal of pool allocators is to increase locality of same sized objects because several studies have shown that objects of similar sizes exhibit similar life-times. Figure 5.4 shows the pool allocator’s general block management. Look up times are constant because the input parameter is enough to determine the pool list for the pool allocator. A pool list is a structure that contains the size of the pool, number of free nodes, a pointer to a free block and a pointer to a wilderness chunk. The wilderness chunk is the last chunk in the pool, and it is the only chunk that can be used to allocate another object from that pool. We use the free list for recently freed objects from the same pool. By far the biggest drawback is on object deallocation. Object freeing is costly because objects do not carry a size header. Thus to find the appropriate pool the pool allocator must traverse every pool in the list and compare the return address with the appropriate pool. An optimization may implement a caching mechanisms for recently freed objects.

Equivalence class based cache-conscious allocator. The equivalence class based allocator (eqmalloc) is described in greater detail in Section 5.2. This allocation strategy relies on the user’s allocation experience to place the blocks in appropriate equivalence classes. The goal of this allocation strategy is to exploit the application’s observed cache behavior in order to optimize overall cache behavior.
5.4.2. Evaluation

To evaluate allocator’s block placement and its effect on the CPU’s cache we will compare different allocator’s block allocation effects on several stress benchmarks that use some common data structures, and some real benchmark applications. The stress-benchmarks include a binary search, a linked-list search, and a multi-list search, and they exhibit the ramp, plateau, and peak memory behaviors defined in Chapter 3. The goal of stress-benchmarks is to evaluate the block placement and the resulting cache behavior when nodes are allocated using different memory allocators and subsequently traversed during run-time. Because the boundary tagged allocator, dlmalloc, incurs memory overhead for every allocated block, we will evaluate every benchmark with different payload sizes. Depending on the benchmark every node will have a base size due to data elements necessary to construct abstract structures. For example the binary tree’s node consists of a long key variable, and two void pointers that point to the left and right nodes of the tree. This implies that every node in that benchmark will have a 24 byte base size. The remaining bytes will be adjusted to pad the remaining cache lines. The smallest payload size is 8 bytes which simulates a pointer field to additional data. As expected node size augmentation will yield varying allocation results. The intent is to focus the evaluation on block placement rather than allocator implementation.

Stress-benchmarks:

Binary search. The binary search benchmark performs a binary search algorithm on an input set of random integer values. The goal of this benchmark is to evaluate the effects of an allocator’s object allocation when traversing a binary-tree structure. The benchmark executes in three phases:

1. In the first phase the benchmark reads a randomly generated input set and generates a dynamically allocated binary tree structure. Note that the generated structure is not a balanced binary tree, thus depending on the input set the tree may be left skewed or right skewed.

2. In the second phase the algorithm generates a search sequence of integer values.

3. In the third phase the algorithm performs a search on this data structure.
Figure 5.5 shows the three data structure and the number of node visits when visiting leaf nodes. As can be observed the root node will incur most references when accessing the leaf nodes. This type of access behavior can be exploited to improve the benchmark’s cache performance by allocating nodes closer to the root in separate equivalence classes. Figure 5.6 shows the benchmark’s stack, global, and heap access pattern when dl_allocator is used. The graph indicates that the majority of accesses are to stack and that the benchmark’s access pattern is uniform. Due to the large number of stack access we can expect that a proportional number of misses will be due to stack references; however, Figure 5.7 shows that virtually all L1 cache misses are due to heap data. We can further investigate the heap misses by categorizing the structure’s individual nodes based on their allocation pattern. In this example, the first 1000 allocated nodes will be denoted as NODE_1k nodes. Similarly, the next 1000 allocated blocks will be traced as NODE_2k nodes, etc.

![Figure 5.5. A binary search stress-test benchmark.](image)

Figure 5.8 shows the number of node misses for each node category for every 100,000 references. We can observe that for most of benchmark’s run-time the majority of misses are attributed to the first 1000 allocated nodes, NODE_1k (dark blue). Approximately 30% of misses for every 100,000 references is attributed to NODE_2k (yellow) and NODE_5k (dark green), and a negligible amount is attributed to nodes NODE_8k (light green) and NODE_10k (light blue).

Figure 5.9 shows the dynamic object’s per cache set mapping for the entire benchmark’s execution. We can observe that overall cache misses per cache set confirm the observation
from Figure 5.8, meaning that lower level nodes (those closer to the root of the tree) incur more misses. We can exploit this allocation pattern to reduce the cache contention between lower level tree nodes by allocating higher level nodes to higher level cache sets. This allocation pattern exploits the observation that lower level tree nodes are accessed more frequently relative to higher level nodes. Our allocation algorithm will keep an allocation counter that increments at every allocation up to the maximum number of sets (64). When it reaches the last set it will complete an allocation cycle and restart the counter at an increment of the previous cycles’ base counter. Therefore, for the first 1,000 allocations the algorithm will utilize the entire cache. The remaining 9,000 allocations will be placed on sets.
Figure 5.8. Binary search: heap node misses using \texttt{dlmalloc}.

32 – 63. Figure 5.10 shows the allocation patterns that we would expect for the first 2,000 nodes, meaning that only the first 1,000 blocks are expected to map to lower level cache sets.

Figure 5.9. Binary search: binary tree’s \texttt{NODE\_1k}, \texttt{NODE\_2k}, \texttt{NODE\_5k} cache mapping using \texttt{dlmalloc}.

Figure 5.11 shows the resulting cache set mapping when using the equivalence allocator and the previously described allocation strategy. We can observe that the first 1,000 nodes map to the entire cache while the remaining nodes occupy only the upper half of the cache. Notice the reduced cache misses on lower level sets. The first cache set only exhibits a single cache miss compared to 2000 misses in Figure 5.9. Similarly lower level sets have rarely over 1,000
Figure 5.10. Binary search: equivalence allocator’s allocation strategy.

misses. The pool allocators allocation pattern can be observed in Figure 5.12. Interestingly, the block misses are also fairly low for the first 1,000 nodes when compared to \textit{dlmalloc}.

Table 5.1, 5.2, 5.3 show the cache miss results using the different allocators. Overall, the equivalence allocator performed the best by reducing 59,866 (44\%) misses. The pool allocator performed a little worse than the equivalence allocator, reducing 56,572 (41\%) misses. The drastic reduction in cache misses using the pool allocator is somewhat surprising because no cache-conscious block placement took place. However, the reduction in misses using the pool allocator can be explained as a result of the allocator’s cache line utilization. Considering that the allocated blocks are exactly 64 bytes, thus fit in an entire cache line, and the blocks are allocated from page boundary aligned memory pools, the allocator performs a perfect block alignment. The cache miss difference between the pool allocator and the equivalence allocator
can be attributed to the cache-conscious block placement of the equivalence allocator. Also, in Figure 5.12 we can observe that there is a small spike in misses on cache set zero using the pool allocator. This is likely due to the pool allocator’s page alignment mapping the first block (block with the largest number of references) to cache set zero. Table 5.4 shows the overall cache hit and miss differences for the application’s stack, global, and heap segments.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>15,309,708</td>
<td>15,309,708</td>
<td>9,640,933</td>
<td>5,668,775</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>137,811</td>
<td>137,811</td>
<td>137,773</td>
<td>38</td>
</tr>
</tbody>
</table>

Table 5.1. Binary search stress-test benchmark’s simulation results for *dlmalloc*.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>15,309,708</td>
<td>15,309,708</td>
<td>9,640,933</td>
<td>5,668,775</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>81,239</td>
<td>81,239</td>
<td>80,948</td>
<td>291</td>
</tr>
</tbody>
</table>

Table 5.2. Binary search stress-test benchmark’s simulation results for *poolmalloc*. 
Table 5.3. Binary search stress-test benchmark’s simulation results for *eqmalloc*.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>15,309,708</td>
<td>15,309,708</td>
<td>9,640,933</td>
<td>5,668,775</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>77,945</td>
<td>77,945</td>
<td>77,911</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 5.4. Binary search: stack, global, and heap segment overall L1 cache misses.

<table>
<thead>
<tr>
<th>Segments</th>
<th>dlmalloc l1-dcache misses</th>
<th>poolmalloc l1-dcache misses</th>
<th>eqmalloc l1-dcache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accesses</td>
<td>Hits</td>
<td>Misses</td>
</tr>
<tr>
<td>Stack</td>
<td>10257 K</td>
<td>10256 K</td>
<td>31</td>
</tr>
<tr>
<td>Global</td>
<td>3055 K</td>
<td>3053 K</td>
<td>1505</td>
</tr>
<tr>
<td>Heap</td>
<td>1997 K</td>
<td>1861 K</td>
<td>136275</td>
</tr>
</tbody>
</table>

*Linked-list search.* The linked-list stress benchmark performs a search on a linked-list data structure. The data structure consists of a key (integer value), a next pointer, and a data structure to occupy the remaining cache bytes. Figure 5.13 shows the linked-list search concept. The benchmark stresses an object’s allocation and access pattern because the benchmark exhibits a non-uniform cache access behavior. That is, nodes that are closer to the head node will be accessed more often than nodes farther way in the list. The worst case scenario is a linked-list traversal where the head node will be accessed \( n \) times, and every subsequent node will be accessed \( n - i \); where \( i = (1, \ldots , n - 1) \). Figure 5.14 shows the cache miss behavior of the linked-list search benchmark for every 300,000 references when dlmalloc allocator is used. Similar to the binary search benchmark we have categorically separated different nodes into 4 distinct categories: the first 500 nodes are denoted as \( \text{NODE}_{500} \), the second 500 nodes are \( \text{NODE}_{1000} \), the third 500 are \( \text{NODE}_{1500} \), and the last 500 nodes are \( \text{NODE}_{2000} \). This allows us to observe the miss behavior of the various nodes that comprise the linked-list.

In Figure 5.14 we can observe that two thirds of misses for every 300,000 references are attributed to the first 500 nodes (blue). The next highest misses are attributed to the second 500 nodes (dark green), and about the same number of misses are attributed to the last 1,000 nodes (light green and yellow). Similar to binary search benchmark we can attempt to exploit the node allocation and access pattern such that the cache contention
between lower level nodes and higher level nodes is reduced.

Figure 5.14 shows the overall stack, global, and heap cache set mappings for the linked-list search benchmark. Our allocation policy will exploit the linked-list’s traversal pattern. As we have observed in Figure 5.14 a large portion of misses are due to the first 500 nodes. Thus, we can exploit the access pattern by applying an allocation pattern described in the binary search stress benchmark. We will keep an allocation counter that increments a base counter forcing later object allocations to map to higher numbered cache sets. This will ensure that the contention on lower cache sets is reduced thus reducing overall cache misses.

Figure 5.16 shows the cache effects of the equivalence allocator using the allocation pattern. We can observe the cache miss reduction on lower level cache sets. Also note that cache misses even out around sets 32 and higher. This is due to the higher contention on these
sets between the first 500 nodes and the lower 1500 nodes. The spike in cache set misses on cache set 7 is initially surprising; however, notice that cache set 7-9 exhibit also higher stack and global misses. This could explain the spike in heap misses around the same sets. One could argue that cache misses should revert to gradual increase on sets 11 − 63; however, if we analyze the heap misses we can see why this is not the case, and the contention on these sets abruptly increases.

Figure 5.15. Linked-list: Stack, global, and heap cache mapping using dlmalloc.

Figure 5.17 shows that the second 500 nodes, NODE_1000, start their mapping at set 8. This explains the abrupt increase in cache misses on set 7 − 63. Increase in cache misses on set 7 can be explained by the cache mapping of heap, stack, and global data; however, the cache misses do not continue their gradual increase from lower numbered sets due to the NODE_1000 blocks that start their mapping to cache set 8 through 63. Compared to dlmalloc, the pool allocator shows little variation in terms of heap accesses. This is to be expected as the linked-list is allocated from a pool of memory. Moreover, the allocator does
not store any object header or footer information thus the cache lines will be fully utilized.

Figure 5.18 shows the benchmark’s stack, global, and heap segments using the pool allocator.

Figure 5.16. Linked-list: Stack, global, and heap cache mapping using eq malloc.

Figure 5.17. Linked-list search stress-test benchmark’s NODE[500, 1000, 1500, 2000]
cache mapping using the equivalence allocator.
Figure 5.18. Linked-list: Stack, global, and heap cache mapping using pool_malloc.

Table 5.5, 5.6, and 5.7 shows the overall cache summaries for all allocators. Overall the equivalence allocator performed the best compared to dl_malloc and poolmalloc. The equivalence allocator reduced the total misses by 3,501,777 misses (22%) compared to dl_malloc. In Table 5.8 we can see that allocation had the most impact on heap data elements. As a result a significant amount of global misses were reduced as well resulting from reduced conflicts with heap objects. The pool allocator performed the worst by increasing misses by 19,996 misses (1%) compared to dl_malloc. A possible explanation is that due to the page alignment, node zero maps to cache set zero. Generally, cache set zero tends to exhibit higher misses because the application’s stack tends to reside around cache set zero.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>103,618,665</td>
<td>103,618,665</td>
<td>89,952,463</td>
<td>13,666,202</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>15,852,278</td>
<td>15,852,278</td>
<td>15,726,274</td>
<td>126,004</td>
</tr>
</tbody>
</table>

Table 5.5. Linked-list search stress-test benchmark’s simulation results for dl_malloc.
<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>103,618,665</td>
<td>103,618,665</td>
<td>89,952,463</td>
<td>13,666,202</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>15,872,274</td>
<td>15,872,274</td>
<td>15,727,789</td>
<td>144,485</td>
</tr>
</tbody>
</table>

Table 5.6. Linked-list search stress-test benchmark’s simulation results for poolmalloc.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>103,618,665</td>
<td>103,618,665</td>
<td>89,952,463</td>
<td>13,666,202</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>12,350,501</td>
<td>12,350,501</td>
<td>12,243,915</td>
<td>106,586</td>
</tr>
</tbody>
</table>

Table 5.7. Linked-list search stress-test benchmark’s simulation results for eqmalloc.

<table>
<thead>
<tr>
<th>Segments</th>
<th>dl_malloc l1-dcache misses</th>
<th>poolmalloc l1-dcache misses</th>
<th>eq_malloc l1-dcache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accesses</td>
<td>Hits</td>
<td>Misses</td>
</tr>
<tr>
<td>Stack</td>
<td>64 M</td>
<td>64 M</td>
<td>126004</td>
</tr>
<tr>
<td>Global</td>
<td>446 K</td>
<td>263 K</td>
<td>182939</td>
</tr>
<tr>
<td>Heap</td>
<td>38 M</td>
<td>22 M</td>
<td>1554335</td>
</tr>
</tbody>
</table>

Table 5.8. Linked-list search: stack, global, and heap segment overall L1 cache misses.

**Multi-list benchmark.** Multi-list stress benchmark performs a search on a multi-list data structure. A multi-list data structure is a list of linked-list data structures. The goal of this benchmark is to stress-test the cache behavior of an allocator’s block placement when objects exhibit different access patterns. In this example the data structure consists of linked-list of head nodes and linked-lists of sub-nodes accessible through the head nodes. Figure 5.19 shows the structure outline; the head nodes, node1, are entries into their sub-lists, node2. The benchmark’s input data consists of random (key1: key2) integer value pair sequences, where key1 values are more frequent than key2. The first value is the key element of the head node and the second value is the key element of the sub-node. The goal of this benchmark is to replicate an access pattern observed in our cache simulator. Every key value pair is a function — variable debug symbol encountered in a Gleipnir generated trace. Thus, the access pattern of key1 values will be different from key2 values. To visualize the behavior of the allocated nodes we will group head nodes into 50 allocations per group. This means that the first 50 head nodes will be denoted as NODE_0, the next 50 as NODE_50,
etc. The head node sub-lists will be denoted as NODE_X_list. For example all NODE_50 sub-nodes will be denoted as NODE_50_list. Figure 5.20 shows the multi-list’s stack, global, and heap references for every 200,000 data store, load, or modifies using dlmalloc. The pattern indicates a uniform reference pattern between stack, global, and heap references. However, Figure 5.21 shows that despite the large number of stack references, the majority of misses are due to heap data.

Figure 5.19. Multi-list search stress-test benchmark.

Figure 5.20. Multi-list: Stack, global, and heap reference counts using dlmalloc.

Further analysis (see Figure 5.22) shows that the majority of references are due to head nodes.
For example notice that the majority of references are attributed to head nodes NODE_0 (yellow), NODE_50 (maroon), and NODE_100 (red). The heap misses denoted with OTHER are heap references to system resources, usually system buffers. In this benchmark we read the data using the `scanf()` function which allocates a buffer to store its data.

![Figure 5.21. Multi-list: Stack, global, and heap miss counts using dlmalloc.](image1)

![Figure 5.22. Multi-list: Heap data references using dlmalloc.](image2)

Figure 5.23 shows the cache misses for heap data. Notice that NODE_0 (yellow), NODE_50 (maroon), and NODE_100 (red) have higher misses. A relatively smaller number of misses is due to the node’s sub-lists. Figure 5.24 shows the benchmarks overall stack, global, and heap data cache set mapping using dl allocator. Heap data is fairly uniformly accessed across all
sets; however, various sets have increased misses on sets with higher stack, and global data accesses.

![Figure 5.23. Multi-list: Heap data misses using dlmalloc.](image1)

Figure 5.23. Multi-list: Heap data misses using \textit{dlmalloc}.

![Figure 5.24. Multi-list: Stack, global, and heap cache mapping using dlmalloc.](image2)

Figure 5.24. Multi-list: \textit{Stack, global, and heap} cache mapping using \textit{dlmalloc}.
Figure 5.25. Multi-list: Heap object’s cache mapping using *dlmalloc*.

Figure 5.25 shows the cache sets mapping of heap nodes. For example we can observe that node sub-lists remain uniformly distributed across cache sets while head nodes are accessed in blocks. The reason for this mapping is that head node objects are interleaved with their sub-lists as they are allocated. An allocation strategy using the equivalence allocator can exploit this access pattern by grouping head node blocks into exclusive cache sets. The goal is to minimize any contention between higher referenced objects and less referenced objects.

Figure 5.26 shows the overall stack, global, and heap segment cache mapping when using the equivalence allocator. We can observe a reduction in cache misses on upper cache sets (32 – 63). Similarly compared to *dlmalloc* overall cache misses on lower numbered cache sets also exhibit lower cache misses. On average every lower numbered cache set, incurs approximately 2000 misses; however, allocating blocks into separate cache areas eliminated heavy cache contention on sets that incurred 10,000 to 100,000 misses when using *dlmalloc*.

In Figure 5.27 we can observe the cache set mapping of stack, global, and heap segments when using the pool allocator. Notice that overall misses remain fairly low except on cache set 51 and cache set 56 due to the extra contention between stack and global data items. The uniform cache set mapping is expected because the pool allocator compacts same sized objects into unique memory regions (viz. head nodes and list nodes are allocated from
different pools) thus accesses to contiguous lists are likely to fill every cache line.

**Figure 5.26.** Multi-list: Stack, global, and heap cache mapping using `eq_malloc`.

**Figure 5.27.** Multi-list: Stack, global, and heap cache mapping using `pool_malloc`.

99
Table 5.9. Multi-list search stress-test benchmark’s simulation results for $dl\text{malloc}$.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>57,839,282</td>
<td>57,839,282</td>
<td>46,267,742</td>
<td>11,571,540</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>910,516</td>
<td>910,516</td>
<td>867,275</td>
<td>43,241</td>
</tr>
</tbody>
</table>

Table 5.10. Multi-list search stress-test benchmark’s simulation results for $pool\text{malloc}$.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>57,839,282</td>
<td>57,839,282</td>
<td>46,267,742</td>
<td>11,571,540</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>54,919</td>
<td>54,919</td>
<td>39,191</td>
<td>15,728</td>
</tr>
</tbody>
</table>

Table 5.11. Multi-list search stress-test benchmark’s simulation results for $eq\text{malloc}$.

<table>
<thead>
<tr>
<th>Segments</th>
<th>dl$\text{malloc}$ l1-dcache misses</th>
<th>pool$\text{malloc}$ l1-dcache misses</th>
<th>eq$\text{malloc}$ l1-dcache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accesses</td>
<td>Hits</td>
<td>Misses</td>
</tr>
<tr>
<td>Stack</td>
<td>36486 K</td>
<td>36462 K</td>
<td>23578</td>
</tr>
<tr>
<td>Global</td>
<td>2935 K</td>
<td>2890 K</td>
<td>45474</td>
</tr>
<tr>
<td>Heap</td>
<td>18417 K</td>
<td>17575 K</td>
<td>841464</td>
</tr>
</tbody>
</table>

Table 5.12. Multi-list search: stack, global, and heap segment overall L1 cache misses.

Table 5.9, 5.10, and 5.11 show the overall cache simulation results for the different allocators. The equivalence allocator’s with the previously described allocation pattern performed the best in reducing overall cache misses by 855,597 (94%) compared to $dl\text{malloc}$. The cache miss reduction can be attributed to the allocation policy as well as better cache line utilization. The pool allocator reduced overall misses by 804,033 misses (88%) compared to $dl\text{malloc}$. From Table 5.12 we can observe that the majority of reduced cache misses stem from reducing heap data misses. $dl\text{malloc}$ incurred 841,464 misses, $pool\text{malloc}$ incurred 98,816 misses, and the $eq\text{-alloc}$ incurred 52,349 misses.

Real benchmarks:

Dijkstra. Dijkstra benchmark from the Mibench suite [20] performs the Dijkstra shortest
path algorithm on a set of nodes. The benchmarks input is a $100 \times 100$ adjacency matrix. The algorithm reads the input file and builds the matrix, then it proceeds to find a number of shortest paths. The main allocated structure is a \textit{QITEM} node (see Table 5.13). The structure represents visited nodes in the graph during the algorithm’s node walk. Figure 5.28 shows the basic principle. Starting from the root node, nodes not visited nodes are enqueued on a \textit{FIFO} list. The head node is dequeued and its neighbours are traversed. If a lower cost path is found the neighbour is enqueued. Every node is 24 bytes including internal structure alignment. In this example The \textit{int} data types are 4 bytes and the pointer is 8 bytes. The compiler will align the pointer on an 8 byte boundary making the structure 24 bytes total.

```
struct _QITEM {
    int iNode;
    int iDist;
    int iPrev;
    struct _QITEM * qNext;
};
```

Table 5.13. Dijkstra node.

![Dijkstra's node walk](image)

Figure 5.28. Dijkstra’s node walk.

The \textit{QITEM} structure definition implies that, assuming the allocator utilizes full cache lines, for a 64 byte cache line size we can pack at most 2 nodes per line and some nodes will be split across 2 cache lines. Considering that the Dijkstra benchmark allocates and deallocates objects relatively often this implies that life-times of objects are \textit{short}. In this implementation the allocation occurs when new nodes are considered for a viable path. The new nodes are enqueued at the end of a \textit{FIFO} queue and only visited when a new node
is appended (allocated) or when nodes are dequeued for cost computing. This is repeated until all nodes are dequeued from the list, meaning that the shortest path was found.

![Figure 5.29. Dijkstra: stack, global, and heap references using dlmalloc.](image)

![Figure 5.30. Dijkstra: stack, global, and heap misses using dlmalloc.](image)

Interestingly if we break down the benchmark’s execution, for dlallocator, we can observe that heap accesses are relatively low compared to global data (see Figure 5.29) and that heap data accounts for very few cache misses (see Figure 5.30). The reason for the majority of global misses are structures AdjMatrix and rgnNodes. AdjMatrix is a two-dimensional matrix that stores the costs between different nodes, and rgnNodes is an array that stores the visited nodes or the path thus far. Figure 5.31 shows the overall cache mapping and cache hits and misses for stack, global, and heap segments using the dlmalloc allocator.
We can also observe that the majority of misses are due to global data, followed by the dynamic objects. Dijkstra’s peak-like memory allocation and deallocation pattern is shown in Chapter 3 Figure 3.4. The peaks are a result of enqueuing nodes at the end of the list. It is somewhat surprising that every set was touched by heap data considering that the average block life-time is short and the application’s dynamic working set size is relatively small. This implies that either the working set size of dynamic blocks exceeds that of a memory page or the allocator policy is not reusing previously freed blocks.

In Figure 5.32 we can see the memory behavior when using the pool allocator. As expected, we do not observe much variation compared to dlmalloc because the allocation pattern allocates only one type of object from each pool. However, we can still compare the overall miss behavior and notice that overall heap misses are reduced. Notice the sets 16 through 38 in Figure 5.31 compared to Figure 5.32; they show that overall misses are lower using the pool allocator. We can explain the miss reduction as a result of object compaction.
Figure 5.32. Dijkstra: stack, global, and heap cache mapping using pool malloc.

Figure 5.33. Dijkstra’s stack, global, and heap cache mapping using eq_malloc.
Remember that the pool allocator incurs no overhead for allocated items from the pool, thus the pool allocator can compact the maximum number of same-sized objects into a cache line. In this benchmark the items are 24 bytes which means that for a 64 byte cache line we can pack 2 full objects into a cache line plus 16 remaining bytes of a 3rd object. This also means that depending on the node element’s access pattern we can potentially have 3 node accesses per cache line. As a consequence we can expect to have a lower number of global misses. In this Dijkstra implementation it is somewhat difficult to apply a one-fits-all allocation policy because the majority of misses stem from global data. Also, the heap access pattern (FIFO pattern) is not well suited to exploit common access patterns. Therefore, to apply our equivalence allocator we can explore an allocation policy that maps objects away from the actively used sets. In this example these are sets 16 − 32 and sets 53−58. Thus our allocation will allocate heap objects from a cache subset using a counter and modulo arithmetic. Figure 5.33 shows the outcome of the previously described allocation pattern using the equivalence allocator. In comparison to Figure 5.31 and Figure 5.32, Figure 5.33 shows the dynamic region exclusively mapped to sets 16 through 40. This has several implications for global and dynamic data cache misses. On one hand the globals are still uniformly mapped to every set but they exhibit greater contention with dynamic data on the mapped sets. On the other hand, global misses are reduced on sets untouched by dynamic data. Therefore, we can expect an overall miss reduction if the difference in misses from exclusively mapping dynamic elements to a portion of the cache outweighs the miss increases on sets that exhibit greater contention. Table 5.14, 5.15, and 5.16 show the overall cache misses on the L1 cache using the pool allocator, dl malloc, and equivalence allocator. Pool allocator results in the least number of overall misses over the standard and equivalence allocator. This reduction can be explained by the amount of data that can be packed into a cache line. Overall misses were reduced by 5,312 misses (11%) compared to dl malloc. As previously noted dl malloc uses boundary tags thus incurs extra 16 bytes of overhead per structure. This means that for a single cache line dl malloc can place at most a single object of 16 bytes and 8 bytes of a second object, thus using more memory. The equivalence
allocator using the previously mentioned allocation pattern decreased the overall misses by 2,353 (5%) compared to \textit{dl\_malloc}. The reason for this reduction is due to the decreased contention between global and heap data on some cache sets. Our simulation confirms our hypothesis as shown in Table 5.17. It shows that heap misses have increased by almost 100% from 3,681 to 6,371 but are offset by the reduction in global misses, for a total overall reduction by 5,047 misses.

<table>
<thead>
<tr>
<th>l1-dcache</th>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>31,562,770</td>
<td>31,562,770</td>
<td>26,029,155</td>
<td>5,533,615</td>
<td></td>
</tr>
<tr>
<td>Demand Misses</td>
<td>47,601</td>
<td>47,601</td>
<td>44,459</td>
<td>3,142</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.14. Dijkstra: simulation results for \textit{dl\_malloc}.

<table>
<thead>
<tr>
<th>l1-dcache</th>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>31,532,848</td>
<td>31,532,848</td>
<td>26,014,208</td>
<td>5,518,640</td>
<td></td>
</tr>
<tr>
<td>Demand Misses</td>
<td>42,289</td>
<td>42,289</td>
<td>39,823</td>
<td>2,466</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.15. Dijkstra: simulation results for \textit{pool\_malloc}.

<table>
<thead>
<tr>
<th>l1-dcache</th>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>31,562,770</td>
<td>31,562,770</td>
<td>26,029,155</td>
<td>5,533,615</td>
<td></td>
</tr>
<tr>
<td>Demand Misses</td>
<td>45,248</td>
<td>45,248</td>
<td>41,306</td>
<td>3,942</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.16. Dijkstra: simulation results for \textit{eq\_malloc}.

<table>
<thead>
<tr>
<th>Segments</th>
<th>\textit{dl_malloc} l1-dcache misses</th>
<th>\textit{pool_malloc} l1-dcache misses</th>
<th>\textit{eq_malloc} l1-dcache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Accesses 6314 K, Hits 6313 K, Misses 598</td>
<td>Accesses 6313 K, Hits 6313 K, Misses 576</td>
<td>Accesses 6313 K, Hits 6313 K, Misses 602</td>
</tr>
<tr>
<td>Global</td>
<td>Accesses 21434 K, Hits 21390 K, Misses 43322</td>
<td>Accesses 21411 K, Hits 21372 K, Misses 38977</td>
<td>Accesses 21464 K, Hits 21425 K, Misses 38275</td>
</tr>
<tr>
<td>Heap</td>
<td>Accesses 3785 K, Hits 3781 K, Misses 3681</td>
<td>Accesses 3807 K, Hits 3804 K, Misses 2736</td>
<td>Accesses 3785 K, Hits 3778 K, Misses 6371</td>
</tr>
</tbody>
</table>

Table 5.17. Dijkstra: stack, global, and heap segment overall L1 cache misses.

\textit{Jpeg} benchmark performs a jpeg compression and decompression. \textit{Jpeg} allocates a fixed number of blocks in the beginning of its execution and then accesses the blocks during runtime. It doesn't perform any deallocation. Note that \textit{Jpeg}\’s source code does not define
any specific structures, therefore we used the Gleipnir framework to identify various block sizes for further analysis (see Table 5.18). We denoted any block that is smaller than 1024 bytes as a *very small block*. Any block that is larger than 1024 bytes but smaller than 2500 bytes we denoted a *small block*. A block that is larger than 2500 bytes but smaller than 32K bytes is a *medium block*. Any block that is larger then 32K bytes (non equivalence region) is a *large block*. We obtained the benchmark’s *stack, global, and heap* cache mappings and behavior as shown in Figure 5.34 using the standard *dlmalloc*. A couple of observations are immediately noticeable. First, because of the large number of dynamic requests, cache is heavily utilized. Likewise, the application’s stack is fairly active, especially around sets 0-10, and sets 52-64. Overall stack data exhibits very few misses with about 10 misses on average. similarly global data occupies every set and is the source of slightly more misses, approximately 20 misses on average per set. Dynamic data miss behavior remains fairly uniform with very slight variations among sets. As can be observed from the figure the heap data misses are the main source of cache misses and remain approximately uniformly distributed at around 500 misses per cache set. In order to further investigate any effects the heap data may have on cache L1 behavior we must further break down dynamic object effects on the cache. For clarity purposes we will plot *very small blocks* and *small blocks*, *medium blocks*, and *large blocks* in three different graphs.

```c
if(sizeofobject < (1024)){
    GL_RECORD_MSTRUCT("VERY_SMALL_BLOCK");
}
else if(sizeofobject < (2500)){
    GL_RECORD_MSTRUCT("SMALL_BLOCK");
}
else if(sizeofobject < (1<<15)){ //32K
    GL_RECORD_MSTRUCT("MEDIUM_BLOCK");
}
else{
    GL_RECORD_MSTRUCT("LARGE_BLOCK");
}
void* tmp = (void *) malloc(sizeofobject);
};
```

Table 5.18. Jpeg dynamic block categories.

In Figure 5.35 we can observe the L1 cache mappings of *very small* and *small blocks*. Their
overall pattern remains fairly uniform with very little variation in their misses. Overall their misses are fairly low with approximately 10-20 misses per cache set. The exception is *small block 4* whose misses are in the range of 40-60. Figure 5.36 shows medium blocks cache mapping. Interestingly the medium blocks are fairly large. Their cache mappings occupy virtually every cache set in cache. The exception is *medium block 0* which maps to a subset of the cache. Their access pattern is constant for block-2, but remains fairly non-uniform for the remaining blocks. Notice that *medium block 3* exhibits greater variations in sets 32 - 48, which explains that it is likely to exhibit more misses on these sets. The misses range from about 10 misses to 200 misses per cache set. Figure 5.37 shows the access pattern of large blocks. Not surprisingly these blocks exhibit uniform accesses and are the result of the majority of misses.

![Figure 5.34. Jpeg: Stack, global, and heap cache mapping using dlmalloc.](image-url)
Figure 5.35. Jpeg: very small and small block cache mapping using `dlmalloc`.

Figure 5.36. Jpeg: medium block cache mapping using `dlmalloc`.

Figure 5.38 shows the stack, global, and heap segment cache behavior using pool allocator. There is little variation compared to `dlmalloc` in Figure 5.34, but there are small observable variations in cache misses on cache sets 0 – 10, and cache miss decreases on higher cache sets. The increase is due to the pool allocator’s implementation and can be explained as follows. A new object of different size will be allocated from a different pool of memory unless the block is larger then 32 Kbytes. This means that every new object will be allocated on a page
boundary thus first instances of same-sized objects will be mapped to cache set 0. Because the Jpeg benchmark utilizes a relatively small number of different sized blocks throughout its execution these blocks will exhibit higher contention around the lower sets. Similarly higher cache sets are left untouched by smaller blocks thus they are likely to exhibit fewer cache misses (see Figure 5.39). The equivalence allocator is unlikely to benefit this benchmark’s dynamic allocation and access pattern due to the large amount of larger that 32K bytes blocks and the fact that the majority of accesses are incurred by these blocks.

![Figure 5.37. Jpeg: large block cache mapping using dlmalloc.](image)

Tables 5.19, 5.20, and 5.21 shows the summary of Jpeg’s cache misses using the different allocators. Surprisingly the pool allocator’s allocation pattern performed the best in reducing the overall misses compared to the standard dlmalloc by 843 misses (2.4%). The miss reduction is previously explained. The Jpeg benchmark allocated 2 small blocks and 3 very small blocks. We can observe from Figure 5.39 that the pool allocator’s allocation strategy will force most blocks to occupy only the lower portion of the cache. This implies that lower contention on higher numbered cache sets may have reduced the overall misses. Furthermore the pool allocator does not perform any cache line block alignment which means that several cache lines that are not fully utilized are likely to be occupied by the neighbouring block. While this may be negligible it can cause those cache lines to be prefetched.
and thereby reduce misses. The equivalence allocators performed the worst with increasing overall misses by 1016 misses compared to \textit{dl\_malloc}.

**Figure 5.38.** Jpeg: stack, global, and heap cache mapping using pool\_allocator.

**Figure 5.39.** Jpeg: heap objects cache mapping using pool\_allocator.
In Table 5.22 we can observe the overall stack, global, and heap data cache behavior. The pool allocator reduced heap misses, but increased global and stack misses compared to $dl\_malloc$. The incurred difference is due to object alignment and object compaction. The equivalence allocator performed the worst in increasing global misses by 1073 compared to $dl\_malloc$. The increase in global misses is somewhat puzzling considering that the heap misses remained approximately the same. Nevertheless, a possible answer for the increase in global misses is that due to the allocator’s implementation, several key internal structures are allocated in the application’s data segment and may be offsetting global elements to sets that exhibit higher contention.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>9159749</td>
<td>9159749</td>
<td>6267969</td>
<td>281780</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>35094</td>
<td>35094</td>
<td>29007</td>
<td>6087</td>
</tr>
</tbody>
</table>

Table 5.19. Jpeg: simulation results for $dl\_malloc$.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>9175178</td>
<td>9175178</td>
<td>6282556</td>
<td>2892622</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>34251</td>
<td>34251</td>
<td>27920</td>
<td>6331</td>
</tr>
</tbody>
</table>

Table 5.20. Jpeg: simulation results for $pool\_malloc$.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>9184398</td>
<td>9184398</td>
<td>6287650</td>
<td>2896748</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>36110</td>
<td>36110</td>
<td>29911</td>
<td>6199</td>
</tr>
</tbody>
</table>

Table 5.21. Jpeg: simulation results for $eq\_malloc$.

<table>
<thead>
<tr>
<th>Segments</th>
<th>$dl_malloc$ 11-dcache misses</th>
<th>$pool_malloc$ 11-dcache misses</th>
<th>$eq_malloc$ 11-dcache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accesses</td>
<td>Hits</td>
<td>Misses</td>
</tr>
<tr>
<td>Stack</td>
<td>5670 K</td>
<td>5669 K</td>
<td>439</td>
</tr>
<tr>
<td>Global</td>
<td>568 K</td>
<td>567 K</td>
<td>1511</td>
</tr>
<tr>
<td>Heap</td>
<td>2920 K</td>
<td>2887 K</td>
<td>33144</td>
</tr>
</tbody>
</table>

Table 5.22. Jpeg: stack, global, and heap segment overall L1 cache misses.
Patricia. Patricia benchmark implements a network routing protocol. It never deallocates nodes and thus exhibits a ramp memory allocation behavior (see Chapter 3). The interesting allocation pattern of Patricia is that it constructs a binary tree during its execution. The benchmark allocates tri-nodes\(^4\) and stores them in a tree. It iterates through the input set and searches the binary tree to find the appropriate location to store the next tri-node. There are three main structures that are accessed during the benchmarks execution, ptree, ptree_mask, mynode (see Table 5.23).

```
struct ptree {
    unsigned long p_key;
    struct ptree_mask *p_m;
    unsigned char p_mlen;
    char p_b;
    struct ptree *p_left;
    struct ptree *p_right;
}

struct ptree_mask{
    unsigned pm_mask;
    void *pm_data;
}

struct MyNode{
    int foo;
    double bar;
}
```

Table 5.23. Patricia’s tri-node.

Structure ptree is the main node for every tri-node. The main structure contains a pointer to the mask later used for comparison with a structure’s key, (unsigned long p_key). Every mask contains another pointer to the mynode structure. The mynode structure is a dummy structure that is only accessed following its allocation. Figure 5.40 is an illustration of a node insertion during the benchmark execution. The algorithm traverses a binary tree and compares the node’s p-key with the input data at every tree level. The algorithm iterates through the tree until it reaches an end-leaf and inserts the new node at the appropriate location. It is apparent that higher level nodes in the tree are accessed more often than lower level nodes. Moreover, cache misses are likely to be higher for elements that are accessed frequently due to the conflicts on cache sets. This of course assumes that there exists cache set contention between the nodes when naively mapped to the cache. Figure 5.41 shows the overall cache access behavior of stack, global, and heap data elements using dlmalloc. We can observe the uniformity of cache misses for global and heap data. Also, notice that Patricia has significantly higher stack and global accesses, but results in significantly higher

\(^4\)A Patricia tri-node consists of three different structure allocations connected through pointers.
heap misses. The heap misses are due to cache set contention between various Patricia nodes. We can observe spikes in heap accesses which represent the frequently accessed nodes that map to those sets.

Figure 5.40. Patricia’s node insertion.

Figure 5.42 shows the cache access pattern for stack, global, and heap data using the pool allocator. As we will explain subsequently the pool allocator exhibits relatively higher misses due to the overlap of frequently accessed objects to same sets. In Figure 5.43 we can observe the overall mapping of ptree, ptree_mask, and mynode structures. We mentioned before that Patricia allocates three different structures, but only two different sized objects of 40 and 16 bytes. This affects object placement in the following way. The same sized objects will be allocated from the same pool. Thus depending on the allocation pattern we will have a memory interleaving between ptree_mask and mynode structures both of which are 16 bytes. This means that using the pool allocator we can expect at most 2 ptree_mask objects per cache line. Note that mynode structure is only accessed once after its allocation. This implies that allocating the mynode structure from the same pool is wasteful. Furthermore, the first allocations exhibit higher references due to Patricia’s binary walk. This means that if higher level nodes (i.e. those closer to the root) map to the same set they will cause greater contention. This behavior is visible in Figure 5.42 and 5.43 where lower level sets exhibit
significantly higher misses, for example set 0 incurs 10,000 misses.

To effectively use an equivalence allocation we must exploit Patricia’s binary allocation and access pattern. A sound allocation strategy could be to attempt to isolate higher level nodes from lower level nodes because the analysis indicates that these nodes incur most misses. This means that lower cache sets (e.g. set 0 – 10) will be occupied by higher level nodes of the binary tree. Our allocation algorithm will utilize a set counter to track the number of allocations starting at base set counter. Because we have an 8-way cache we can allocate up to 6 ways for every object per set incurring limited contention between nodes, and reserve 2 ways for stack and global segments. Our set counter will increment at every allocation and wrap at max sets (64). When we exhaust the threshold we will increment our base set counter and repeat. Our hypothesis is that the effect of this allocation pattern will ensure that higher level nodes (i.e. earlier allocations and therefore frequently accessed nodes) will incur less contention on lower numbered cache sets. Inversely, lower level nodes (i.e. later allocations and therefore less frequently accessed nodes) will incur greater contention on
higher level sets.

Figure 5.42. Patricia: stack, global, and heap cache mapping using pool_malloc.

Figure 5.43. Patricia: heap structures cache mapping using pool_malloc.

Figure 5.44 shows the stack, global, and heap data cache set mapping and behavior using the equivalence allocator and our allocation strategy. Notice that the incurred misses per
Figure 5.44. Patricia: stack, global, and heap cache mapping using eq_malloc.

cache set for heap data items have significantly fewer cache misses on lower numbered sets. For example sets 0 – 20 have cache misses in the range from 10 – 100. In comparison with dl_malloc (Figure 5.41) the same sets have well over 500 misses on average. The misses progressively increase as the number of allocated nodes to these sets increases. Also, notice the sudden increase in heap misses on sets 48 – 56 and another increase on sets 56 – 63. The reason for the increase is due to the allocated ptree mask and mynode structures which we have reserved to that cache area. A further optimization could be to implement a similar allocation strategy even for the mask data structure. Figure 5.45 shows the cache access mapping for Patricia’s tri-node structures. We can observe that the ptree structure misses were significantly reduced as a result of exploiting the benchmark’s allocation and access pattern. The Table 5.24, 5.25, 5.26 show the differences in L1 cache misses. The equivalence allocator reduced the misses by 2647 misses (10%) compared to dl_malloc. Table 5.27 shows the difference in misses for the stack, global, and heap data segments. The pool allocator increased misses by a noticeably higher amount. The increase in misses is likely due to
pool alignment because the lower level nodes of the tree will be allocated from pool memory regions that will, due to pool memory page alignment, map to lower cache sets. From Figure 5.41 we can notice that the benchmark’s stack is fairly active. Thus, if high reference nodes are also mapped to the same cache region we can expect to see higher cache set contention.

![Figure 5.45. Patricia: structure cache mapping using eq_malloc.](image)

<table>
<thead>
<tr>
<th>11-dcache</th>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Demand Fetches</td>
<td>30871290</td>
<td>30871290</td>
<td>18766697</td>
<td>12104593</td>
</tr>
<tr>
<td></td>
<td>Demand Misses</td>
<td>26728</td>
<td>26728</td>
<td>7619</td>
<td>19109</td>
</tr>
</tbody>
</table>

Table 5.24. Patricia: simulation results for dl_malloc.

<table>
<thead>
<tr>
<th>11-dcache</th>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Demand Fetches</td>
<td>30871592</td>
<td>30871592</td>
<td>18766923</td>
<td>12104669</td>
</tr>
<tr>
<td></td>
<td>Demand Misses</td>
<td>66018</td>
<td>66018</td>
<td>42202</td>
<td>23816</td>
</tr>
</tbody>
</table>

Table 5.25. Patricia: simulation results for pool_malloc.


<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Fetches</td>
<td>30871583</td>
<td>30871583</td>
<td>18766914</td>
<td>12104669</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>24081</td>
<td>24081</td>
<td>7371</td>
<td>16710</td>
</tr>
</tbody>
</table>

Table 5.26. Patricia: simulation results for eq malloc.

<table>
<thead>
<tr>
<th>Segments</th>
<th>dl malloc l1-dcache misses</th>
<th>pool malloc l1-dcache misses</th>
<th>eq malloc l1-dcache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accesses</td>
<td>Hits</td>
<td>Misses</td>
</tr>
<tr>
<td>Stack</td>
<td>25090 K</td>
<td>25090 K</td>
<td>43</td>
</tr>
<tr>
<td>Global</td>
<td>4580 K</td>
<td>4577 K</td>
<td>2824</td>
</tr>
<tr>
<td>Heap</td>
<td>1200 K</td>
<td>1177 K</td>
<td>23861</td>
</tr>
</tbody>
</table>

Table 5.27. Patricia: stack, global, and heap segment overall L1 cache misses.

5.4.3. Results Summary

In this subsection we will discuss allocator’s performance with respect to their response time to allocation requests, overall memory utilization, and block placement in terms of cache misses. The time that it takes an allocator to service an allocation request will be termed time-to-service or simply its allocation speed. An optimal allocator will incur minimal time-to-service delay. Maximizing allocation speed usually comes at the cost of fragmentation and locality. In our experiments we have measured the allocation speed as an average of executed instructions using a semi-random allocation/deallocation micro benchmark. Allocation speed is an important metric for applications that do not benefit from cache conscious block placement because a fast allocation implies very little cache thrashing. We will also consider overall memory usage in terms of page utilization. Finally, we will summarize cache simulation results from the previous sections.

Allocation performance.

To stress test the allocators we performed a series of allocation and deallocation micro benchmarks. The benchmark’s main purpose is to test allocators general allocation and deallocation functions rather than to stress test a random allocation pattern. The reason for this is that both pool allocator and equivalence allocator are not designed for allocation

---

5A random stress-test is not an optimal benchmark to compare these allocators because they were not optimized to service random allocation requests.
patterns that are completely random.

<table>
<thead>
<tr>
<th>Initialization (8 bytes)</th>
<th>Allocation (10000 × 8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocator</strong></td>
<td><strong>Instructions</strong></td>
</tr>
<tr>
<td>dl_memalloc</td>
<td>59,916</td>
</tr>
<tr>
<td>pool_memalloc</td>
<td>98m</td>
</tr>
<tr>
<td>eq_memalloc</td>
<td>14,607</td>
</tr>
</tbody>
</table>

(A) (B)

<table>
<thead>
<tr>
<th>Allocation (1000 × (8 byte incr.))</th>
<th>Object allocation/deallocation (1000 steps)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocator</strong></td>
<td><strong>Instructions</strong></td>
</tr>
<tr>
<td>dl_memalloc</td>
<td>318,424</td>
</tr>
<tr>
<td>pool_memalloc</td>
<td>--</td>
</tr>
<tr>
<td>eq_memalloc</td>
<td>39,514,742</td>
</tr>
</tbody>
</table>

(C) (D)

Table 5.28. Allocator’s instruction and data performance comparison.

The main premise of using either allocator is that an allocation pattern is, to a certain degree, predictable. Moreover, as we have explained in Chapter 3, an arbitrary allocation pattern is rare in real world applications. Table 5.28 summarizes the instruction and data references during the tests. The first test (Table 5.28a) was a simple allocation to test the allocators initialization time. It may seem odd that the pool allocator performed the worst, but the performance degradation is the result of linking blocks within a memory pool. The allocator does not implement a pool wilderness chunk. That is to say, there is only one pointer per region that points to the memory pool. Therefore, every block must be linked when the pool is allocated. However, once a memory pool is initialized the allocator incurs very little overhead for new requests (see Table 5.28b). Table 5.28c shows results of an incremental allocation pattern.\(^7\) It may seem surprising to observe the equivalence allocator perform poorly, but the reason is simple. First, the equivalence allocator relies on the user to specify

\(^6\)The first initialization was performed prior to tracing.

\(^7\)Allocation requests for incrementally larger blocks.
the equivalence class to allocate a block. Because an incremental pattern will eventually request blocks that do not fit in a single equivalence class or cache set, the allocator will start performing partitioning for every new block. This means that it will incur overhead to acquire more equivalence regions to satisfy all the requests. An optimization would be to group requests into larger size classes in order to avoid frequent partitioning. Table 5.28d shows the performance results in a series of allocations and deallocations requests. The goal of the benchmark is to test an average number of instructions when allocation deallocation occur. The equivalence class allocator’s block freeing is expensive due to the time it takes to find the appropriate equivalence arena where the block is returned to. In a practical sense this means resetting pointers to equivalence classes; however, finding the pointers may result in traversing every allocated equivalence arena. In general the allocators extra instruction overhead is costly in term of actual performance thus these allocators in their current implementation are unlikely to benefit overall execution times.

Memory utilization. Memory utilization is a key characteristic of memory allocators since it reflects the amount of fragmentation. Virtually all allocators will attempt to minimize memory fragmentation in order maximize memory utilization. Memory utilization is correlated with translation look-aside buffer (TLB) misses. To understand how memory utilization is affected by different allocators we can compare virtual memory usage for different benchmarks. What follows are a series of plots that depict memory utilization of \textit{dlmalloc}, \textit{pool allocator}, and \textit{equivalence allocator}.

In Figure 5.46 we can see the virtual memory utilization of the \textit{multi-list} stress benchmark when ran with the \textit{dlmalloc} allocator. The plot is a horizontal layout of all touched memory references during the benchmarks execution. The \textit{Y-axis} represents 1024 bytes, and the \textit{X-axis} represents a 1024 byte granularity (64 $\times$ 1024 bytes per tic). A single square represents a virtual page. Note that the memory flows from highest $Y$ value (1024) to lowest $Y$ value (0), therefore a contiguous memory page is represented as 4 neighboring vertical lines. In this particular benchmark it is immediately noticeable that the majority of references are to heap data. The red stripe on the right edge are \textit{stack} references. It may seem as if the
allocator is utilizing the memory very well (no white dots in the heap area); however, if we
zoom into the heap area we can notice that there are fragments of memory that were not
referenced (see Figure 5.47).

Note that fragments of unused memory do not necessarily represent fragmentation.
They are simply pockets of heap memory that were either untouched or used for allocator
control blocks. Figure 5.48 shows the multi-list virtual memory utilization using the pool
allocator. We can observe that heap data in the pool allocator resides very close to the
stack. There are a couple of reasons for this. Memory regions assigned by the pool are

\[\text{Figure 5.46. Virtual memory utilization when using }\text{dLmalloc.}\]

\[\text{Figure 5.47. Multi-list: virtual memory utilization when using }\text{dLmalloc (heap zoomed in).}\]

\[\text{Figure 5.48 shows the multi-list virtual memory utilization using the pool allocator. We can observe that heap data in the pool allocator resides very close to the stack. There are a couple of reasons for this. Memory regions assigned by the pool are}\]

\[\text{Depending on the benchmark, we omitted traces for preallocated blocks.}\]
serviced using the `mmap()` system call. This means that they will be serviced from a virtual memory area above the `sbrk_limit` (closer to `stack`). Moreover, because the application is instrumented with Valgrind, its `stack`, `global`, and `heap` regions are packed much closer in the virtual space than when the application is running natively. It is somewhat surprising to see pockets of memory unused because the pool allocator uses only minimal control structure per pool; however, additional testing reveals the reason for the empty pockets of unreferenced data. The `multi-list` benchmark allocates 64 byte sized nodes. This means that allocated objects will be serviced from the same pool of memory. Furthermore, the nature of the benchmark will allocate a head node followed by several sub-nodes. Node alignment data is not referenced as it only serves to align nodes to cache lines. Thus, pockets of unused memory represent locations of alignment data between two consecutive allocations. Figure 5.49 confirms this because we can distinguish small random sized tics that represent head nodes that have an extra pointer type element.

Figure 5.50 shows the virtual memory view of the `multi-list` benchmark using the equivalence allocator. The equivalence allocator also aligns blocks on a cache line, thus we can explain the non referenced areas (white space) in the same way (non used node data) as we did for pool allocator. Notice that the heap memory is used very uniformly. The reason for the uniformity is the cache partitioning during the allocation in the `multi-list` benchmark.
Remember that our allocation pattern splits the cache into two partitions. Memory areas that map to the lower cache sets were reserved for sublists, while memory areas that map to higher cache sets were reserved for head nodes. This example is unique in that memory is evenly utilized. Figure 5.51 shows the virtual memory usage for Patricia benchmark using the equivalence allocator. This is for illustration purposes only because it shows that exploiting memory access patterns will not always yield evenly utilized memory. Table 5.29 and Table 5.30 summarize the heap usage results for all the benchmarks. In terms of total heap and page usage for most benchmark the equivalence allocator performed the worst. The reason for that is that for most benchmarks we have allocated memory to specific cache regions thus wasted fragments of virtual memory. This is expected because the allocator is tailored to exploit access patterns rather than compact memory.

![Figure 5.49. Multi-list: virtual memory utilization using pool allocator (heap zoomed in).](image)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>dl_malloc</th>
<th>pool_malloc</th>
<th>eq_malloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search (64byte node)</td>
<td>232,104</td>
<td>18,830,199</td>
<td>540,600</td>
</tr>
<tr>
<td>Linked-list (64byte node)</td>
<td>529,328</td>
<td>18,643,888</td>
<td>317,168</td>
</tr>
<tr>
<td>Multi-list (64byte node)</td>
<td>1,269,600</td>
<td>19,659,632</td>
<td>2,650,672</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>10,072</td>
<td>10,263,432</td>
<td>26,072</td>
</tr>
<tr>
<td>Jpeg</td>
<td>2,150,120</td>
<td>38,775,656</td>
<td>2,304,328</td>
</tr>
<tr>
<td>Patricia</td>
<td>1,220,337</td>
<td>71,422,273</td>
<td>6,746,072</td>
</tr>
</tbody>
</table>

Table 5.29. Comparing overall heap usage.
Figure 5.50. Multi-list: virtual memory utilization using the equivalence allocator.

Figure 5.51. Patricia: virtual memory utilization using the equivalence allocator.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Used pages total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search (64byte node)</td>
<td>77</td>
</tr>
<tr>
<td>Linked-list (64byte node)</td>
<td>48</td>
</tr>
<tr>
<td>Multi-list (64 byte node)</td>
<td>325</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>54/54</td>
</tr>
<tr>
<td>Jpeg</td>
<td>111/259</td>
</tr>
<tr>
<td>Patricia</td>
<td>346/1763</td>
</tr>
</tbody>
</table>

Table 5.30. Comparing overall page usage.
Summary of cache simulations. Figure 5.52 and Table 5.31 show the summary of cache misses. The figure’s Y-axis is in logarithmic scale. The outcome of potential benefits in using the equivalence allocator are highly dependent on the benchmark. Usually, if the benchmark exhibits common data structures and common access pattern, the equivalence allocator will likely benefit the benchmark’s memory management. More specifically, the stress benchmarks were intended to show that exploiting access patterns can yield substantial L1 cache improvements. If the access patterns are known in real benchmarks (eg. Patricia) then we can use a custom allocator (such as the equivalence class allocator) to exploit and improve overall cache behavior. Examples of benchmarks with non-regular access patterns are Dijkstra and Jpeg. Even though we were able to reduce Dijkstra’s L1 cache misses through analysis of stack and global data behavior it is likely that the allocation strategy is not portable.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DL_malloc</th>
<th>Pool malloc</th>
<th>Eq_malloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary-search (64 byte node)</td>
<td>137,811</td>
<td>81,239</td>
<td>77,945</td>
</tr>
<tr>
<td>Binary-search (48 byte node)</td>
<td>80,256</td>
<td>94,758</td>
<td>77,720</td>
</tr>
<tr>
<td>Binary-search (32 byte node)</td>
<td>78,390</td>
<td>50,494</td>
<td>49,099</td>
</tr>
<tr>
<td>Linked-list (64 byte node)</td>
<td>15,852,278</td>
<td>15,872,274</td>
<td>12,350,501</td>
</tr>
<tr>
<td>Linked-list (40 byte node)</td>
<td>9,332,784</td>
<td>7,654,015</td>
<td>12,805,719</td>
</tr>
<tr>
<td>Linked-list (24 byte node)</td>
<td>5,943,514</td>
<td>4,131,058</td>
<td>5,014,820</td>
</tr>
<tr>
<td>Multi-list (64 byte node)</td>
<td>910,516</td>
<td>106,483</td>
<td>54,919</td>
</tr>
<tr>
<td>Multi-list (48 byte node)</td>
<td>297,343</td>
<td>37,858</td>
<td>54,460</td>
</tr>
<tr>
<td>Multi-list (32 byte node)</td>
<td>221,137</td>
<td>21,906</td>
<td>28,343</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>47,601</td>
<td>42,289</td>
<td>45,248</td>
</tr>
<tr>
<td>Jpeg</td>
<td>35,094</td>
<td>34,251</td>
<td>36,110</td>
</tr>
<tr>
<td>Patricia</td>
<td>26,728</td>
<td>66,018</td>
<td>24,081</td>
</tr>
</tbody>
</table>

Table 5.31. Comparing overall L1 cache misses.

The summary in Table 5.31 shows that for stress benchmarks with a 64 byte node size the equivalence allocator reduced cache misses by 43%, 22%, and 93% for the binary, linked-list,

The figure shows stress-benchmarks for 64 byte node size.
and multi-list search stress benchmark, respectively when compared to \texttt{dl\_malloc}. However, the data indicates that as the benchmark’s node payload size approaches its base size, the pool allocator’s cache line compacting benefits overall cache behavior. There are several reasons for this. First, if the node size cannot use the entire cache line the pool allocator will split unused bytes across a node. However, the equivalence allocator will cause fragmentation because unused memory will go wasted due to the current implementation. Second, the memory footprint is too small to evaluate any potential benefits of allocating conflicting nodes into different cache sets. The real benchmarks show that the equivalence allocator reduced overall L1 cache misses by 5\% and 10\% for Dijkstra and Patricia compared to \texttt{dl\_malloc}, but increased misses by 2.81\% for the Jpeg benchmark compared to \texttt{dl\_malloc}. The pool allocator improved stress benchmarks’ overall L1 cache misses by 41\%, 88\% for binary, and multi-list search respectively but increased misses by less than 1\% for the linked-list search compared to \texttt{dl\_malloc}. For real benchmarks it improved misses for Dijkstra and Jpeg by 11\% and 2\% respectively but increased L1 misses by 60\% for Patricia.

![Figure 5.52. Allocator misses cache summary.](image)

\textit{Average memory access time (AMAT).} The average memory access times is computed using the following formula: \textit{Hit Time} + \textit{Miss Rate} \times \textit{Miss Penalty}. For a hypothetical system consisting of an L1, L2 CPU cache and RAM, the formula is given by: \textit{L1 Hit Time} + \textit{L1 Miss Rate} \times (\textit{L2 Hit Time} + \textit{L2 Miss Rate} \times (\textit{RAM Access Time})). Our simulator’s cache
configuration defined a 32KB L1 cache with 64 bytes per line and 8-way associativity, and a 2MB L2 cache with 64 bytes per line and 8-way associativity. The computed average memory access times shown in Table 5.32 assumed a 1\(\text{ns}\) L1 access time, a 7\(\text{ns}\) L2 access time, and a 100\(\text{ns}\) RAM access time. We can conclude that reducing L1 cache misses have very little effect on average memory access times. This is due to the benchmark’s small memory footprint and thus low cache miss rates especially at L2 cache.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DL malloc</th>
<th>Pool malloc</th>
<th>Eq malloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary-search (64 byte node)</td>
<td>1.10ns</td>
<td>1.11ns</td>
<td>1.11ns</td>
</tr>
<tr>
<td>Binary-search (48 byte node)</td>
<td>1.12ns</td>
<td>1.09ns</td>
<td>1.11ns</td>
</tr>
<tr>
<td>Binary-search (32 byte node)</td>
<td>1.10ns</td>
<td>1.00ns</td>
<td>1.00ns</td>
</tr>
<tr>
<td>Linked-list (64 byte node)</td>
<td>2.05ns</td>
<td>2.05ns</td>
<td>1.84ns</td>
</tr>
<tr>
<td>Linked-list (40 byte node)</td>
<td>1.49ns</td>
<td>1.49ns</td>
<td>1.84ns</td>
</tr>
<tr>
<td>Linked-list (24 byte node)</td>
<td>1.28ns</td>
<td>1.28ns</td>
<td>1.35ns</td>
</tr>
<tr>
<td>Multi-list (64 byte node)</td>
<td>1.18ns</td>
<td>1.00ns</td>
<td>1.00ns</td>
</tr>
<tr>
<td>Multi-list (48 byte node)</td>
<td>1.11ns</td>
<td>1.00ns</td>
<td>1.00ns</td>
</tr>
<tr>
<td>Multi-list (32 byte node)</td>
<td>1.00ns</td>
<td>1.00ns</td>
<td>1.00ns</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>1.00ns</td>
<td>1.00ns</td>
<td>1.00ns</td>
</tr>
<tr>
<td>Jpeg</td>
<td>1.00ns</td>
<td>1.00ns</td>
<td>1.00ns</td>
</tr>
<tr>
<td>Patricia</td>
<td>1.00ns</td>
<td>1.00ns</td>
<td>1.00ns</td>
</tr>
</tbody>
</table>

Table 5.32. Comparing average memory access times.
CHAPTER 6

CONCLUSIONS

The performance of software applications is hindered by a variety of factors, but the CPU-Memory speed gap (or Memory Wall) is the most critical since this speed gap causes the CPU sitting idle waiting for data to be brought from memory to processor caches. While caches offer hardware techniques to alleviate the speed gap, cache indexing inadvertently introduces a non-uniformity of accesses to cache sets. Some sets are used more often than others and the heavily used cache sets cause conflict misses. The non-uniformity of accesses depends heavily on how program objects are located in the user address space.

There are several system components that play a role in object placement: the operating system, the CPU memory subsystems, compiler, and software memory managers. Each entity is responsible for a different level that ultimately shapes an application’s memory layout. The operating system determines the physical memory pages that will be assigned to an application when the application is loaded into memory. The compiler determines the memory layout of the application’s internal structures (stack and global). And the cache memory hierarchy of a CPU determines how a physical (or virtual address) is mapped to a cache line (or set).

To understand the issue of object placement and explore solutions, it is necessary to develop tools that can track memory accesses of program objects and their mapping to cache locations. The topic of application tuning, optimization, and profiling and hardware simulation is complex and with over 30 years of research there is an abundance of literature covering this area. In this dissertation we included a survey of some relevant tools. However, most of the available tools do not generate information that can be used to identify which memory objects frequently conflict, or which elements of a structure are more frequently accessed. To address these issues we developed a tool called *Gleipnir* for tracing memory accesses and a cache simulator *GL-cSim* that maps the accesses to cache sets. This framework allows researchers to better understand object placement, to refactor code or to refactor data.
Tracing dynamic memory accesses is achieved using a combination of `malloc` function wrapping and an automated or user-invoked structure definition mechanism. When an object is allocated Gleipnir will attempt to automatically parse the object’s name from the application’s source-code; however, if the user feels that a more descriptive name is more appropriate he/she can preset the object name using client-call interfaces. In addition, for applications that use manual memory management the framework supports user-interface calls to track any number of arbitrary user-defined memory regions. In order to relate static variable and structure cache behavior the framework can enable debug symbol parsing through the Valgrind framework which relates data load, store, or modify instruction and source code variables. Because this feature slows down the instrumentation speed by an order of magnitude Gleipnir also supports user level calls that enable structure-specific object tracing. Users can also enable or disable program tracing at relevant code sections. Note that the tracing and simulation capabilities of our framework extend beyond the scope presented in this dissertation. To relate application’s data structures and their cache behavior we also implemented a cache simulator `GL-cSim`\(^1\) for fine-grained cache analysis. GL-cSim tracks the cache behavior of all trace-encountered variables and data structures. In addition to overall cache behavior the analysis data provided by GL-cSim includes a detailed per application’s data segment, function, or variable cache analysis. While most other tools provide cache behavior per source-code line or a similar performance metric, GL-cSim is a data-centric simulator that relates cache behavior with respect to individual variables or structures. Therefore, the user can determine the exact variable or structure conflicts that incur most cache misses.

Most modern programming languages rely on dynamically allocated memory for program objects so that memory is acquired only when needed. Software memory managers (viz., allocators) are responsible for associating virtual memory for these dynamically allocated objects. There has been extensive research on designing new and improved or custom allocators. Most of these works focused on improving memory fragmentation and speed with

\(^1\)This simulator is an extensively modified version of DineroIV\[23\].
which an allocation request is completed. A few researchers also explored allocator designs to improve locality of allocated objects, which may lead to better utilization of cache memories.

More importantly, we demonstrated the capabilities of our framework consisting of Gleipnir for tracking memory accesses and relating them back to program objects and the GL-cSim for understanding how these accesses map to cache locations. We demonstrated one application of the framework in designing and evaluating custom memory allocators. The framework can also be used to change access patterns (viz. code refactoring) or change object structures (viz., data refactoring) with the goal of improving cache memory performance.

In this dissertation we included a general description of memory allocation techniques. We then used our framework to evaluate memory allocators for their performance with respect to object placement and the effect of the object placement on cache performance. We developed a new memory allocator known as equivalence class allocator, or eqallocator, which attempts to find (virtual) memory addresses for objects in order to place them in specific equivalence classes. It should be noted that cache sets form equivalence classes - all addresses that map to a cache set form an equivalence class. Thus the eqallocator can be used to place objects in different cache sets, for example, to minimize the number of objects in a class when one of the object in that class is heavily referenced. We used some stress benchmarks and benchmark applications from MiBench suite to compare eqallocator with two widely used allocators, Doug-Lea allocator and pool allocator.

Our conclusion is that any custom allocator requires user’s knowledge on how the objects are created and accessed. We demonstrated this by showing how a detailed understanding of data structures used in stress benchmarks can be used to define equivalence classes for objects to minimize cache conflicts. Our stress-benchmarks; however, indicate that the equivalence allocator’s overall performance is relatively slow due to its implementation and complexity of internal structures. The reason is that managing internal memory pointers at a 64 byte granularity becomes increasingly time consuming as the number of allocated objects grows, especially for objects that span multiple cache lines (i.e. objects larger than 64 bytes). Measuring overall timing differences is difficult due to the extra overhead
incurred by our instrumentation tool. However, hardware support for eq_alloc can overcome the allocation performance.

6.1. Future Work

With the ever increasing complexity of computer programs and software systems there will always be a need for tools for understanding program behavior and explore solutions to remove performance bottlenecks. Although we demonstrated how our tools can aid in the analysis of memory related problems, in its current state, the framework can only be used by those with extensive knowledge in computer architecture and systems software. Moreover, Gleipnir relies on a heavy weight framework (Valgrind) and thus takes a long time to trace large and complex applications. Another limitation of the environment is related how multithreaded applications are traced - threads’ traces are separated, rather than interleaved. We hope to address these limitations in our future research.

We have shown that on average the equivalence allocator incurs twice as many instructions compared to dl_malloc. The reason for this overhead is due to the time spent searching for equivalence memory arenas when allocating new objects on a relatively large list. The outcome of this is likely to be an overall performance degradation despite the reduction in misses due to the extra time spent on allocating objects. A possible improvement could be to introduce a caching mechanisms where recently used arenas are promoted to top of the list. Additional techniques or allocators implementations could be to offload any memory operation to a specialized hardware memory manager thus offsetting any allocation performance drawbacks. Note that the equivalence allocator only affects CPU’s L1 cache behavior because at present our work relies on the virtual address space, which is assumed to be contiguous. However, operating systems map pages from virtual space into pages in physical memory, and the physical pages need not be contiguous. So it is necessary to investigate approaches whereby the OS is informed of the need to maintain contiguous physical addresses.

Software driven memory partitioning and object placement is one mechanisms to affect application’s cache behavior; however, other mechanisms include multiple cache decoders
or *way-partitioning*. A cache decoder is a small hardware unit that uses data’s address bits to index the cache (see Chapter 1), if we assume that the hardware supports reprogramming these units during application’s execution then we could potentially explore the same effect as changing object placement from a purely hardware perspective. Similarly, virtually every CPU cache is designed as a multi-way (or multi-associative) cache. Thus, future hardware designs could support mechanisms to pin various cache ways to specific data-structures.

6.2. Lessons Learned

When we first set out to explore frameworks to support our work we looked at various frameworks that included mechanisms necessary for developing our own tools. We opted for Valgrind because of several built-in features that we deemed invaluable in building fine-grained tracing and profiling tools similar to Gleipnir. However, as our research and understanding of available frameworks progressed we concluded that many features can be replicated using simple programming circumventions. For example tracing debug information requires enabling the complex Valgrind debug-parsing mechanisms which unfortunately slows down the system by an order of magnitude or worse. However, our analysis shows that the majority of cache related problems stems from a few very large data structures which can be traced using Gleipnir’s internal mechanisms that are based purely on tracking memory regions. Moreover, there are several cache related performance metrics that we are not currently taking into account and that need to be addressed, for example the effects of TLB misses on overall system performance, or effects of shared memory regions. Our current simulator is not designed to analyze such performance metrics thus our future design decisions may include building a completely new tool to address various memory hierarchy performance issues from a multi-core or multi-process perspective. From a software design perspective, building a tool such as Gleipnir required an in-depth understanding of various system components as well as good software engineering practices. For example understanding LLC effects required an understanding of operating system virtual to physical mapping as well as kernel virtual page management. The debug parser required an understand of compiler and software interactions. Building the cache simulator required an in-depth un-
derstand of CPU memory subsystems. Overall, the entire experience contributed to a better and more importantly vital understanding of various system components.
BIBLIOGRAPHY


