# Timing-Predictable Memory Allocation In Hard Real-Time Systems

## Dissertation

zur Erlangung des Grades des

*Doktors der Ingenieurwissenschaften*

der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von

**Jörg Herter**

Saarbrücken
2014

| | |
|---|---|
| Dekan: | Prof. Dr. Mark Groves |
| Prüfungsausschuss: | Prof. Dr. Jan Reineke (Vorsitzender)<br>Prof. Dr. Dr. h.c. Reinhard Wilhelm (Gutachter)<br>Prof. Dr. Eljas Soisalon-Soininen (Gutachter)<br>Prof. Dr. Alfons Crespo (Gutachter)<br>Alejandro Salinger, Ph.D. (akademischer Mitarbeiter) |
| Tag des Kolloquiums: | 3. März, 2014 |

**Impressum**

Druck und Verlag: epubli GmbH, Berlin, www.epubli.de
ISBN 978-3-8442-8742-4

# Abstract

For hard real-time applications, tight provable bounds on the application's worst-case execution time must be derivable. Employing dynamic memory allocation, in general, significantly decreases an application's timing predictability. In consequence, current hard real-time applications rely on static memory management. This thesis studies how the predictability issues of dynamic memory allocation can be overcome and dynamic memory allocation be enabled for hard real-time applications. We give a detailed analysis of the predictability challenges imposed on current state-of-the-art timing analyses by dynamic memory allocation. We propose two approaches to overcome these issues and enable dynamic memory allocation for hard real-time systems: automatically transforming dynamic into static allocation and using a novel, cache-aware and predictable memory allocator. Statically transforming dynamic into static memory allocation allows for very precise WCET bounds as all accessed memory addresses are completely known. However, this approach requires much information about the application's allocation behavior to be available statically. For programs where a static precomputation of a suitable allocation scheme is not applicable, we investigate approaches to construct predictable dynamic memory allocators to replace the standard, general-purpose allocators in real-time applications. We present evaluations of the proposed approaches to evidence their practical applicability.

# Zusammenfassung

Harte Echtzeitsysteme bedingen beweisbare obere Schranken bezüglich ihrer maximalen Laufzeit. Die Verwendung dynamischer Speicherverwaltung (DSV) innerhalb eine Anwendung verschlechtert deren Zeitvorhersagbarkeit im Allgemeinen erheblich. Folglich findet sich derzeit lediglich statische Speicherverwaltung in solchen Systemen.
Diese Arbeit untersucht Wege, Probleme bezüglich der Vorhersagbar von Anwendungen, die aus dem Einsatz einer DSV resultieren, zu überbrücken. Aufbauend auf einer Analyse der Probleme, denen sich Zeitanalysen durch DSV konfrontiert sehen, erarbeiten wir zwei Lösungsansätze. Unser erster Ansatz verfolgt eine automatische Transformation einer gegebenen DSV in eine statische Verwaltung. Dieser Ansatz erfordert hinreichend genaue Information über Speicheranforderungen der Anwendung sowie die Lebenszyklen der angeforderten Speicherblöcke. Hinsichtlich Anwendungen, bei denen dieser erste Ansatz nicht anwendbar ist, untersuchen wir neuartige Algorithmen zur Implementierung vorhersagbarer Verfahren zur dynamischen Speicherverwaltung. Auf diesen Algorithmen basierende Speicherverwalter können die für Echtzeitsysteme ungeeigneten, allgemeinen Speicherverwalter bei Bedarf ersetzen. Wir belegen weiter die praktische Anwendbarkeit der von uns vorgeschlagenen Verfahren.

# Acknowledgements

Many people contributed to this thesis in a myriad of ways: through collaborations, enlightening discussions, and guidance, by providing an excellent working and research environment; or by simply being a role model.

First and foremost, I owe my sincere gratitude to my supervisor *Professor Reinhard Wilhelm* for always providing guidance and a research environment that continuously offered countless opportunities and the freedom to steer one's research in a variety of interesting directions.

I would also like to thank all the members of my thesis committee, namely *Professor Eljas Soisalon-Soininen* and *Professor Alfons Crespo* for carefully reviewing this thesis as well as *Professor Jan Reineke* and *Dr. Alejandro Salinger* for serving on my committee.

Some of the research leading to this thesis was done in collaboration with other researchers: *Jan Reineke*, *Peter Backes*, *Sebastian Altmeyer*, and *Christoph Mallon*. To these colleagues I owe a big *Thank You!* for our rewarding collaboration, their insights and help.

Last but not least, I want to thank all other current or previous members of Reinhard Wilhelm's research group that, knowingly or not, had a positive influence on me and/or my research: *Daniel Grund*, *Oleg Parshin*, *Sebastian Hahn*, *Mo-*

*hamed Abdel Maksoud*, *Andreas Abel*, *Claire Maiza*, *Philipp Lucas*, and *Florian Haupenthal*.
In case I inadvertently omitted anybody to whom acknowledgment is due, I sincerely apologize to that person.

# Contents

# 1 Introduction

> But how can I hope to explain myself here; and yet, in some dim, random way, explain myself I must, else all these chapters might be naugh.
>
> HERMAN MELVILLE
> *in Moby Dick (1851)*

Will my plane crash? Will my car's airbag save me in case of a traffic accident? The answer to both questions highly depends on the functional and timing behavior of embedded applications running on controller chips like airplane and airbag controllers. Consider the car crash example where the airbag controller has to correctly and also timely detect the crash and fire the appropriate airbags. Failure to finish within a few milliseconds may be fatal and—even given all computations of the controller were correct—may be considered an application failure. Such computer programs for which the correctness of operations depends not only upon their functional correctness, but also upon the time in which they are performed, are called *real-time applications*. Usually, one further distinguishes between *soft* and *hard real-time systems*. In soft real-time systems, deadlines may sometimes be missed by single operations or procedures, upon which the system may respond with decreased quality of service. Consider, for example, a software for displaying video. If some operation does not finish in a timely manner, single frames may be

dropped while displaying a video. Hence, the application does not fail completely, but the quality is decreased. In hard real-time systems, it is in contrast mandatory for all operations to always meet their respective deadlines. Such strong timing constraints are required of applications for which reacting in time is safety-critical. The airbag controller and the controlling system of an airplane examples from the beginning of this chapter constitute such safety-critical hard real-time applications. In case any of these two applications misses a deadline and does not react timely to a critical event, human lives are at stake.

Automatic proofs that all tasks involved in time-critical computations always meet their deadlines are derived by so-called *schedulability analyses*. These analyses rely in turn on safe bounds on the single tasks' worst-case execution times (WCETs). Deriving safe and precise WCET bounds for a given task is a challenging problem. While a program's functional behavior usually only depends on its input data, a worst-case execution time analysis considering a program's timing behavior has to take into account also the timing behavior—and consequently the initial state—of the hardware on which the program will be executed. Modern embedded hardware, unfortunately, implements a number of average-case execution-time enhancing features like caches, pipelines, and speculation that make timing behavior less predictable. Caches, for instance, strive to reduce processor waiting times by closing the ever increasing gap between increasing processor speeds and slowly decreasing memory latencies. They achieve their goal by exploiting the principle of locality. I.e., the tendency observed in computer programs to access memory locations more likely if they were already accessed recently or if they are close to recently accessed locations. Today, caches are ubiquitous, even in embedded hardware. While significantly decreasing memory access times, they also add to hardware complexity and unpredictability with respect to WCET bounds. A WCET analysis—or more precisely, a cache analysis—needs to address these predictability issues with caches. Simply treating each memory access as a cache miss is, although a conservative approach, not a feasible one. It has been shown that turning off caches completely may result in a 30-fold increase of running times [LTH02]. Hence, the oversimplification that in the worst case all memory accesses are cache misses yields overly pessimistic and virtually useless WCET bounds. Furthermore, some hardware architectures

are susceptible to timing anomalies [LS99, RWT+06]: situations where a locally faster execution leads to an increased global execution time of the application. For such architectures, assuming all memory accesses to be cache misses is even unsafe.

Predictability of a program's timing behavior is not dependent on hardware features alone. Certain programming techniques and features of the programming language used in the implementation of an embedded application also strongly affect the timing predictability of this application. Consider, for example, dynamic memory allocation. Dynamic memory allocation allows programs to request memory during their runtime as well as freeing, i.e., giving back, this memory at any time. Dynamic memory allocation provides desirable advantages over static allocation to programmers and has consequently been supported by most programming languages since the 1960s. While often yielding better structured source code, another, more important advantage of dynamic memory allocation is to alleviate the programmer from finding ways to cope with a small amount of memory. I.e., alleviating him or her from assigning memory addresses such that objects not allocated contemporaneously can share memory locations to reduce overall memory consumption. The C standard [ISO99] defines functions for dynamic memory (de-)allocation. However, hard real-time applications do currently not—or better very rarely—make use of dynamically allocated memory. The unpredictability of an application's cache behavior introduced by dynamic memory allocation simply outweighs its advantages. What are the sources of this unpredictability? Current algorithms for dynamic memory allocators introduce cache unpredictability in two ways. First, they are not designed to provide any guarantees about the cache set that a newly allocated block maps to. Static cache analyses, however, rely on this information to classify memory accesses as cache hits or cache misses. Such a classification for most memory accesses, again, is a prerequisite for the derivation of precise WCET bounds. The second source of cache unpredictability stems from the execution of the allocator itself. Modern allocators manage free memory blocks in internal data structures that themselves reside on the heap. Finding a free block to satisfy allocation requests often involves traversals of these structures to find the best suitable blocks. These traversals and consequently their effects on the cache are statically unpredictable. Hence, also otherwise derived

information about cache contents may be lost whenever the allocator is invoked and traverses its internal data structures. Furthermore, the unpredictability of the cache performance during the traversal itself may lead to drastic overestimations of the execution times of memory (de-)allocations. In summary, using dynamic memory allocation as provided by common programming languages may render a program's cache behavior completely unpredictable.

This thesis investigates two approaches to enable a predictable dynamic memory allocation for hard real-time systems: automatically transforming dynamic into static allocation and using a cache-aware, predictable memory allocator.

**Automatically Replacing Dynamic by Static Memory Allocation** WCET analyses already impose a number of restrictions on the programs they are able to analyze. They require all program loops—or, if recursion is supported, the maximal recursion depth—to be bounded, at least parametrically. In consequence, the number of objects an application may allocate is also bounded. Our first approach investigates how we can exploit this boundedness of allocations to automatically precompute suitable memory addresses for all possibly allocated objects. These addresses can subsequently be used to statically replace dynamic memory allocation by a static allocation scheme. This approach allows for very precise WCET bounds as all accessed memory addresses are completely known. However, memory efficiency, i.e., how efficient memory addresses are shared among objects not allocated at the same time, depends on the precision of the (objects') liveness information that can be derived statically.

**Predictable, Cache-Aware Dynamic Memory Allocation** There are programs, however, where static precomputation of a suitable allocation scheme is not applicable. We already noted that memory efficiency depends on the statically available liveness information. When this information is too imprecise, memory efficiency of the precomputed scheme may degenerate to a point where the memory consumption becomes prohibitively large. Furthermore, consider *reactive systems* that produce outputs upon stimuli from within or outside the system, and virtually run forever. Such systems often run as real-time applications and sufficiently precise WCET bounds may often be derived for how long it may take a

given system to produce an output. However, due to the long execution periods of the overall systems, static address computations for dynamically allocated objects may quickly become infeasible given limited memory and imprecise liveness information. When precomputing suitable memory addresses to replace dynamic memory allocation is not applicable, a predictable dynamic memory allocator may be utilized. Our proposed allocator, CAMA, is such a predictable constant-time dynamic memory allocator that aims at eliminating all sources of unpredictability common in standard allocators. To enable cache-aware memory allocation, CAMA receives allocation requests with a target cache set as an additional argument. The cache influence of the (de-)allocation procedures themselves are bounded and statically known. Predictability and hence compatibility with WCET analyses is achieved as follows. Free blocks are managed in segregated free lists to allow for constant look-up times and hence constant response times. CAMA uses cache-aware splitting and coalescing techniques relying on an indirect free block management to keep external fragmentation low. Internal fragmentation is reduced by using multi-level segregated lists.

The focus of this thesis lies on the second approach, constructing a versatile, cache-predictable dynamic memory allocator to replace general-purpose allocators in real-time applications. A more detailed summary of the contributions of this thesis is given in the following subsection.

## 1.1 Contributions and Structure of this Thesis

### 1.1.1 Contributions of this Thesis

This thesis discusses the applicability of dynamic memory allocation in real-time systems. It extends our main publications on cache-aware memory allocation [HBHR11] and static precomputations of memory addresses for data structures [HR09, HA10] in that it provides a uniform presentation and significant improvements of those approaches, a thorough and formal discussion of the challenges that dynamic memory allocation poses in a hard real-time environment, and an extended evaluation of our proposed techniques.

The following paragraphs summarize the technical contributions of this thesis.

**Discussion of the Challenges Introduced by Dynamic Memory Allocation for State-Of-The-Art Timing Analyses** We give a comprehensive overview on existing dynamic memory allocation techniques and the current state-of-the-art of static cache analysis. Furthermore, we formally show how and to what degree dynamic memory allocation introduces unpredictability into the presented cache analyses.

**Algorithms to Statically Precompute Data Locations to Replace Dynamic Memory Allocation** We propose algorithms to statically precompute addresses for a program's otherwise dynamically allocated objects. These addresses can be used to subsequently transform dynamic into static memory allocation and hence circumvent all predictability issues of the dynamic approach. We furthermore discuss for what classes of real-time applications such an approach is applicable.

**Predictable, Cache-Aware Memory Allocator** We propose CAMA, a novel, highly configurable dynamic memory allocator, directly tailored towards usage in real-time systems. The key innovation is CAMA's indirect management of memory blocks. In contrast to other, existing dynamic memory allocators, CAMA does not store data necessary to manage memory blocks directly at the respective blocks themselves. Internal management information is instead stored in so-called descriptors: small management units with a statically predefined cache-set mapping. CAMA is able to manage memory blocks by only working on their descriptors. Even memory consumption optimizing operations like splitting and merging are available and can be performed accessing descriptors only. Using descriptors allows to give strong guarantees about CAMA's influence on the systems' cache. Cache predictability is further increased by cache-set guided allocations. I.e., CAMA allows the user to pass an additional argument to the allocation routine that defines to which cache set the returned memory address shall be mapped. Furthermore, segregated free-lists are used to enable a constant-time lookup for currently free blocks during allocations. The latter is a well-known

technique used in constant-time memory allocation algorithms that could be easily adapted to work with CAMA's descriptors.

**Evaluation of the Memory Consumption of Several Dynamic Memory Allocators on Different (De-)Allocation Sequences, Typical for Real-Time Applications** We measure the memory consumption of a meaningful set of allocators for randomized (de-)allocation sequences modeling allocation behaviors typical for real-time systems to evidence the competitiveness of our proposed allocator. Furthermore, we give a comprehensive discussion of the different sources of memory waste contributing to the measured overall memory consumption. I.e., making explicit the contribution of internal and external memory fragmentation as well as incomplete memory use to the overall memory consumption. We furthermore identify the design choices contributing to these different sources of memory waste.

### 1.1.2 Structure of this Thesis

This thesis is structured as follows. Chapter 2 introduces dynamic memory allocation, caches, and worst-case execution time (WCET) analysis. This chapter discusses the state-of-the-art of these concepts as well as their respective research histories. The challenges this thesis tackles are also identified in this chapter. We propose and discuss our two approaches to enable dynamic memory allocation for hard real-time systems in Chapter 3 and Chapter 4, respectively. Experimental evaluations of both approaches are presented and discussed in Chapter 5. In this chapter, we also evaluate and discuss typical memory allocation patterns of hard real-time systems and how well different allocation algorithms cope with those. We conclude in Chapter 6. A brief discussion of future research directions based on our proposed solutions follows in Chapter 7.

### Publications Contributing to this Thesis

Key parts of this thesis have already been published in the proceedings of peer-reviewed conferences as well as peer-reviewed workshops. The following publications contributed to key parts of this thesis:

- Jörg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. **CAMA: A Predictable Cache-Aware Memory Allocator**. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS'11)*, 2011.
- Jörg Herter, Jan Reineke, and Reinhard Wilhelm. **CAMA: Cache-Aware Memory Allocation for WCET Analysis**. In *Proceedings of the Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems (ECRTS'08)*, 2008.
- Jörg Herter and Jan Reineke. **Making Dynamic Memory Allocation Static To Support WCET Analyses**. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.
- Jörg Herter and Sebastian Altmeyer. **Precomputing Memory Locations for Parametric Allocations**. In *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2010.

# 2

# On Dynamic Memory Allocation, Caches, and Static WCET Analysis

> The advanced reader who skips parts that appear too elementary may miss more than the reader who skips parts that appear too complex.
>
> GYÖRGY PÓLYA
> *in Mathematics and Plausible Reasoning: Patterns of plausible inference (1954)*

## 2.1 Chapter Overview

This chapter introduces general notions of dynamic memory allocation, caches, and cache analysis in the context of a worst-case execution time (WCET) analysis. Section 2.2 elaborates on dynamic memory allocation and summarizes relevant work in this area. In Section 2.3, general cache notions are introduced. This section explains the architecture of caches, their merits as well as pitfalls. The state-of-the-art of static cache analysis is reviewed in Section 2.4. This final introductory chapter also elaborates on the challenges dynamic memory allocation and caches themselves pose on the derivation of tight WCET bounds. A detailed

overview on relevant literature for further reading on the concepts and techniques introduced in this chapter is given in Section 2.5.

## 2.2 Dynamic Memory Allocation

> Management of many is the same as management of few. It is a matter of organization.
>
> SUN TZU
> *in The Art of War (c. 6th century BC)*

**Dynamic memory allocation** or sometimes also called dynamic storage allocation, denotes techniques that allow programs to request memory during their runtime as well as freeing, i.e., giving back, this memory at any time. We call a request for a block of memory an *allocation*, while the term *deallocation* denotes the freeing of a memory block. Consequently, a *dynamic memory allocator* is an application that keeps track of which blocks of memory are currently in use by a program, and which are free. When receiving an allocation request, the dynamic memory allocator has to select a free block it deems suitable to satisfy the request. This decision can in general not be revised later. Consider for example a language like C that allows the programmer to access dynamically allocated objects via arbitrarily complex pointer arithmetics. As neither the compiler nor the allocator can in general find and update all pointers to dynamically allocated objects, an allocator that moves in-use blocks around may break the program semantics. When receiving a deallocation request, it must change the status of the respective memory block from “in use” to “free”. Therefore, a dynamic memory allocator is an *online algorithm* as it must respond to requests immediately and in the strict sequence the application presents them to it. As allocators cannot compact memory by moving allocated memory blocks around to have free and in use blocks contiguous in memory, they are prone to *memory fragmentation*.

**Memory fragmentation** refers to the fragmenting of the available memory into blocks that are in use, i.e., allocated by an application, separated by blocks

that are either free or used by the allocator itself. Fragmentation becomes a problem when each free block is either too small to satisfy a waiting request to the allocator or the allocator uses an overly large amount of memory for its internal management data. Note that fragmentation may arise from two different sources: through management overhead, including rounded block sizes, and bad placement decisions. To account for this, two types of fragmentation are distinguished: *internal fragmentation* and *external fragmentation*, respectively [Ran69]. Internal fragmentation denotes all memory not usable by the application due to rounded up requests to ensure block alignment or to map blocks to given size classes. Furthermore, storage used by the allocator to place its own management data contributes to internal fragmentation. This kind of fragmentation is independent of the processed (de-)allocation sequence and may be statically bounded with reasonable precision. External fragmentation is caused by the inability of the allocator to use free memory either because it is fragmented into blocks smaller than the requested size or free blocks cannot be found by the allocator when requested. This second kind of fragmentation is dependent on the sequence of requests posed to the allocator and consequently harder to analyze statically. For some allocators, memory waste can additionally become problematic when, given a waiting allocation request, sufficiently large free blocks exist, but the allocator cannot find any of these blocks.

How can we precisely quantify fragmentation, especially external fragmentation due to the inability of the allocator to use free memory to satisfy waiting allocation requests? Whether fragmentation is problematic (or really existent in a sense) depends not only on the number and sizes of holes of free memory segregating in-use blocks, but also on the future memory requests of the application and whether the allocator can then use these holes to satisfy them. If the allocator can later make use of these holes, the current fragmentation may be seen as prearrangement for future requests, instead of an increased memory consumption. We may conclude that fragmentation is less of an issue at times of lower overall memory usage than when an application's memory usage is at its highest. Or, in the words of [WJNB95], the "average fragmentation is less important than peak fragmentation—scattered holes in the heap most of the times may not be a problem if those holes are well-filled when it counts." To account for those

observations, we quantify the fragmentation an applications causes within a given allocator as the ratio of the maximum memory usage to the maximum memory need of the application. Hence, we define (the percentage of) fragmentation as:

$$fragmentation = \left( \frac{max.\ memory\ usage}{max.\ memory\ need} - 1 \right) \cdot 100\%$$

High fragmentation bears the potential to become a disastrous problem; at least when applications are allowed to allocate arbitrarily sized memory blocks at arbitrary times and also dispose of them any time they choose. Especially in embedded systems where memory is more limited, a program failure due to memory exhaustion for seemingly moderate memory requirements of the application is an ever-present threat that cannot be ignored. But may high fragmentation really occur in practice or are memory allocators sophisticated enough to make efficient use of a system's memory? Unfortunately, there is no algorithm for dynamic memory allocation that ensures efficient memory usage. Even worse, not "only are there no provable good allocation algorithms, there are proofs that any allocator will be bad for some possible applications." [WJNB95, Rob71, GGU72, Rob74, Rob77]. However, real programs do in general not allocate arbitrarily sized blocks at arbitrary times. They are designed to solve a specific problem which certainly has a strong effect on their allocations patterns. Hence, real applications may exhibit regularities rather than random allocation behavior that may be exploited by allocators to use memory more efficiently. Nevertheless, [WJNB95] concludes that these regularities are still surprisingly poorly understood, despite 35 years of allocator research at the time the cited paper was published.

**Dynamic memory allocators** try to counteract and minimize fragmentation mainly with three techniques: (1) splitting larger memory blocks to satisfy requests for smaller blocks, (2) coalescing blocks to yield larger free blocks, and (3) applying an appropriate placement choice. Splitting a larger free memory block into two smaller blocks when lacking a free block just large enough to satisfy a waiting request is intuitively more favorable than getting additional memory to satisfy the request. Also, coalescing two adjacent free blocks into one larger free block is often advantageous. A larger block has a higher probability of being

useful to satisfy future allocation requests than two smaller blocks. Given the allocator's ability to split blocks, the large block can be used to satisfy requests for small and large blocks. Still, both operations introduce additional computational costs for the allocator that may decrease their usefulness for some applications. Consider for example the following simple sequence of requests posed to an allocator

$$\left(\alpha_8^x \alpha_8^y \delta^x \delta^y\right)^+$$

where $\alpha_{size}^{addr}$ is an allocation request for *size* bytes that the allocator satisfies by a pointer to memory address *addr* and $\delta^{addr}$ is a request to deallocate the memory block starting at memory address *addr*. Furthermore, assume the blocks allocated in the first two requests were adjacent in memory. An allocator that uses splitting and coalescing to counteract fragmentation may in this scenario merge the two free blocks after the two deallocation requests (requests three and four in the sequence) and split the resulting larger block again, once the sequence repeats. Hence, nothing is gained by always merging when possible after a deallocation and these merge operations just add to the computational costs of dynamic memory allocation. To prevent such kind of behavior, some allocators use *deferred coalescing*. I.e., they avoid coalescing at deallocation requests, but use it intermittently to counteract fragmentation at times where they would without coalescing be forced to request additional memory instead of efficiently using already claimed memory.

Most of the basic designs still used in dynamic memory allocators were conceived in the 1960's or even the late 1950's, including *sequential fits*, *segregated free lists*, and *buddy systems* [Knu97, WJNB95]. Sequential fit allocators manage all currently free blocks in a single linked list. This list is typically doubly-linked "so that entries may conveniently be deleted from random parts of the list" [Knu97]. To speed up coalescing operations even more, Knuth's boundary tag technique [Knu97] is normally implemented in these allocators [WJNB95]. While easy to implement, sequential fits do not scale well when a large number of free blocks needs to be managed. Best-fit sequential fits that always search for a smallest free block still large enough to satisfy an allocation request have time costs linear in the number of managed free blocks. Returning always the first block large enough to

fulfill a request (first fit sequential fit) may yield better average case performance with respect to time costs, but may also introduce more splitting operations and thus negating any positive effect from shorter list traversals. Searching for a block of (at least) a given size can be accelerated by having dedicated segregated free lists for different block sizes. A simple segregated free list allocator implementing this idea was described by Comfort in 1964 which used three segregated free lists for single-, double-, and triple-word blocks [Com64]. The described allocator is already able to perform simple split operations. If a double-word block is requested but the respective free list is empty, the allocator splits a triple-word block into a single- and double-word block. Analogously, double-word blocks can be used to get two single-word blocks. To make this concept of segregated free lists generally applicable, a consequent variation is to collect blocks of the same size class, but not necessarily the same exact size in segregated free lists. Blocks from a given size class can be used to satisfy requests for any size equal to or smaller than the smallest size contained in that size class. A common scheme to build size classes is to use size classes of powers of two, e.g., classes containing blocks of at least 4, 8, 16 bytes, and rounding-up allocation requests to the next larger size class [WJNB95]. Upon allocation, the allocator just needs to determine the appropriate free list to serve a request for the given size and return *any* block from that list.

Segregated free-list allocators can be further subdivided into *simple segregated storage* and *segregated fits*. Following the notions of [WJNB95], simple segregated storage denotes an algorithm that completely avoids splitting in order to serve requests for smaller blocks using larger blocks. If a free list is empty, one or more virtual memory pages of additional memory are requested from the underlying operating system. These are then split into same-sized blocks, strung together, and put into the empty free list. Hence, simple segregated storage always yields areas (relatively large units like pages) that contain blocks of only one size class. This gives the allocator two advantages. First, no headers are required for managing single in-use blocks; size information can be recorded per page. Second, coalescing can be implemented by simply making a whole page available (for all size classes) once all blocks of the page became free. Segregated fits handle empty free lists by looking for a non-empty larger size class and splitting a larger

than requested block. If no such block is available, just the minimal amount of memory to satisfy the request is requested from the underlying operating system. Both variants allow for fast, constant-time allocation and deallocation procedures; where simple segregated storage is in general faster than segregated fits. Especially simple segregated storage is usually quite fast when objects of a given, single size class are repeatedly deallocated and re-allocated. The advantage of simple segregated storage, avoiding split operations and its simple coalescing, turns into a disadvantage with respect to memory fragmentation. The worst-case application would allocate many blocks of one size class, deallocate all but one of those blocks again and then do the same for many other size classes. This application would eventually cause the allocator to use one page per allocated object. Compared to sequential fits, segregated free lists provide a constant-time access to a suitable free block without any search. Even better: a segregated free list's "first fit" policy that always returns the first free block in a free list, actually constitutes a "good fit" policy due to the sorting by size classes. However, the need for rounding requested sizes and using possibly much larger blocks than needed bears the potential for high memory fragmentation. Buddy systems are yet another variant of segregated free list allocators that provide efficient splitting and coalescing. A free memory block may in such an allocator only be merged with its *buddy* block. This block can efficiently be determined by a simple address computation.

Buddy systems also guarantee that a block will always be either entirely available and hence free to merge or will be an unavailable block. Consequently, coalescing is fast and requires little memory overhead. Normally, a single bit indicating whether or not a block is entirely free suffices; as the buddy systems remove the need to *link*, i.e., reference, neighboring blocks for merging. The three best-known buddy schemes are *binary buddies*, *fibonacci buddies*, and *weighted buddies* [PN77, WJNB95]. Binary buddies are the earliest and simplest variant of a buddy system, firstly described in [Kno65]. In binary buddy schemes, all buddy sizes are a power of two and each buddy is divided into two equal parts. This makes address computations very simple: all blocks are aligned on the boundary of a power-of-two offset from the starting address of heap memory and each bit of a block's offset represents one level in the systems hierarchical splitting. I.e., if a bit is set, the block is the second block of a pair of buddies, otherwise the

first. Despite efficient operations enabled by this simple address scheme, closer size spacings may still be more desirable with respect to internal fragmentation. Fibonacci buddies [Hir73] provide such a closer size spacing by using size classes based on a Fibonacci series. In this scheme, blocks can be split into two blocks of (different) sizes that are also in the series, as each Fibonacci number is the sum of its two predecessors in the series. Weighted buddies [SP74] base their size classes again on power-of-two size classes, however, additionally add three-times-a-power-of-two size classes in-between. The used series for building size classes is thus: $2, 3, 4, 6, 8, 12, 16, 24, \ldots$ (words). In this scheme, power-of-two blocks are split as in the binary buddy system, whereas blocks divisible by three can be split in two different ways. They may be split evenly in two, yielding two smaller three-times-a-power-of-two blocks, or be split in a 1 to 2 ratio, yielding two smaller power-of-two blocks (of different sizes). While offering very efficient coalescing and splitting operations, in experiments using real and (unfortunately in most studies) synthetic traces, buddy systems generally exhibit significantly more fragmentation than other segregated fits [WJNB95].

**Allocators that may be used in real-time systems** are restricted to designs that allow for constant-time allocation and deallocation operations. Hence in theory, any segregated free-list allocator that does not use searching to implement a best-fit strategy but relies on a first-fit strategy that returns always the first (or last) element of an appropriate free list may be used within a real-time system. In practice, however, most segregated fit allocators are still unsuitable. When implementing techniques to counteract fragmentation like splitting and merging, execution times may be bounded only by a constant, but overly large number. Simpler segregated fits without further means to decreasing fragmentation, however, bear the risk of failing due to not being able to cope with the often very limited memory available in real-time systems. In 1995, Ogasawara proposed a segregated-fit allocator, called `Half-Fit`, explicitly designed for usage within real-time applications. Ogasawara aimed at a dynamic memory allocation algorithm "whose memory efficiency is better than that of buddy systems" and "that touches very few cache lines" [Oga95]. The second design goal was motivated by the observation that, in the worst case, "buddy systems access many memory

addresses at both allocation and deallocation time", as split and merge operations may occur. Furthermore, a "high probability that the accesses will result in cache misses and TLB entry misses, since buddies are distant from each other when free blocks are large" was observed [Oga95]. Hence, the constant worst-case time bound for buddy systems easily becomes forbiddingly large. `Half-Fit` circumvents these issues as follows. The allocator groups free blocks with sizes in the range $\left[2^i, 2^{i+1}\right)$ in a segregated free-list indexed by $i$. Upon an allocation request for $s$ bytes, the allocator returns the first block from the free list indexed by $i'$, where

$$i' = \begin{cases} 0 & \text{if } s = 1 \\ \lfloor \log_2 (s-1) \rfloor + 1 & \text{otherwise} \end{cases}$$

When the free list indexed by $i'$ does not contain any blocks, a block from the subsequent non-empty free list whose index is closest to $i'$ is taken. To avoid the need for linear searching or other non-constant-time operations, `Half-Fit` maintains a word-length bit vector to keep track of which free lists are empty and which are not. If the free block is larger than needed, the algorithm splits it into two blocks: one large enough to satisfy the request and a remainder that is relinked on the appropriate free list. Upon deallocation, the algorithms checks whether adjacent memory blocks are also free. This can be done in constant time as links to physically adjacent blocks are always maintained by the allocator and at most 3 blocks are to be merged at a time. The index of the free list to hold the newly freed block can be easily computed as $i = \lfloor \log_2 s \rfloor$, where $s$ is the size of the respective block. As `Half-Fit`'s simplified allocation and deallocation operations can be implemented by a small number of machine instructions, its WCET can be tightly bounded well below bounds provable for binary buddy systems [Oga95]. For simulations using synthetic (de-)allocation sequences and a limited amount of memory, the failure ratios of `Half-Fit` are strictly smaller than those of a binary buddy allocator [Oga95]. Analytically, the worst-case fragmentation of a binary buddy system can be bounded by $2\mathcal{M}(1 + \lceil \log_2 m \rceil)$, `Half-Fit`'s worst-case memory consumption, however, is unbounded when block sizes are not rounded-up and bounded by $2\mathcal{M}(1 + \lceil \log_2 m \rceil)$, too, otherwise. Where $\mathcal{M}$ denotes the maximal live memory and $m$ the largest block size manageable by the allocator. We formally show those claims in the following paragraphs.

**Theorem 2.1** (Worst-Case Memory Consumption of a Binary Buddy)**.** *The worst-case memory consumption including combined internal and external fragmentation of a binary buddy system is* $2\mathcal{M}(1 + \lceil \log_2 m \rceil)$*, where* $\mathcal{M}$ *is the peak memory need and m the largest allocation/block size. We observe that due to rounded block sizes, internal fragmentation may cause a peak memory use of approximately two times the peak memory need. External fragmentation (for the rounded blocks) is bounded by* $\mathcal{M}' \cdot (1 + \log_2 m')$*, where* $\mathcal{M}'$ *is the peak memory need for rounded blocks,* $m'$ *the maximal actual block size.*

We prove the claimed bounds on external and internal fragmentation separately.

*Proof.* Assume a binary buddy allocator as described in [Knu97], where all (managed) memory blocks have a power-of-two size; as do allocation requests. Furthermore, memory blocks/requests have maximal size $m = 2^r$. Assume a total memory of $\mathcal{I} = 2^{x+r}$. Then, the worst-case memory usage $\mathcal{H}$ for a peak memory need $\mathcal{M}$ does not exceed $\mathcal{M} \cdot (1 + \log_2 m)$. We show the claim by induction over $r$. For $r = 0$, $\mathcal{H} \leq \mathcal{M} \cdot (1 + \log_2 m) = \mathcal{M}$ trivially holds as all blocks are of the same size and hence no external fragmentation can occur. Assume the claim holds for $r = n$. We show that it must then consequently hold for $r = n + 1$, too, i.e., $\mathcal{H} \leq \mathcal{M} \cdot \left(1 + \log_2 \left(2^{n+1}\right)\right) = \mathcal{M} \cdot (1 + n + 1)$. Per induction, $\mathcal{M} \cdot (1 + n)$ memory is sufficient to serve all the requests for blocks of sizes less than or equal to $2^n$. We observe that due to the allocator's policy to always select a smallest available free block to satisfy a request (possibly including a split operation for this smallest block) all requests for blocks of sizes $\leq 2^n$ are served from the same blocks of size $\leq 2^{n+1}$. Hence, the remaining $\mathcal{M}$ memory is guaranteed to be untouched and can always be split into completely free blocks of size $2^{n+1}$ (assuming such a size is requested and hence $\mathcal{M}$ is large enough). With the same reasoning as in the $(r = 0)$-case, we can serve possible remaining requests for blocks of size $2^{n+1}$ with those blocks. Again, as all blocks are of the same size, no further fragmentation can occur. Furthermore, the maximal requested amount of memory in blocks of size $2^{n+1}$ obviously has to be less than the peak memory need of $\mathcal{M}$. □

We note that such a buddy system allocator does not suffer from internal fragmentation. However, the worst-case bound on $\mathcal{H}$ requires strong assumptions

about the requested block sizes: they have to be a power of two. In practice, not all requests are for blocks of power-of-two sizes. Hence, allocators implementing such buddy systems round-up all requests to the next power-of-two. This additional internal fragmentation is maximal for requests for blocks of size $2^{r-1}+1$, which are rounded up to $2^r$, again assuming $2^r$ to be the maximal block size. This worst-case internal fragmentation can be easily bounded by a factor of 2, i.e., $\mathcal{H} \leq 2 \cdot \mathcal{M}$.

*Proof.* Let there be $n$ requests for blocks of size $2^{r-1}+1$ so that the allocator always serves blocks of size $2^r$. Putting peak memory need and peak memory use into relation yields the desired factor:

$$\lim_{r\mapsto\infty} \frac{\mathcal{H}}{\mathcal{M}} = \lim_{r\mapsto\infty} \frac{n \cdot 2^r}{n \cdot (2^{r-1}+1)} = \lim_{r\mapsto\infty} \frac{n \cdot 2^{r-1}}{n \cdot 2^{r-1}} \cdot \frac{2}{1+\frac{1}{2^{r-1}}} \overset{r\mapsto\infty}{\to} \frac{2}{1+0} = 2$$

□

In summary, total memory consumption of a buddy system for a peak memory need of $\mathcal{M}$ is bounded by $2\mathcal{M}(1+\lceil \log_2 m \rceil)$ as claimed in Theorem 2.1.

**Theorem 2.2** (Worst-Case Memory Consumption of `Half-Fit`)**.** *The worst-case memory consumption of* `Half-Fit` *is unbounded when block sizes are not rounded up to the next power of two (and the allocator does not give memory back to the operating system), as initially proposed in [Oga95]. If block sizes are rounded to the next power of two,* `Half-Fit`*'s memory consumption can be bounded by* $2\mathcal{M}(1+\lceil \log_2 m \rceil)$.

*Proof.* We show the claim for an implementation of `Half-Fit` without rounded block sizes first. Assume the following (de-)allocation sequence:

$$\left(\alpha^x_{2^{i_{max}+1}-1}\delta^x\right)^n$$

that allocates a block of the maximal block size, deallocates this block again, and repeats those two operations. We observe that when blocks are deallocated, `Half-Fit` puts them in the free list for size class $[2^{i_{max}}, 2^{i_{max}+1}-1]$. At allocations, however, the allocator cannot pick a block from this size class, as this class also contains blocks smaller than requested, i.e., those of sizes $[2^{i_{max}}, 2^{i_{max}+1}-2]$. Hence, assuming no memory is returned to the operating system, a new block needs to be created at each allocation although fitting ones would be available. □

*Proof.* Let us consider next an instantiation of the `Half-Fit` algorithm that rounds requested sizes to the next power of two. In this case, we can conclude a worst-case bound of $2\mathcal{M}(1 + \lceil \log_2 m \rceil)$ on the actual memory use for a peak memory need of $\mathcal{M}$ from two observations. Rounding requests to the next power of two may at most double the memory need by adding internal fragmentation as shown earlier (observation one). Rounding all requests also entails that the internal size classes contain only blocks of the same size: the lower bound of the respective size class. All blocks of a size classes are of equal size and the algorithm always selects a block from the smallest size class large enough to satisfy the request. Hence, such an instance of the `Half-Fit` algorithm selects free blocks according to the same strategy as a binary buddy system with all block sizes being a power of two (observation two). The worst-case bound on the memory use of such a binary buddy system is $\mathcal{M} \cdot (1 + \log_2 m)$ as shown before. □

It is interesting to note that there exists memory waste in `Half-Fit` when not rounding the requested sizes that can neither be attributed to internal nor (traditional) external fragmentation. In `Half-Fit`, free blocks larger than the base size of their respective free list can never be used to serve requests for blocks of sizes larger than this base size. There may be free blocks suitable to satisfy requests, but the lookup mechanism does not find them as it always searches for blocks in the next bigger size class (than requested). We exploited this flaw to hide suitable, free blocks from the allocator when showing that, without rounding requests, `Half-Fit` may be forced to allocate an unbounded amount of memory for a fixed peak memory use. Ogasawara coined the term *incomplete memory use* to describe this problem of `Half-Fit` [Oga95]. Rounding up requested sizes to the next size class simply transforms incomplete memory use into internal fragmentation which is more predictable and allows for much better worst-case bounds on the memory use to be proven. Ogasawara did, however, not propose such a transformation of incomplete memory use into internal fragmentation, but opted for having no internal fragmentation. This is likely due to him not observing incomplete memory use to be overly occurring within his benchmark programs, while the otherwise introduced internal fragmentation would have been observable.

**TLSF** Masmano et al. proposed a more refined scheme to building size classes for their constant-time dynamic memory allocator TLSF [MRCR04, MRR+08]. While otherwise behaving analogously to Half-Fit (with rounded requests), their algorithm builds size classes in a two-level approach. The logical first level segregates free blocks that are a power of two apart, i.e., a first-level index of $i$ is assigned to blocks whose sizes are in the range $\left[2^i, 2^{i+1} - 1\right]$. The second level splits each such interval linearly into an over all first-level intervals constant number $\mathcal{J}$ of equally-sized ranges. I.e., a second-level index $j$ is assigned to blocks whose sizes are in the range $\left[2^i + \frac{2^i}{\mathcal{J}} \cdot j, 2^i + \frac{2^i}{\mathcal{J}} \cdot (j+1) - 1\right]$. While providing equally good WCET bounds (in terms of operations) for allocation and deallocation operations as Half-Fit, due to finer grained size classes, TLSF's internal fragmentation can be arbitrarily decreased by adjusting the number of second-level ranges (per power-of-two size class), i.e., by increasing $\mathcal{J}$.

**Theorem 2.3** (Worst-Case Memory Consumption of TLSF)**.** *TLSF worst-case memory consumption can be bounded by* $\frac{\mathcal{J}+1}{\mathcal{J}} \cdot \mathcal{M}(m-2)$.

We can show this claim by using the same observations and arguments we used for Half-Fit in the proof of Theorem 2.2 paired with an adapted bound on the internal fragmentation as follows. In contrast to Half-Fit, TLSF's block sizes are not guaranteed to be a power of two. Instantiating TLSF with $\mathcal{J} = 1$ still leads to block sizes of a power of two and consequently a worst-case memory consumption equal to Half-Fit. In fact, Half-Fit is just the special case of instantiating TLSF with $\mathcal{J} = 1$. Increasing $\mathcal{J}$ *degenerates* TLSF to approximate the behavior of a best-fit sequential fit allocator. TLSF selects always the best fitting free block and once $\mathcal{J}$ is large enough, there exists a size class for every possible block size. For best-fit allocators, a worst-case memory consumption due to external fragmentation of $\mathcal{M}(m-2)$ can be shown [Rob77].

Furthermore, the worst-case internal fragmentation for TLSF can be bounded by a factor of $\frac{\mathcal{J}+1}{\mathcal{J}}$.

*Proof.* Let there be $n$ requests for blocks of size $2^{i_{max}} + 1$, so that the allocator always serves blocks of size $2^{i_{max}} + \frac{2^{i_{max}}}{\mathcal{J}}$. Putting peak memory need $\mathcal{M}$ and peak memory use $\mathcal{H}$ for this worst-case sequence of allocations into relation yields the desired result:

$$\begin{aligned}
\lim_{i_{max}\mapsto\infty} \frac{\mathcal{H}}{\mathcal{M}} &= \lim_{i_{max}\mapsto\infty} \frac{n\left(2^{i_{max}} + \frac{2^{i_{max}}}{\mathcal{J}}\right)}{n\left(2^{i_{max}} + 1\right)} \\
&= \lim_{i_{max}\mapsto\infty} \frac{n \cdot 2^{i_{max}}}{n \cdot 2^{i_{max}}} \cdot \frac{1 + \frac{1}{\mathcal{J}}}{1 + \frac{1}{2^{i_{max}}}} \\
&\overset{i_{max}\mapsto\infty}{\rightarrow} 1 + \frac{1}{\mathcal{J}} \\
&= \frac{\mathcal{J} + 1}{\mathcal{J}}
\end{aligned}$$

□

The worst-case memory consumption of these three constant-time allocators (binary buddy, `Half-Fit`, `TLSF`) may seem overly high at first glance. However, also for non-constant-time allocators, (de-)allocation sequences can be given which lead to equally high worst-case memory consumption. Even for linear time sequential fits that may take the time to always consider all currently managed memory blocks, worst-case memory consumption is close to the theoretical worst-case for any allocator (not prone to incomplete memory use): $\mathcal{M} \cdot m$ (amount of maximally live memory times the largest requestable size) [Rob77].

The worst-case memory consumption of a first-fit sequential fit allocator is $\frac{\mathcal{M}}{\ln 2} \cdot \sum_{i=1}^{m} \frac{1}{i}$ or about $\mathcal{M} \cdot \log_2 m$. A best-fit sequential fit allocator may even use up to $(\mathcal{M} - 4m + 11)(m - 2)$ words (or about $\mathcal{M} \cdot m$) in order to satisfy requests for $\mathcal{M}$ words [Rob77]. Despite that for real programs the best-fit allocator performs usually better and both perform nowhere near their respective worst cases [Rob77, WJNB95].

Unfortunately, drastic worst-case memory consumption is an inherent problem to dynamic memory allocation. We already pointed out that for any allocation algorithm there exists a worst-case (de-)allocation sequence for which memory consumption becomes overly high. In a hard real-time system where provable bounds on an application's memory consumption are required, we suggest to

rather show the absence of *bad* (de-)allocation sequences instead of constructing a *predictably bad allocator*[1].

Table 2.1 summarizes the worst-case performance properties of the (representatives of the most relevant classes of) dynamic memory allocators discussed in this chapter. In this table, we also list Doug Lea's memory allocator (DLMalloc) [Lea96] as a representative for *hybrid allocators*. Hybrid allocators combine several mechanisms—for example depending on the requested allocation size a different lookup mechanism is chosen—to improve on certain performance characteristics like memory consumption and execution times. We choose DLMalloc as a representative for this class of allocators as it is widely considered one of the best or even the best general purpose dynamic memory allocator when using average-case memory consumption and execution times to judge an allocator's performance.

## 2.3 Caches

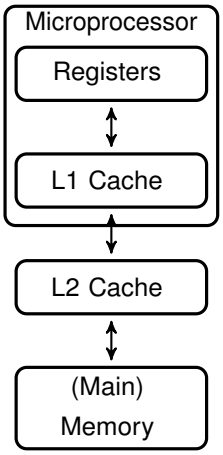
> Science never solves a problem without creating ten more.
>
> GEORGE BERNARD SHAW
> *Playwright (1856–1950)*

The storage of modern computing systems is normally not a single, uniform memory, but a hierarchical system composed of several (sub-)memories. Different memories or components differ—often strongly—in their respective properties. Registers, for example, have access latencies of just up to a single processor cycle, while accesses to the hard disk may require millions of cycles. A computer's main memory, the random access memory (RAM), may take hundreds of cycles to be accessed. Its cache memory can usually be accessed in just a few cycles, especially in case of the level 1 cache. Level 2 and level 3 caches have higher

[1] An allocator that provably performs close to its worst-case memory behavior which, in turn, is better than the worst-case behavior of the allocators discussed here, but much worse than the memory consumption of these for normal programs without (mostly theoretical) bad (de-)allocation sequences.

| Allocator | Allocation (temporal worst-cases) | Deallocation | Max. Memory Use $\mathcal{H}$ | References |
|---|---|---|---|---|
| Sequential Fit (First-Fit) | $O\left(\frac{\mathcal{H}}{2\mathcal{M}}\right)$ | $O(1)$ | $\frac{\mathcal{M}}{\ln 2}\sum_{i=1}^{m}\frac{1}{i}$ | [MRBC08] |
| Sequential Fit (Best-Fit) | $O\left(\frac{\mathcal{H}}{2\mathcal{M}}\right)$ | $O(1)$ | $\mathcal{M}\cdot(m-2)$ | [MRBC08] |
| Binary Buddy | $O\left(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right)\right)$ | $O\left(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right)\right)$ | $2\mathcal{M}(1+\lceil\log_2 m\rceil)$ | [MRBC08] [Knu97] Theorem 2.1 |
| DLMalloc | $O\left(\frac{\mathcal{H}}{\mathcal{M}}\right)$ | $O(1)$ | $\mathcal{M}\cdot m$ | [MRBC08] |
| `Half-Fit` | $O(1)$ | $O(1)$ | $\infty$, $2\mathcal{M}(1+\lceil\log_2 m\rceil)$ | [MRBC08] Theorem 2.2 |
| `TLSF` | $O(1)$ | $O(1)$ | $\frac{\mathcal{J}+1}{\mathcal{J}}\mathcal{M}(m-2)$ | [MRBC08] Theorem 2.3 |

Table 2.1: Worst-case properties of different dynamic memory allocators. $\mathcal{H}$ denotes the (actual) memory consumption for an application with maximum live memory $\mathcal{M}$. The maximal size of an allocation request is denoted by $m$.




| Memory Type | Size (in words) | Access Times (in cycles) |
|---|---|---|
| Register | 64 - 256 | ≤ 1 |
| L1 Cache | ≤ 8K | 1 - 2 |
| L2 Cache | ≤ 256K | 5 - 15 |
| (Main) Memory | ≤ 4G | 40 - 100 |

Figure 2.1: A typical memory hierarchy.

latencies, but are still significantly faster than accesses to the main memory. Given such significant performance advantages of some memory techniques over others, why is not the entire storage of a computer built-up of the fastest kind of memory? Production costs are certainly an obvious factor, i.e., faster memory is more expensive to produce. Hence, the ratio of faster to slower memory within a system depends on one's willingness or ability to pay for the improved performance. However, the more important factors are technological constraints. The fastest memories, the processor registers, are part of the central processing unit, and hence limited by the small space available on the processor die. Caches, for example, can only be built efficiently up to a certain size. If the size of the cache becomes too large, lookup operations can no longer be efficiently implemented.

Figure 2.1 depicts a typical memory hierarchy of modern computing systems. Such memory hierarchies aim to provide average access latencies close to the latencies of its fastest component as well as costs per bit and an overall size close to the component that can technically provide the largest amount of bits at the smallest costs per bit. The second goal is trivially fulfilled: as the cheapest memory, the system's hard disk, accounts for the largest part of the overall memory, the average costs per bit are strongly shifted towards the costs of a bit of hard

disk storage. When, in practice, the system manages to serve a better part of the memory accesses by faster components like (memory) caches, also the first goal, fast average access latencies, is achieved. Fortunately, computer programs tend to obey, to one degree or another, the so-called *principle of locality* or *locality of references*. This principle asserts that

1. a program distributes references it accesses during any interval of time non-uniformly over the pages it accesses, i.e., during any time interval, some references or pages are accessed more frequently than others,

2. access frequencies to given pages change only slowly over time, and

3. the correlation between immediate past and immediate future page accesses tends to be high, whereas this correlation decreases when the time between the accesses increases.

We can extract two, typically distinguished properties from this principle to which program memory accesses tend to adhere:

1. addresses are more likely to be accessed if they were accessed recently, which is typically denoted by *temporal locality*, and

2. addresses are more likely to be accessed if addresses near to them have already been accessed (recently), usually called *spatial locality*.

Denning and Schwartz explain this behavior abstraction by three observations [DS72]. First, a program's loop structures cluster references to given pages in short time intervals. Second, programmers tend to use divide-and-conquer approaches and consequently concentrate on small parts of larger problems for moderately long intervals. And third, programs run efficiently with only a subset of their pages in main memory. Hence, for programs to run efficiently, at a given time interval, only a very limited subset of memory needs to be accessible with low latencies. In consequence, when the memory hierarchy performs well in keeping the at any time interval relevant memory pages in the faster components, average access times tend towards the access times of the faster memories as desired. The principle of locality was first exploited in the concept of working sets used in the

late 1960s and early 1970s to prevent *thrashing*, the sudden collapse of throughput as the number of programs executed simultaneously rises. Intuitively, a working set is the smallest subset of the pages accessed by a given program that must reside in fast memory components in order that this program operates efficiently [DS72]. Today, ubiquitous caches exploit the principles of temporal and spatial locality in order to decrease memory latency. They *cache*, i.e., keep in faster memories, recently accessed addresses to exploit temporal locality. Furthermore, upon a cache miss, not only the requested but not cached memory block is loaded into the cache. A so-called cache line, a larger block of contiguous data, is transferred into the cache that also contains the immediate neighborhood of the requested address, thus exploiting spatial locality.

Technically, caches are usually implemented as follows. Let $\mathcal{M}$ be a memory component cached by a cache memory component $C$. To reduce management and traffic overhead of the cache itself and between $C$ and $\mathcal{M}$, respectively, $\mathcal{M}$ is logically partitioned into a set of memory blocks of size $b$. Choosing $b$ to be a power of two—as is common practice—allows for efficient lookup operations: the block offset, i.e., the memory block that contains a given address, is determined by the most significant bits of the memory address. When accessing a memory block, it has to be determined whether this block is contained in $C$, in which case the cache memory can serve this request at low latency. We call this a *cache hit*. When the requested block is not contained in $C$, it is transferred from $\mathcal{M}$ to $C$ which, in turn, can then satisfy the request. We call this a *cache miss*. The increased access latency caused by the access to $\mathcal{M}$ is called the *cache miss penalty*. To keep the latency of the cache low, determining whether a memory block is contained in the cache has to be sufficiently fast. It is therefore necessary to limit the number of cache lines at which a given memory block may reside. To this end, caches are partitioned into equally-sized cache sets, each holding $k$ (possibly empty) cache lines. This $k$, the size of a single cache set, measured in cache lines, is called the *associativity* of the cache. In order to further increase the efficiency of a cache lookup, $k$ is chosen such that the overall number of cache sets is a power of two. Consequently, the set number in which a memory block may reside is determined by the least significant bits of the block number (assuming the size of a memory block equals the size of a cache line). Obviously, cache performance, in terms of

latency, decreases when $k$ and/or $b$ is increased. Hence, a desired access latency restricts the overall cache size. Caches can therefore hold just a small fraction of the memory component succeeding them in the memory hierarchy. Consequently, most cache misses will not simply evict empty lines from the cache, but other, previously accessed memory blocks. What is evicted from cache is decided by a so-called *replacement policy*. Such a policy aims at minimizing cache misses by always evicting the "least useful" memory block currently contained in the respective cache set. Popular replacement strategies are:

1. Least-recently used (LRU) replacement policy that, upon a cache miss, evicts the least-recently-used cache line from the cache. This policy performs very well in practice and follows directly the principle of temporal locality: according to this principle, the probability for a cache line to be accessed decreases with the time since the last access to this line increasing. By always evicting the least-recently accessed cache line, this policy strives to always evict the cache line with the presumably smallest probability to be accessed again. LRU is the most predictable replacement policy [Rei08] and has been the target of most cache analyses [FMW97, FW99, GMM98, CPHL01, CP03]. The main advantage of LRU with respect to predictability is the fact that cache hits and cache misses are treated similarly: the currently accessed line will become the most recently used entry. Hence, even when a cache analysis cannot classify a given memory access as a hit or miss, useful information about the cache state after the access can be derived.

2. Pseudo least-recently-used (PLRU) is a tree-based approximation to LRU. PLRU is significantly cheaper to implement than LRU in terms of space (storage requirement) and time (update logic) and hence more common in existing hardware. The policy performs also well in practice [AZMM04]. However, PLRU may not always evict the least-recently-used block and, consequently, perform worse than LRU. This unwanted behavior is also detrimental to the predictability of the policy as shown in [Rei08].

3. First In, First Out (FIFO) always evicts the block that has been cached

for the longest time span. FIFO caches can thus be implemented as a simple queue, making them very cheap to implement and consequently very popular in practice. However, they may incur severe performance degradation compared to LRU caches, but tend to perform only slightly worse than LRU in many benchmarks [AZMM04]. Predictability of FIFO caches is low due to their non-uniform treatment of cache hits and misses: blocks may be evicted right after an access when they are at the first-in position [Rei08].

4. The most-recently-used (MRU) policy keeps status bits for each cache line, where a 1 indicates recent use. If a cache line needs to be evicted, the first, i.e., the cache line with the lowest index, set to 0 is evicted. This policy is the most unpredictable due to asymmetries from the update logic of the status bits: upon an access, the bit of the accessed line is set to 1; if this operation sets the last remaining 0 to 1, all other bits are set back to 0 again. [Rei08] shows that it is impossible to ever infer the precise cache contents for MRU caches.

With respect to write accesses, the following design choices are available. The *write policy* that determines when data is written back to $\mathcal{M}$ and can either be a *write-through* or *write-back* policy. A write-through policy updates $\mathcal{M}$ immediately on every write access and may incur additional, unnecessary data transfer if a cache line is modified several times. A write-back policy writes a cache line back to $\mathcal{M}$ upon its eviction from $\mathcal{C}$. Write-back caches require additional storage in the form of status bits to keep track of which cache lines have to be written back upon eviction. Write accesses to currently non-cached blocks, so-called *write-misses*, can also be treated in two ways. A write-miss may bypass the cache completely and be written directly in the subsequent memory (*no-write-allocate*). Given a write-back policy, it may, however, be advantageous to transfer the respective block to the cache and modify it there (*write-allocate*). And hence be able to decrease access times when more write accesses to this block occur in the recent future.

## 2.4 Cache Analysis in the Context of WCET Analysis

> If we can really understand the problem, the answer will come out of it, because the answer is not separate from the problem.
>
> JIDDU KRISHNAMURTI
> *in The Penguin Krishnamurti Reader (1970)*

A *cache analysis* aims to statically determine the cache behavior of a given program or program fragment on a set of possible inputs for an initially unknown cache state. As the cache behavior depends on both, the varying inputs and the varying initial cache states, such an analysis can in general only derive approximations of the cache behavior of the analyzed program. A concrete classification of all memory accesses into cache hits and cache misses cannot be achieved.

However, classifying all possible memory accesses as cache hits or cache misses is often not required. In compiler optimizations, cache analyses simply strive to give good estimates on the overall number of cache misses a given code fragment generates. In most cases, such estimates are sufficient to guide code optimizations for improving cache performance by making occurring sequences of memory accesses more cache-friendly.

Ghosh et al. as well as Chatterjee et al. observe that there are two main approaches to automatically improving data locality of loop-oriented programs: *loop nest restructuring*, i.e., loop transformations, and *data layout optimizations* [GMM98, CPHL01]. Loop transformations reorder access patterns to achieve better temporal as well as spatial locality by mechanism like iteration space tiling [Wol89], loop fusion [War84], and loop permutation (reordering of inner/outer loops, such that fewer cache lines are accessed) [IT88, MCT96]. While data layout transformations include techniques like intra- and inter-array padding, array merging, and blocking, i.e., *tiling of data structures* [RT98, RwT98, LW94]. Both approaches rely on a proper choice of parameters for the different techniques to be effective, i.e., choosing good tile sizes, inter-array pads, etc. A compiler that relies on heuristics to choose these parameters risks applying

transformations that severely degrade cache performance, instead of improving cache performance. In [CS00], Chatterjee and Sen provide empirical evidence that, when using heuristics to choose inter-array pad, most choices may be catastrophically bad for some programs. In consequence, Chatterjee et al. propose an approach to quantitatively determine the number of cache misses of a proposed transformation in order to guide the choice of parameters for transformation [CPHL01]. Their approach works without explicit simulation by using Presburger formulas to express the various kinds of cache misses as well as the state of the cache at the end of a loop nest. Ghosh et al. introduce *cache miss equations* as a means to precisely represent the cache misses of a loop nest and count cache misses in a given code fragment [GMM97]. In their model, a solution to a system of cache miss equations corresponds to a potential cache miss. Their approach enabled them to even find optimal solutions (where no solutions to the cache miss equations exist) to padding and blocking algorithms. In summary, these works provide strong evidence that precisely estimating the number of cache misses a given code fragment will produce is sufficient to guide automatic compiler optimizations. And that such estimates can be derived fast enough to be of practical use within a compiler.

When aiming to statically derive (bounds on) the worst-case execution time of programs, however, the requirements for a cache analysis are different. The overall number of possible cache misses is not a suitable abstraction anymore; when and where these misses may occur becomes of interest.

But why does a precise classification of accesses into cache hits and misses become important instead of just the number of hits and misses a sequence of memory accesses will produce? A *timing analysis* [Wil06, WEE+08] is normally done within the context of (hard) real-time applications. Assume a hard real-time system whose functionality is given by a set of tasks $\tau$. For a *timing validation* of this system for a given hardware platform, a *scheduling analysis* has to check whether all the timing constraints of the tasks contained in $\tau$ will always be met on that hardware. In order to do this, scheduling analysis requires bounds on the worst-case execution times of all tasks of $\tau$. The aim of a WCET analysis, i.e., a timing analysis tailored to derive upper bounds on a task's execution times, is to provide

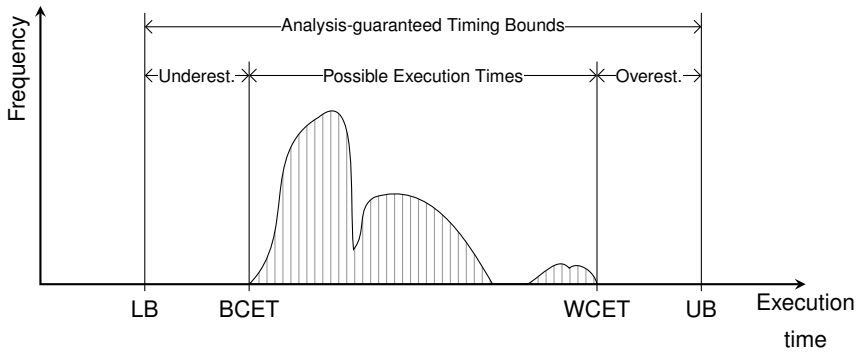



Figure 2.2: Hypothetical distribution of the execution times of an application. LB and UB denote the statically derivable lower bound and upper bound, respectively, on the application's execution times. BCET and WCET its actual best- and worst-case execution time.

these time bounds. These bounds need to be *safe* and *tight*. Otherwise, the results of the scheduling analysis might deem a non-schedulable system schedulable (in case the bound was not safe), possibly resulting in system failures when such a system is deployed. Or, in case the bound was not tight enough, the scheduler might deem a schedulable system not schedulable, leading to increased hardware costs (when switching to faster hardware in order to become schedulable) or simply non-deployment of the system.

Figure 2.2 shows the distribution of possible execution times for a hypothetical application. While the application and hence its execution times and frequencies thereof are hypothetical, we can still observe typical properties of real applications with which a timing analysis has to cope. Most importantly, there is often a high variability of execution times, depending on inputs to the application and the hardware state in which execution starts. *Timing accidents* are situations that drastically increase local timing variability. Consider for example a cache hit and a cache miss. While the former may take just 1 to 2 cycles, a miss may require 6 to 66 clock cycles to serve the accessed data [HP96]. Hence, cache performance is one of the major factors contributing to this variability. Additionally, many hardware

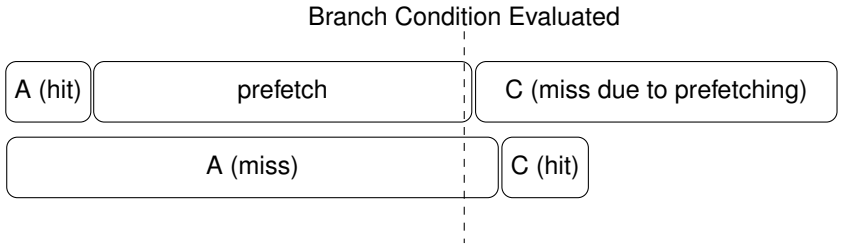



Figure 2.3: Example for a Speculation Anomaly from [RWT+06]. In the first case (first row), the access to *A* is a cache hit and while the condition for a subsequent branch is not yet available, the processor mispredicts the branch and an unnecessarily prefetched instruction evicts *C* from the cache. Thus, the subsequent access to *C* results in a cache miss. In the second case, while *A* is served after a cache miss, this longer execution of the access to *A* prevents a misprediction of the branch. The subsequent access to *C* results in this case in a cache hit. Which in turn results in a shorter global execution time compared to the first scenario.

architectures exhibit so-called timing anomalies, i.e., situations where locally faster execution leads to an increased global execution time [LS99, RWT+06].

[RWT+06] provides an easy-to-follow example of a *speculation anomaly* where a mispredicted branch after a cache hit leads to a longer execution time than the one resulting after a cache miss. Figure 2.3 shows this example in more detail. This example serves to illustrate that knowing where cache hits or misses occur is crucial when striving for safe and also tight bounds on a program's execution times.

While required to be more precise, fortunately, a cache analysis in the context of timing analysis is allowed to take significantly longer than the previously discussed cache analyses used in compilers to guide optimizing program transformations.

In Subsections 2.4.1 and 2.4.2, we describe two cache analyses tailored to provide precise information about an application's cache performance for a timing analysis such that safe and tight timing bounds are derivable. These analyses also

represent the current state-of-the-art. At the end of each subsection, we discuss why these analyses yield only very imprecise results for programs using dynamic memory allocation and employing standard, non-cache-aware allocators.

### 2.4.1 Ferdinand's LRU Cache Analyses

Intuitively, when aiming to classify each memory access as either a cache hit or a cache miss, one needs to know what is definitely in the cache at the time of the access and what is definitely not in the cache. The former allows for a classification of accesses as cache hits, the latter for a classification as cache misses. Ferdinand et al. propose a static program analysis based on abstract interpretation [CC77] that computes this information [Fer97, FW99]. I.e. per program point, an over- as well as an under-approximation of the cache contents are computed. When the memory block referenced at a program point is contained in the under-approximation, a cache hit can be guaranteed. When the referenced block is not contained in the over-approximation, a cache miss can be safely predicted. In the remainder of this subsection, we will summarize their analysis and discuss its applicability to programs using dynamic memory allocation.

Consider a $k$-way set associative cache architecture with $n$ cache sets, a cache-line and memory-block size of $b$ bytes, and an overall capacity $c = n \cdot k \cdot b$ bytes. We assume the utilized replacement strategy to be LRU. Note that fully associative and direct mapped caches are special cases of such an architecture where $n = 1$ and $n = \frac{c}{b}$, respectively. Cache and memory can be described by a set of sequences of cache lines $C = \{f_1, \ldots, f_n\}$, where $f_i = \langle l_1, \ldots, l_k \rangle$, and a set of memory blocks $M = \{m_1, m_2, \ldots\}$. Each $f_i$ models the contents of a single cache set. The ordering of the sequence of cache lines $l_j$ for each cache set $f_i$ is such that $j$ represents the *age* of the entry stored in line $l_j$. Two functions $addr : M \to \mathbb{N}$ and $set : M \to C$ map memory blocks to their addresses and to the cache sets in which they are to be placed when being cached. The latter, *set*, can be defined in terms of *addr* and the cache parameters $c$, $b$, and $k$.

**Definition 2.1** (Cache Set Mapping)**.** *Let $addr : M \to \mathbb{N}$ constitute a mapping of memory blocks to their memory addresses, c the capacity of the cache, b the size of a cache line, and k the cache's associativity. A function $set : M \to C$ mapping memory blocks to the cache set in which they are to be stored is then given by:*

$$set(m) = f_{(addr(m) \bmod \frac{c}{b \cdot k})+1}, m \in M$$

We introduce a new element $I$ to express the absence of any memory block in a cache line and define a *concrete set state* as follows.

**Definition 2.2** (Concrete Set State)**.** *A concrete set state for a cache set $f_i = \langle l_1, \ldots, l_k \rangle$ is a function*

$$s : \bigcup_{1}^{k} l_j \to M \cup \{I\}$$

*such that s is injective except on I, i.e., memory blocks are stored at most once. Furthermore, let $S$ denote the set of all concrete set states.*

Note that we have now the means to describe the relative age of a cached memory block according to the LRU replacement strategy, as $s(l_x) = m \wedge m \neq I$ entails that memory block $m$ has relative age $x$. Furthermore, we can use $S$ to define concrete cache states.

**Definition 2.3** (Concrete Cache State)**.** *A concrete cache state is a function $cs : C \to S$, where*

$$\forall f_i = \langle l_1, \ldots, l_k \rangle \in C \,.\, \forall l_j \,.\, (cs(f_i))(l_j) \neq I \Rightarrow set((cs(f_i))(l_j)) = f_i$$

*Furthermore, let CS denote the set of all concrete cache states.*

To model the effects of accesses to the memory on the current cache state, we introduce two update functions: a *set update function* $\mathcal{U}_S : S \times M \to S$ and a *cache update* function $\mathcal{U}_{CS} : CS \times M \to CS$ to describe the new set state and the new cache state, respectively, for a given set or cache state and a referenced memory block. Let $[x \mapsto y]$ denote a function that maps $x$ to $y$ and $f\,[x \mapsto y]$ denote a function that maps $x$ to $y$ and $\forall z \neq x : z \mapsto f(z)$. Then, $\mathcal{U}_S$ and $\mathcal{U}_{CS}$ can be defined as follows.

**Definition 2.4** (Set and Cache Update)**.**

$$\mathcal{U}_S(s,m) = \begin{cases} [l_1 \mapsto m\ , & \\ l_i \mapsto s(l_{i-1}) \mid i = 2 \ldots h & \\ l_i \mapsto s(l_i) \mid i = h+1 \ldots k] & : \textit{if } \exists l_h\ .\ s(l_h) = m \\ [l_1 \mapsto m, & \\ l_i \mapsto s(l_{i-1}) \mid i = 2 \ldots k] & : \textit{otherwise} \end{cases}$$

*and*

$$\mathcal{U}_{CS}(cs,m) = cs[set(m) \mapsto \mathcal{U}_S\ (cs\,(set(m))\,,m)]$$

We assume the program to be analyzed to be given as its control-flow graph $(V, E \subseteq V \times V, start \in V)$ with its nodes $V$ representing the program's basic blocks. I.e., branch- and halting-point-free sequences of instructions where control flow enters at the first instruction of a sequence and always exits at the last instruction. We also assume that there is a mapping of these basic blocks to the sequences of memory blocks that are accessed within this block: $\mathcal{L} : V \to M^*$. We can easily extend the update function $\mathcal{U}_{CS}$ to model the effects of sequences of memory accesses on a given cache state $c$:

$$\mathcal{U}_{CS}\,(c, \langle m_1, \ldots, m_e \rangle) = \mathcal{U}_{CS}\,(\ldots \mathcal{U}_{CS}\,(c, m_1) \ldots, m_e)$$

And consequently, we can model the effects of a program execution, i.e., a path $\pi = \pi_1, \ldots, \pi_p$ in the control-flow graph on an initial cache state $c_I$ by:

$$\mathcal{U}_{CS}\left(c_I, \left\langle \mathcal{L}(\pi_1), \ldots, \mathcal{L}(\pi_p) \right\rangle\right)$$

In general, there exist several, possibly infinitely many paths from the program's start node to the other nodes. As a cache analysis needs to derive properties that hold for all possible program executions, we continue by setting up an abstract domain describing the so-far developed concrete domain. The analyses described by Ferdinand et al. will then use these abstract domains.

**Definition 2.5** (Abstract Set State)**.** *An abstract set state for a cache set* $f_i = \langle l_1, \ldots, l_k \rangle$ *is a function*

$$\widehat{s} : \bigcup_{1}^{k} l_j \to 2^M$$

*mapping cache lines of a given cache set* $f_i$ *to sets of memory blocks, such that*

$$\forall l_a, l_b \in f_i \,.\, \forall m \in M \,.\, m \in (\widehat{s}(l_a) \cap \widehat{s}(l_b)) \Rightarrow l_a = l_b$$

*Furthermore, let* $\widehat{S}$ *denote the set of all abstract set states.*

**Definition 2.6** (Abstract Cache State)**.** *An abstract cache state is a function* $\widehat{cs} : C \to \widehat{S}$*, where*

$$\forall f_i = \langle l_1, \ldots, l_k \rangle \in C \,.\, \forall l_j \,.\, \forall m \in M \,.\, m \in (\widehat{cs}(f_i))(l_j) \Rightarrow set(m) = f_i$$

*Furthermore, let* $\widehat{CS}$ *denote the set of all abstract cache states.*

The aim of Ferdinand's analyses is to classify memory references as *always hit*, *always miss* or *not classified*. In order to do so, two analyses are performed. A *must analysis* to determine which memory blocks are definitely in the cache in order classify memory references as *always hit*. As well as a *may analysis* to determine for each program point the set of memory blocks that may be in the cache. This information can be used to classify references to memory blocks that are not contained in the computed sets as *always miss*.

The definitions of the abstract update functions depend on what the analysis aims at: the sets of memory blocks that must be or that may be in the cache. The same holds true for a join function $\widehat{\mathcal{J}} : \widehat{C} \times \widehat{C} \mapsto \widehat{C}$ that combines two abstract cache states. In the following paragraphs, we will briefly summarize both analyses and give their respective definitions for appropriate update and join functions.

#### Ferdinand's Must Cache Analysis

To determine the set of definitely cached memory blocks, abstract cache and set states are used with the invariant that the position of a memory block in the abstract set state $\widehat{s}$ (or in other words its relative age) is an upper bound on the positions of this block in all concrete set states that $\widehat{s}$ describes. Accordingly, the

set of concrete cache and set states described by abstract states is given by the following concretization functions.

**Definition 2.7** (Concretization Functions for LRU Must Cache Analysis)**.**

$$\gamma^{\cap}_{\widehat{CS}}\left(\widehat{cs}\right) = \left\{cs \in CS \mid \forall f_i \,.\, cs\left(f_i\right) \in \gamma^{\cap}_{\widehat{S}}\left(\widehat{cs}\left(f_i\right)\right)\right\}$$

*where*

$$\gamma^{\cap}_{\widehat{S}}\left(\widehat{s}\right) = \{s \in S \mid \forall a \in [k] \,.\, \forall m \in \widehat{s}(l_a) \;.\; \exists b \,.\, s\left(l_b\right) = m \wedge b \leq a\}$$

We can define appropriate abstract update formulæ to model the effect of referencing memory addresses on abstract cache states as follows.

**Definition 2.8** (Abstract Set and Cache Update for LRU Must Cache Analysis)**.**

$$\widehat{\mathcal{U}}^{\cap}_{\widehat{S}}(\widehat{s}, m) = \begin{cases} [l_1 \mapsto \{m\}\,, & \\ l_i \mapsto \widehat{s}(l_{i-1}) \mid i = 2 \ldots h-1 & \\ l_h \mapsto \widehat{s}(l_{h-1}) \cup \left(\widehat{s}\left(l_h\right) \backslash \{m\}\right) & \\ l_i \mapsto \widehat{s}(l_i) \mid i = h+1 \ldots k] & : \; \textit{if } \exists l_h \,.\, m \in \widehat{s}(l_h) \\ [l_1 \mapsto \{m\}, & \\ l_i \mapsto \widehat{s}(l_{i-1}) \mid i = 2 \ldots k] & : \; \textit{otherwise} \end{cases}$$

*and*

$$\widehat{\mathcal{U}}^{\cap}_{\widehat{CS}}(\widehat{cs}, m) = \widehat{cs}\left[set(m) \mapsto \widehat{\mathcal{U}}^{\cap}_{\widehat{S}}\left(\widehat{cs}\left(set(m)\right), m\right)\right]$$

Furthermore, we need a way to join several abstract cache states as, in general, more than a single path may lead to a given program point. Definition 2.10 defines such a join function that computes a least upper bound on two abstract cache states. Joining more than two abstract states can be done by repeatedly applying this function. Definition 2.9 defines the join function on abstract set states that is used in Definition 2.10.

**Definition 2.9** (Join Function for Abstract Set States for LRU Must Cache Analysis)**.** *Let* $\widehat{\mathcal{J}}^{\cap}_{\widehat{S}} : \widehat{S} \times \widehat{S} \to \widehat{S}$ *be an associative function:*

$$\widehat{\mathcal{J}}^{\cap}_{\widehat{S}}(\widehat{s}_1, \widehat{s}_2) = \widehat{s}$$

*such that*

$$\forall l_x \,.\, \widehat{s}(l_x) = \{m | \exists l_a, l_b \,.\, m \in \widehat{s_1}(l_a) \wedge m \in \widehat{s_2}(l_b) \wedge x = \max\{a, b\}\}$$

**Definition 2.10** (Join Function for Abstract Cache States for LRU Must Cache Analysis)**.** *Let* $\widehat{\mathcal{J}}^{\cap}_{\widehat{CS}} : \widehat{CS} \times \widehat{CS} \to \widehat{CS}$ *be an associative function:*

$$\widehat{\mathcal{J}}^{\cap}_{\widehat{CS}}(\widehat{cs}_1, \widehat{cs}_2) = \left[ f_i \mapsto \widehat{\mathcal{J}}^{\cap}_{\widehat{S}}(\widehat{cs}_1(f_i), \widehat{cs}_2(f_i)) \middle| i \in [k] \right]$$

### Ferdinand's May Cache Analysis

The May Cache Analysis, in contrast, computes the sets of memory blocks that may be present in the cache at the different program points by deriving abstract cache states in which the position of a memory block is always a lower bound on the positions of this block in all concrete set states that these abstract states describe. With this in mind, concrete and abstract states shall be connected via the following concretization functions.

**Definition 2.11** (Concretization Functions Connecting Abstract Set and Cache States of the LRU May Cache Analysis to Concrete Set and Cache States)**.**

$$\gamma^{\cup}_{\widehat{CS}}(\widehat{cs}) = \left\{ cs | \forall f_i \,.\, cs(f_i) \in \gamma^{\cup}_{\widehat{S}}(\widehat{cs}(f_i)) \right\}$$

$$\gamma^{\cup}_{\widehat{S}}(\widehat{s}) = \{s | \forall a \in [k] \,.\, s(l_a) \neq I \,.\, \exists b \,.\, s(l_a) \in \widehat{s}(l_b) \wedge b \leq a\}$$

Definition 2.12 gives appropriate update functions for a may cache analysis assuming an LRU replacement policy. The definitions of the join functions for abstract set and cache states used in the may cache analysis are given in Definition 2.13

**Definition 2.12** (Abstract Set and Cache Update for LRU May Cache Analysis)**.**

$$\widehat{\mathcal{U}}^{\cup}_{\widehat{S}}(\widehat{s}, m) = \begin{cases} [l_1 \mapsto \{m\}\ , & \\ l_i \mapsto \widehat{s}(l_{i-1})\ \mid i = 2 \ldots h & \\ l_{h+1} \mapsto \widehat{s}(l_{h+1}) \cup (\widehat{s}(l_h) \backslash \{m\}) & \\ l_i \mapsto \widehat{s}(l_i)\ \mid i = h+2 \ldots k] & : \textit{ if } \exists l_h\ .\ m \in \widehat{s}(l_h) \\ [l_1 \mapsto \{m\}, & \\ l_i \mapsto \widehat{s}(l_{i-1})\ \mid i = 2 \ldots k] & : \textit{ otherwise} \end{cases}$$

*and*

$$\widehat{\mathcal{U}}^{\cup}_{\widehat{CS}}(\widehat{cs}, m) = \widehat{cs}\left[set(m) \mapsto \widehat{\mathcal{U}}^{\cup}_{\widehat{S}}\left(\widehat{cs}\left(set(m)\right), m\right)\right]$$

**Definition 2.13** (Abstract Set and Cache State Join Functions for LRU May Cache Analysis)**.** *Let $\widehat{\mathcal{J}}^{\cup}_{\widehat{S}} : \widehat{S} \times \widehat{S} \to \widehat{S}$ be an associative function to compute the least upper bound of two abstract set states, where*

$$\widehat{\mathcal{J}}^{\cup}_{\widehat{S}}\left(\widehat{s}_1, \widehat{s}_2\right) = \widehat{s}$$

*such that*

$$\begin{aligned} \widehat{s}(l_x) \quad &= \quad \{m | \exists l_a, l_b\ .\ m \in \widehat{s}_1\left(l_a\right) \wedge m \in \widehat{s}_2\left(l_b\right) \wedge x = \min\{a, b\}\} \\ &\cup \quad \{m | m \in \widehat{s}_1\left(l_x\right) \wedge \forall l_a\ .\ m \notin \widehat{s}_2\left(l_a\right)\} \\ &\cup \quad \{m | m \in \widehat{s}_2\left(l_x\right) \wedge \forall l_a\ .\ m \notin \widehat{s}_1\left(l_a\right)\} \end{aligned}$$

*Furthermore, let $\widehat{\mathcal{J}}^{\cup}_{\widehat{CS}} : \widehat{CS} \times \widehat{CS} \to \widehat{CS}$ be an associative function to compute the least upper bound of two abstract cache states:*

$$\widehat{\mathcal{J}}^{\cup}_{\widehat{CS}}\left(\widehat{cs}_1, \widehat{cs}_2\right) = \left[ f_i \mapsto \widehat{\mathcal{J}}^{\cup}_{\widehat{S}}\left(\widehat{cs}_1(f_i), \widehat{cs}_2(f_i)\right) \middle| i \in [k] \right]$$

So far, this models only read accesses to the cache. But while originally aimed to analyze just the behavior of instruction caches where only reads occur, Ferdinand's approach can also be applied to data caches and combined instruction/data caches [FMWA96, FW99]. Assume a program in a language with only global functions, like C, that uses only scalar variable types. The addresses within a function's stack frame of local variables, including function parameters, can in such a scenario be statically computed by static stack level simulation [WM95]. The address of the

stack frame of an instance of a function, i.e., an active call to this function, can be computed as an offset to the caller function's stack frame. This can also be done statically. To do so, however, we need to extend our function $\mathcal{L}$ that maps the nodes of the program's control-flow graph $V$ to a list of accessed memory blocks by a set of possible absolute stack addresses (for these nodes). Furthermore, let $\mathcal{H} : V \to \mathbb{N}_0$ be a function mapping control-flow graph nodes to their relative stack frame offset, i.e., their stack height. Similarly to $\mathcal{L}$, the update and join functions need to be extended to operate on pairs of abstract cache states and sets of possible stack frame addresses. In [FMWA96], the authors extend $\mathcal{L}$, $\widehat{\mathcal{U}}$, and $\widehat{\mathcal{J}}$ as follows.

**Definition 2.14** (Extended Mapping Function from Control-Flow Graph Nodes to Memory Blocks, Extended Update Function, and Extended Join Functions)**.** *Let $\mathcal{L}' : V \times 2^{\mathbb{N}_0} \to S^*$ be a function mapping pairs of nodes and absolute stack frame addresses to referenced memory blocks. Furthermore, let the adjusted cache-set update function $\widehat{\mathcal{U}}'_{\widehat{CS}} : \widehat{CS} \times M \times 2^{\mathbb{N}_0} \to \widehat{CS} \times 2^{\mathbb{N}_0}$ be defined as*

$$\widehat{\mathcal{U}}'_{\widehat{CS}}(\widehat{c}, m, \{h_1, \ldots, h_x\}) = \begin{cases} \left(\widehat{\mathcal{U}}'_{\widehat{CS}}(\widehat{c}, m, \{h_1 + \mathcal{H}(n), \ldots, h_x + \mathcal{H}(n)\})\right) & \text{for a call node } n \\ \left(\widehat{\mathcal{U}}'_{\widehat{CS}}(\widehat{c}, m, \{h_1, \ldots, h_x\})\right) & \text{otherwise} \end{cases}$$

*where $\left(\widehat{\mathcal{U}}'_{\widehat{CS}}(\widehat{c}, m, \{h_1, \ldots, h_x\})\right)$ is the obvious extension of $\widehat{\mathcal{U}}^{\cap}_{\widehat{CS}}(\widehat{cs}, m)$ or $\widehat{\mathcal{U}}^{\cup}_{\widehat{CS}}(\widehat{cs}, m)$, depending on whether a must- or a may-analysis is to be implemented. Finally, $\widehat{\mathcal{J}}'_{\widehat{CS}} : \left(\widehat{CS} \times 2^{\mathbb{N}_0}\right) \times \left(\widehat{CS} \times 2^{\mathbb{N}_0}\right) \to \left(\widehat{CS} \times 2^{\mathbb{N}_0}\right)$ is defined as*

$$\widehat{\mathcal{J}}'_{\widehat{CS}}((\widehat{cs}_1, H_1), (\widehat{cs}_2, H_2)) = \left(\widehat{\mathcal{J}}^{\cap\backslash\cup}_{\widehat{CS}}, H_1 \cup H_2\right)$$

*where, again, depending on whether a must- or a may-analysis is to be implemented, $\widehat{\mathcal{J}}^{\cap\backslash\cup}_{\widehat{CS}}$ resolves to $\widehat{\mathcal{J}}^{\cap}_{\widehat{CS}}$ or $\widehat{\mathcal{J}}^{\cup}_{\widehat{CS}}$.*

For programs with recursive procedures, the number of stack frame addresses may grow indefinitely and thus prevent the analysis from terminating. However, a suitable widening operator [CC76] can be used to ensure termination [FMWA96].

Lastly, the analysis has to correctly model the influence of writes on the cache state. Besides different cache behavior induced by the two write policies (write-through and write-back), upon a *write miss*, two cache designs are common. The block to write is first loaded into the cache (*write-allocate*) or the write changes only the main memory (*no-write-allocate*). For a cache architecture employing a write-through/write-allocate policy, writes can simply be treated as reads and no adjustments need to be made. For an architecture with a write-through/no-write-allocate policy, we have to distinguish two cases. In case the written memory block is currently contained in the (abstract or concrete) cache state, we can again treat this access as a read. In case the memory block is not currently cached, the cache state does not change and the update function is just the identity function.

For write-back architectures, more adjustments are required. Write-back caches write modified cache lines back to main memory at the time the line is evicted from the cache. Hence, the analysis needs to track whether memory blocks were modified while cached, in order to determine whether evicting a block will cause a write back to memory. To keep track of modified cache lines, the concrete and abstract set state function can easily be extended by a *dirty bit* indicating whether a line was modified (i.e., is **d**irty) or not (is **c**lean).

**Definition 2.15** (Concrete and Abstract Set States For Write-Back Caches)**.** *Given a write-back cache architecture, a concrete set state for a cache set $f_i = \langle l_1, \ldots, l_k \rangle$ is a function*

$$s : \bigcup_{1}^{k} l_j \to \{d, c\} \times (M \cup \{I\})$$

*An abstract set state $f_i = \langle l_1, \ldots, l_k \rangle$ is a function*

$$\widehat{s} : \bigcup_{1}^{k} l_j \to 2^{\{d,c\} \times M}$$

Furthermore, update functions are required to distinguish between read and write accesses and set the dirty bit on writes to a cache line.

How can the extended analysis determine whether write-backs (may) occur? Consider a situation as depicted in Figure 2.4, where a memory block $m$ is accessed and the analysis computed pairs of may and must information that are

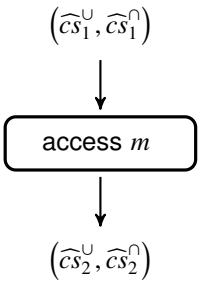



Figure 2.4: An access to a memory block $m$ with may- and must-cache states $\left(\widehat{cs}_1^{\cup}, \widehat{cs}_1^{\cap}\right)$ and $\left(\widehat{cs}_2^{\cup}, \widehat{cs}_2^{\cap}\right)$ before and after the access.

valid immediately before (denoted $\left(\widehat{cs}_1^{\cup}, \widehat{cs}_1^{\cap}\right)$) and after (denoted $\left(\widehat{cs}_2^{\cup}, \widehat{cs}_2^{\cap}\right)$) this access. Let $l_m$ be the cache line where $m$ is stored in $\widehat{cs}_2^{\cup}$. Consequently, $\widehat{cs}_1^{\cup}(l_m)$ contains all blocks that may have been evicted by accessing $m$. The analysis can then distinguish three cases.

C1 $\widehat{cs}_1^{\cup}(l_m)$ does not contain any modified lines, i.e., $\left\{x \middle| (d,x) \in \widehat{cs}_1^{\cup}(l_m)\right\} = \emptyset$. In this case, the analysis can guarantee the absence of a write-back.

C2 There exists a memory block $m_d$ in $\left\{x \middle| (d,x) \in \widehat{cs}_1^{\cup}(l_m)\right\}$ such that $m_d$ is an always hit in $\widehat{cs}_1^{\cap}$ and an always miss in $\widehat{cs}_2^{\cup}$. In this case, the analysis can guarantee the occurrence of a write-back.

C3 Otherwise, the analysis has to consider a possible write-back.

#### Concluding Remarks and Discussion of Applicability on Programs Using Dynamic Memory Allocation

Ferdinand's cache analyses can be considered current state-of-the-art and have been implemented in industrial strength tools. Namely, the aiT WCET analyzers [HFG04] that aim to statically compute tight bounds on the WCET of tasks in real-time systems. Those analyzers take into account the intrinsic cache behavior of programs in order to yield tighter WCET bounds. The proposed must- and may-cache analyses discussed in this chapter have since been proved to be precise

enough to enable a subsequent timing analysis to derive tight bounds on the WCET of real-life industrial code as well as academical benchmark programs [San04, Seh05, SPH+05, FHW+08, Tan06].

But how useful can the results of such cache analyses be in the presence of dynamic memory allocation? Consider an access to a dynamically allocated object. Statically, we cannot predict its memory address as dynamic memory allocators only provide guarantees about the sizes, not the locations of the memory blocks they return upon allocation requests. Hence, the best we can do is to define abstract update functions to extend those given in Definitions 2.8, 2.12, and 2.14 that account for a single, unknown access to every cache set. Definition 2.16 defines appropriate update functions that can be used when referencing a dynamically allocated object with unknown memory address mapping. Note that, for the sake of readability, we simplified the function signature and omitted the accessed (abstract) memory block as well as the potential stack heights as we cannot extract any information from them. For other memory blocks (instructions, statically allocated objects), we further rely on Ferdinand's update functions.

**Definition 2.16** (Abstract Set and Cache Update for LRU Must and May Cache Analysis When Referencing Unknown Memory Addresses)**.**

$$\widehat{\mathcal{U}}_{\widehat{S}}^{\mathcal{D}^\cap}(\widehat{s}) = \begin{cases} [l_1 \mapsto \{\}, \\ l_i \mapsto \widehat{s}(l_{i-1}) \mid i = 2 \ldots k] \end{cases}$$

$$\widehat{\mathcal{U}}_{\widehat{S}}^{\mathcal{D}^\cup}(\widehat{s}) = \begin{cases} [l_1 \mapsto M, \\ l_i \mapsto \{\} \mid i = 2 \ldots k] \end{cases}$$

*and*

$$\widehat{\mathcal{U}}_{\widehat{CS}}^{\mathcal{D}^\cap}(\widehat{cs}) = \left[ f_i \mapsto \widehat{\mathcal{U}}_{\widehat{S}}^{\mathcal{D}^\cap}(f_i) \mid i = 1 \ldots k \right]$$

$$\widehat{\mathcal{U}}_{\widehat{CS}}^{\mathcal{D}^\cup}(\widehat{cs}) = \left[ f_i \mapsto \widehat{\mathcal{U}}_{\widehat{S}}^{\mathcal{D}^\cup}(f_i) \mid i = 1 \ldots k \right]$$

After an access to an unknown memory location, the must analysis retains no information about the youngest entries of the different cache sets, but information about other entries may be retained. The effect on a may-cache is, however, more drastic. Every memory block may potentially be contained in the cache, all information about the current cache state is de facto lost. We observe that this

does not only prevent the analysis from predicting cache hits or cache misses for dynamically allocated objects, we also lose otherwise derived information about the cache state. Potentially, this may of course also prevent the analysis from predicting hits or misses for other, statically allocated objects.

We also need to account for the cache influence of the allocation and deallocation routines themselves, i.e., calls to `malloc` and `free`. If we assume the allocator to be a black box, what guarantees about the behavior of an allocator can be safely given? Realistically, we assume that only upper bounds on the maximum number of pairwise different cache lines (per cache set) accessed during an allocation and a deallocation request, respectively, may be provided. Other properties that might be stated (like bounds on execution times) are not relevant for a static cache analysis.

In Definition 2.17, we provide appropriate update functions to model the influence of invocations of the allocator on the abstract cache state. As with accesses to unknown memory blocks, invoking the dynamic memory allocator leads to a complete loss of information in case of a may cache analysis. The information loss in case of a must analysis depends on the allocator, i.e., whether $L_{\mathcal{A}}$ or $L_{\mathcal{F}}$ are larger than the associativity of the cache. When $L_{\mathcal{A}} \geq k$, an allocation also leads to a complete loss of information regarding the current (must) cache state. Unfortunately, for standard allocators, we must almost always assume $L_{\mathcal{A}} \geq k$ due to their search policies. The same pessimistic assumption must often be made for $L_{\mathcal{F}}$ either due to unpredictable merging operations or the allocator keeping a sorted list of free blocks which entails a sorted insertion.

In summary, we observe that in the presence of dynamic memory allocation, a cache analysis as discussed in this chapter is quickly rendered useless. This is, however, not a problem of precision of the analysis, but caused by uncertainties inherent to the allocator itself. When a statically unknown memory block is accessed, any block may be consequently copied to the cache. Hence, any conservative may cache analysis must revert to the least precise cache state, where *any memory block may be in youngest position in any cache set*.

**Definition 2.17** (Abstract Set and Cache Update for LRU Must and May Cache Analysis For Invocations of the Dynamic Allocator)**.** *Let $L_{\mathcal{A}}$ and $L_{\mathcal{F}}$ be the maximum number of pairwise different cache lines (per cache set) accessed during an allocation and a deallocation request, respectively. Furthermore, let*

$$\widehat{\mathcal{U}}^{\mathcal{A}^\cap}_{\widehat{S}}(\widehat{s}) = \begin{cases} [l_i \mapsto \{\} \mid i = 1 \ldots L_{\mathcal{A}}, \\ \ l_i \mapsto \widehat{s}(l_{i-L_{\mathcal{A}}}) \mid i = L_{\mathcal{A}} + 1 \ldots k] \end{cases}$$

$$\widehat{\mathcal{U}}^{\mathcal{A}^\cup}_{\widehat{S}}(\widehat{s}) = \begin{cases} [l_1 \mapsto M, \\ \ l_i \mapsto \{\} \mid i = 2 \ldots k] \end{cases}$$

$$\widehat{\mathcal{U}}^{\mathcal{F}^\cap}_{\widehat{S}}(\widehat{s}) = \begin{cases} [l_i \mapsto \{\} \mid i = 1 \ldots L_{\mathcal{F}}, \\ \ l_i \mapsto \widehat{s}\left(l_{i-L_{\mathcal{F}}}\right) \mid i = L_{\mathcal{F}} + 1 \ldots k] \end{cases}$$

$$\widehat{\mathcal{U}}^{\mathcal{F}^\cup}_{\widehat{S}}(\widehat{s}) = \begin{cases} [l_1 \mapsto M, \\ \ l_i \mapsto \{\} \mid i = 2 \ldots k] \end{cases}$$

*and*

$$\widehat{\mathcal{U}}^{\mathcal{A}^\cap}_{\widehat{CS}}(\widehat{cs}) = \left[f_i \mapsto \widehat{\mathcal{U}}^{\mathcal{A}^\cap}_{\widehat{S}}(f_i) \mid i = 1 \ldots k\right]$$

$$\widehat{\mathcal{U}}^{\mathcal{A}^\cup}_{\widehat{CS}}(\widehat{cs}) = \left[f_i \mapsto \widehat{\mathcal{U}}^{\mathcal{A}^\cup}_{\widehat{S}}(f_i) \mid i = 1 \ldots k\right]$$

$$\widehat{\mathcal{U}}^{\mathcal{F}^\cap}_{\widehat{CS}}(\widehat{cs}) = \left[f_i \mapsto \widehat{\mathcal{U}}^{\mathcal{F}^\cap}_{\widehat{S}}(f_i) \mid i = 1 \ldots k\right]$$

$$\widehat{\mathcal{U}}^{\mathcal{F}^\cup}_{\widehat{CS}}(\widehat{cs}) = \left[f_i \mapsto \widehat{\mathcal{U}}^{\mathcal{F}^\cup}_{\widehat{S}}(f_i) \mid i = 1 \ldots k\right]$$

*where $\widehat{\mathcal{U}}^{\mathcal{A}^\cap}_{\widehat{CS}}$ and $\widehat{\mathcal{U}}^{\mathcal{F}^\cap}_{\widehat{CS}}$ model the effects of an allocation and deallocation request, respectively, on the abstract may cache (state); $\widehat{\mathcal{U}}^{\mathcal{A}^\cap}_{\widehat{CS}}$ and $\widehat{\mathcal{U}}^{\mathcal{F}^\cap}_{\widehat{CS}}$ the corresponding effects on a must cache.*

### 2.4.2 Relational Cache Analysis

There are certain drawbacks to state-of-the-art cache analyses that use memory blocks as abstract cache elements—like Ferdinand's cache analyses described in the previous subsection. In particular, the precision of such an analysis strongly depends on the precision of a previous value or address analysis, i.e., how precisely referenced addresses can be determined.

In [HG12], Hahn and Grund make the following observations:

- A cache analysis cannot predict hits for accesses with imprecisely determined addresses when memory blocks are used as abstract cache elements.
- A cache analysis suffers from excessive information loss in the presence of accesses to imprecisely determined addresses when memory blocks are used as abstract cache elements.
- State-of-the-art cache analyses need to be highly context sensitive in order to be precise.

However, a cache analysis does not inherently require the exact information that can be obtained when memory blocks are used as abstract cache elements. Why is knowing the precise memory block that is accessed normally not needed? Being able to determine that an accessed memory block and a cached memory are the *same block* is sufficient to predict a cache hit; regardless of *what* memory block this is, i.e., regardless of its (memory) address. An analysis can exclude possible cache evictions once it can determine that an accessed memory block and cached memory blocks map to *different cache sets*. Again, regardless of the addresses of those memory blocks. In order to argue exertion of influence on cache sets, determining that an accessed memory block and cached memory blocks are mapped to the *same set, but are different memory blocks* suffices.

Consequently, Hahn and Grund propose to use *symbolic names*—unique identifiers of occurrences of address expressions—as abstract cache elements and approximate concrete cache contents exploiting relations between these symbolic names. Formally, their relational cache analysis as proposed in [Hah11, HG12] is defined as follows. As abstract cache elements, symbolic names are utilized, where a symbolic name is a name that uniquely identifies an occurrence of an address expression within a program. At all program points, a symbolic name $s$ associated with an occurrence of an address expression $o$ represents the most recently computed address at $o$.

The analysis itself consists of two modules: a so-called congruence analysis module and the actual relational cache analysis module. The sole purpose of the congruence analysis is to compute relations between possibly unknown addresses

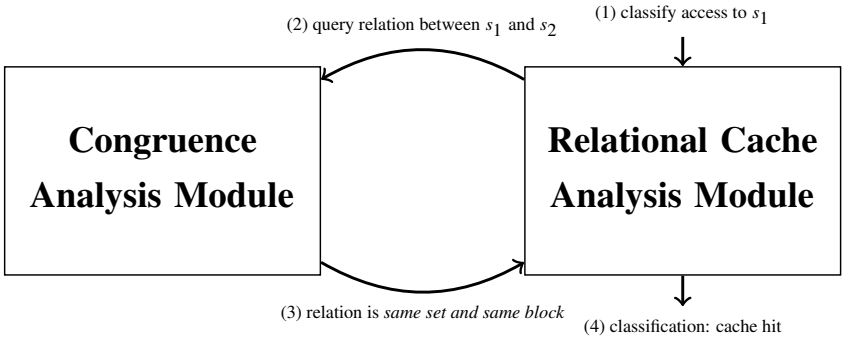
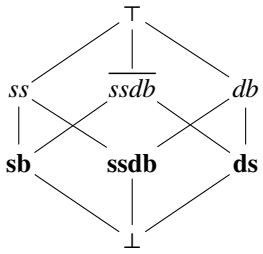



Figure 2.5: Interaction between the congruence analysis and the relational cache analysis modules.

and consequently relations between the symbolic names representing these addresses. The relational cache analysis module computes approximations to the cache contents by exploiting knowledge about relations between accessed and cached symbolic names provided by the congruence analysis module. Figure 2.5 depicts this interaction.

We already stated the relations a cache analysis can exploit (namely: same block, same set and different block, different set). However, a sound static analysis striving to determine these relations needs to be incomplete. Hence, they introduce additional relations in order to build-up a complete lattice of relations. Figure 2.6 shows the Hasse diagram of this lattice of relations $\mathcal{R}$.

The meaning of these relations can be defined as follows. Consider *reduced execution traces* that contain only the address computations and memory accesses of the original complete executions traces. Furthermore, $\langle s \mapsto a \rangle$ stands for an address computation yielding the address $a$ computed at the occurrence uniquely identified by symbolic name $s$. A memory access to the address computed most recently at the occurrence identified by symbolic name $s$ is denoted by $\langle s \rangle$. For




| Relation | Meaning |
|---|---|
| $ss$ | same cache set |
| $ds$ | different cache set |
| $sb$ | same block |
| $db$ | different block |
| $ssdb$ | $ss \wedge db$ |
| $\overline{ssdb}$ | $ds \vee sb \,(= \neg\,(ss \wedge db))$ |

Figure 2.6: Hasse diagram of the lattice of relations $\mathcal{R}$.

instance, given a reduced execution trace $\tau$, where

$$\begin{aligned}\tau \;=\; & \langle s_1 \mapsto \text{0x00A20128}\rangle\,\langle s_1\rangle\,\langle s_2 \mapsto \text{0x00A2012C}\rangle \\ & \langle s_2\rangle\,\langle s_2 \mapsto \text{0x00A20144}\rangle \\ & \langle s_1\rangle\,\langle s_2\rangle\,\langle s_1 \mapsto \text{0x00A21144}\rangle \\ & \langle s_1\rangle\,\langle s_3 \mapsto \text{0x00A20128}\rangle\,\langle s_1\rangle\,\langle s_2\rangle\,\langle s_3\rangle\end{aligned}$$

the following relations hold when assuming a cache architecture with 128 cache sets and a cache line size of 32 bytes. At the end of line 1, the same block relation ($sb$) holds between $s_1$ and $s_2$. By the end of line 2, their relation changed to the different set relation ($ds$). Their relation changes once more to same set, different block ($ssdb$) by the end of line 3. In the last line of the trace, symbolic name $s_3$ is introduced related to $s_1$ and $s_2$ by $ds(s_1, s_3)$ and $ds(s_2, s_3)$, respectively.

Formally, given a function $mb$ that maps memory addresses to the memory block encompassing this address, and a function $cs$ that maps memory addresses to the cache set their encompassing memory block is mapped to, the relation between two memory addresses $a$ and $b$ is given by

$$rel(a,b) = \begin{cases} sb & : mb(a) = mb(b) \\ ssdb & : cs(a) = cs(b) \wedge mb(a) \neq mb(b) \\ ds & : cs(a) \neq cs(b) \end{cases}$$

Let $\mathcal{S}$ denote the set of symbolic names, then congruence information maps

a tuple of symbolic names from $\mathcal{S}$ to a relation, i.e., an element of $\mathcal{R}$. Hence, congruence information can be modeled as a set of functions $\{cgr_v | v \in PLoc\}$, defined as

$$cgr_v : \mathcal{S} \times \mathcal{S} \mapsto \mathcal{R}$$

where *PLoc* is the set of program locations and a function value of $cgr_v$ is to be interpreted according to function *rel*. In general, any program point $v$ may be reached by several traces through the program. Useful congruence information at a program point needs to safely approximate relations holding on all these traces. Given a program $P$, we denote $\mathrm{T}_{P,v}$ the set of all traces through $P$ up to program point $v$. A function $cgr_v$ safely approximates congruence information at program point $v$ if for all symbolic names $s$ and $t$ as well as for all traces $\tau \in \mathrm{T}_{P,v}$ it holds:

$$cgr_v(s,t) \sqsupseteq \widehat{rel}(last(\tau, s), last(\tau, t))$$

where $last(\tau, s)$ evaluates to the address that has been computed most recently for symbolic name $s$ on trace $\tau$—or to $\bot$ if no address computation with symbolic name $s$ occurs on $\tau$. The function $\widehat{rel}$ extends *rel* as follows:

$$\widehat{rel}(a,b) = \begin{cases} \bot & : a = \bot \vee b = \bot \\ rel(a,b) & : \text{otherwise} \end{cases}$$

How can congruence information be used to implement a (relational) cache analysis? As in Ferdinand's cache analysis, Hahn and Grund propose to maintain upper bounds on the age of abstract cache elements. However, both approaches differ in that Hahn and Grund use symbolic names as abstract cache elements instead of memory blocks. Furthermore, Ferdinand proposes to analyze cache sets independently. With symbolic names and therefore the absence of absolute memory addresses, we do not have any information about which cache set is accessed. Hence, a concrete block represented by a symbolic name may reside in any cache set. What the analysis maintains is a function $ab : \mathcal{S} \to \{0, \ldots, k-1, \infty\}$ mapping symbolic names to upper bound on their age,where $k$ is the associativity. On a memory access, the evaluation of *ab* may need to be modified in order to still produce valid age bounds. A suitable update function is given in Definition 2.18.

**Definition 2.18** (Abstract Update Function for Relational Cache Analysis)**.** *Let*

$$U^{\leq}_{rel}(ab, s) : (\mathcal{S} \to \{0, \ldots, k-1, \infty\}) \times \mathcal{S} \to (\mathcal{S} \to \{0, \ldots, k-1, \infty\})$$

*be defined as:*

$$U^{\leq}_{rel}(ab, s) := \lambda t. \begin{cases} 0 & : c = sb \\ ab(t) & : c \in \{ds, \overline{ssdb}\} \\ ab(t) & : c \sqsupseteq ssdb \wedge ab(s) \leq ab(t) \\ ab(t)+1 & : c \sqsupseteq ssdb \wedge ab(s) > ab(t) \wedge ab(t) < k-1 \\ \infty & : c \sqsupseteq ssdb \wedge ab(s) > ab(t) \wedge ab(t) \geq k-1 \\ \infty & : c = \bot \end{cases}$$

*where* $c = cgr_v(s, t)$.

Again, program locations may in general be reached via several paths. Hence, age bounds computed along the different paths need to be combined in a sound manner.

Striving for a must-cache analysis, a sound combination of age bounds is to take the maximum age of all abstract cache elements. A sound join function for age bound functions can therefore be defined as

$$ab_1 \sqcup ab_2 = \lambda s \in \mathcal{S}. \max\{ab_1(s), ab_2(s)\}$$

Given such age bounds for abstract cache elements, a classification function

$$Class^{\leq}_{rel} : \mathcal{S} \to \{0, \ldots, k-1, \infty\} \times \mathcal{S} \to \{\mathrm{H}, \mathrm{M}, \top\}$$

to classify memory accesses as always hit (H) or always miss (M) can be easily defined as

$$Class^{\leq}_{rel}(ab, s) := \begin{cases} \mathrm{H} & : ab(s) \leq k-1 \\ \top & : \text{otherwise} \end{cases}$$

### Concluding Remarks and Discussion of Applicability on Programs Using Dynamic Memory Allocation

The relational cache analysis was just recently proposed and has not yet been implemented in industrial-strength tools. However, such a relational cache analysis

is guaranteed to always be at least as precise as Ferdinand's analysis. But it is also often able to classify stack-relative references and array references as cache hits in cases where Ferdinand's analysis fails to do so due to limited context sensitivity.

We can thus safely conclude that precise information about memory addresses is not a prerequisite for a precise cache analysis, but precise relations are. A relational cache analysis, hence, strictly lowers the requirements on the precision of absolute addresses for a precise cache analysis.

But how well can such a relational cache analysis perform in the presence of dynamically allocated objects? Accessing dynamically allocated data with an unknown cache set mapping does not require a modification of the update function given in Definition 2.18. However, we note that when referencing a symbolic name $m$ with a completely unknown memory address, $cgr_v(m, t)$ will evaluate to $\top$ for all $t \neq m$ (and to $sb$ for $t = m$). Hence, the update function will age all $t \neq m$ by 1 and set the age of $m$ to 0.

We observe that again, every entry of the cache is conservatively aged. However, as symbolic names are used and concrete addresses are not required, the relational analysis adds $m$ with age 0 to the cache. Unlike Ferdinand's analysis, where the youngest entry for all cache sets is set to unknown, the relational cache analysis may potentially derive cache hits for further, contemporaneous references to $m$.

Furthermore, we need to account for the cache influence of the allocation and deallocation routines themselves. We, again, assume the dynamic memory allocator to be a black box incurring a maximum of $L_{\mathcal{A}}$ and $L_{\mathcal{F}}$ accesses to pairwise different cache lines per cache set during allocation and deallocation, respectively. Then Definition 2.19 suitably models the influence of invocations of the allocator on the current cache state.

Again, when $L_{\mathcal{A}} \geq k$ or $L_{\mathcal{F}} \geq k$, each allocation and deallocation, respectively, leads to a complete loss of information regarding the current (must) cache state.

In summary, a relational cache analysis may be able to classify accesses to dynamically allocated objects as cache hits. However, there is still a significant loss of precision of information about cache states each time the analysis needs to consider references to unknown memory locations. Additionally, invocations of the dynamic allocator, whether they may be allocation or deallocation requests, result in a complete loss of information about potential cache states for (most)

dynamic allocators. The latter two observed sources of decreased precision are again inherent to the dynamic allocators and references to unknown memory locations and not a weakness of the analysis itself.

**Definition 2.19** (Abstract Update Function for Relative Cache Analysis For Invocations of the Dynamic Memory Allocator)**.** *Let*

$$U_{rel}^{\mathcal{A}^{\leq}}(ab) : (\mathcal{S} \to \{0, \ldots, k-1, \infty\}) \to (\mathcal{S} \to \{0, \ldots, k-1, \infty\})$$

*be defined as:*

$$U_{rel}^{\mathcal{A}^{\leq}}(ab) := \lambda t. \begin{cases} ab(t) + L_{\mathcal{A}} & : ab(t) < k - L_{\mathcal{A}} \\ \infty & : otherwise \end{cases}$$

*and*

$$U_{rel}^{\mathcal{F}^{\leq}}(ab) : (\mathcal{S} \to \{0, \ldots, k-1, \infty\}) \to (\mathcal{S} \to \{0, \ldots, k-1, \infty\})$$

*be defined as:*

$$U_{rel}^{\mathcal{F}^{\leq}}(ab) := \lambda t. \begin{cases} ab(t) + L_{\mathcal{F}} & : ab(t) < k - L_{\mathcal{F}} \\ \infty & : otherwise \end{cases}$$

## 2.5 References & Further Reading

**Dynamic Memory Allocation** Dynamic memory allocation is employed and has been a topic of research for over half a century now. Consequently, any summary of the topic as the one given in this chapter is bound to be brief on some aspects and leave out other aspects that are less relevant in the context. For a more detailed survey on dynamic memory allocation, we refer to [WJNB95] which reviews the majority of the literature on dynamic memory allocators published between 1961 and 1995. This survey establishes now widely accepted notations and performance measures to classify and compare dynamic memory allocators, dynamic memory allocation strategies, policies, and techniques. Dynamic memory allocation suitable for real-time applications, however, is not discussed in this survey. This is no surprise, as the *first* real-time dynamic memory allocator was proposed in 1995 in a time when real-time applications became more complex.

Until then, simple and fast constant-time allocators like the buddy systems were considered a suitable real-time alternative. An early, but very detailed review of these buddy systems can be found in [PN77]. The current state-of-the-art of real-time dynamic memory allocation is `TLSF` which was first proposed (as a *work-in-progress*) in 2003 [MRC03].

**Static Cache Analyses** Besides the two cache analyses we presented in detail, other approaches to cache analysis aiming at a local classification of memory accesses into cache hits and misses exist. The work probably closest to Ferdinand's analyses is the *static cache simulation for direct-mapped instruction caches* by Mueller et al. [MH93]. Their analysis classifies memory accesses as *always-miss*, *always-hit*, *first-miss* or *conflict*. While this seems to be more precise, similar results are obtained when combining Ferdinand's approach with virtual loop unrolling. Static cache simulation was also subsequently extended to cope with data caches [WHW+97].

Furthermore, Li et al. can handle caches within their approach to timing analysis using ILP formulations [LMW96]. With the extension proposed in [LMW96], their work can cope with caches and pipelines and handle path analysis. However, encoding all these aspects in an ILP formulation causes severe complexity issues, in practice rendering their approach infeasible but for very limited and simple cache and pipeline architectures.

# 3

# Static Precomputation of Allocation Schemes for Otherwise Dynamic Allocation Schemes

The best-laid schemes o' mice an' men
Gang aft agley,
An' lea'e us nought but grief an' pain,
For promis'd joy!

ROBERT BURNS
*in To A Mouse (1785)*

## 3.1 Chapter Overview

This chapter summarizes our work on algorithms for a static precomputation of memory locations for (otherwise) dynamically allocated memory structures.

As discussed in the previous chapter, dynamic memory allocation makes the static determination of tight bounds on an application's worst-case execution time challenging. As a consequence, current best practice in industry is to simply resort to static memory allocation. And, hence, circumvent the predictability issues of using dynamic memory allocation completely. However, there are advantages of dynamic memory allocation. With the techniques presented in this chapter, we

aim to statically precompute memory addresses and replace calls to the memory allocator by (functions returning) sequences of fixed addresses. This way, a programmer can make full use of dynamic memory allocation to alleviate the task of efficiently reusing memory, thus often significantly decreasing implementation as well as maintenance times. Once dynamic memory management is automatically replaced by a static allocation scheme, the final program does not suffer any additional predictability issues due to dynamic memory management. Furthermore, the modified program can be analyzed using current WCET analyses.

Figure 3.1 visualizes the field of application for the algorithms proposed in this chapter. As depicted, the workflow we aim at can be summarized as follows. We start with a program using dynamic memory allocation that is, however, intended to be deployed in a hard real-time setting. In a first phase, we apply a static program analysis to compute liveness information for the dynamically allocated objects. This information together with statically derived or user supplied loop and recursion bounds is used as input to our algorithms for precomputing static memory addresses. In the final phase, we replace calls for dynamic memory allocation, i.e., calls to `malloc`, by functions that simply return a sequence of precomputed addresses. Using fixed memory addresses, calls to `free` become obsolete and are consequently removed from the program. In the modified output program, the memory addresses of all objects are statically known. For such a program, current WCET analyses may compute tight bounds on its worst-case execution time as no additional uncertainty regarding its cache behavior is introduced by its memory allocation scheme anymore.

The remainder of this chapter is organized as follows. Section 3.2 summarizes our work on algorithms to precompute memory addresses for dynamic allocations in programs with numerical bounds on the number and sizes of allocations. The algorithms described in this section are already published in [HR09]. Section 3.3 elaborates on an algorithm that can also cope with programs for which only parametric bounds on the number and sizes of their allocations are available. This section reports on work already published in [HA10]. In Section 3.4, we discuss a suitable preanalysis to derive the inputs to the algorithms proposed in the previous sections. Section 3.5 concludes the chapter.

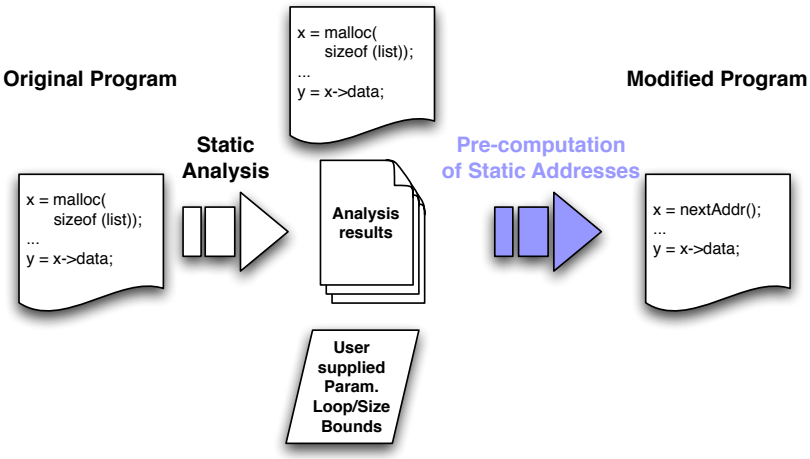



Figure 3.1: Workflow in which to employ our algorithms to precompute static allocations.

## 3.2 Precomputing Allocation Schemes for Programs with Numerical Bounds on Loop Iterations and Allocation Sizes

It is often argued that dynamic memory allocation prohibits a precise WCET analysis. We evidenced such an effect in Chapter 2.4. But may we nonetheless be able to exploit the special requirements for hard real-time systems to actually circumvent predictability issues introduced by dynamic allocation? Hard real-time software is guaranteed to contain no unbounded loops nor unbounded recursion. Hence, there can be no unbounded number of allocations. Assume all loops/iterations can even be bounded by a numerical value; as do the requested sizes. We can then use this information to transform dynamic memory allocation into static memory allocation.

Our proposed algorithms statically determine a memory address for each dynamically allocated heap object such that

- the WCET bound calculated by the standard IPET method is minimized,
- as a secondary objective the memory consumption is minimized, and
- no objects that may be contemporaneously allocated overlap in memory.

For the algorithms proposed in this section, we assume that for hard real-time applications a good memory mapping has to enable the computation of small WCET bounds. I.e., it has to enable a timing analysis to derive small bounds by either increasing the precision of the analysis or decreasing the actual WCET of the application.

**Memory Mapping** What do we mean by a memory mapping? Assume the main memory to be partitioned into *memory blocks* the size of a cache line such that each block maps into exactly one cache set. I.e., the memory blocks are aligned with the cache sets. We will later address the memory at the granularity of such memory blocks and never below. Consequently, *memory address* $(i-1)$ refers to the $i^{\text{th}}$ memory block. Analogously to the main memory, we also partition the program's dynamically allocated memory into blocks the size of a cache line which we call *heap allocated objects*. A memory mapping, then, assigns to each heap allocated object a memory block.

A heap allocated object may contain several program objects or just a fraction of a single program object. If a program object does not fit into a single heap allocated object, it has to be spread over several. Heap allocated objects holding a single program object have to be placed consecutively in memory in order to not break the program semantics. To this end, we introduce *heap allocated structures*: ordered sets of heap allocated objects. The heap allocated objects of a heap allocated structure must always be mapped to consecutive memory addresses.

We can relate heap allocated structures to dynamically allocated memory blocks of the analyzed program, i.e., program objects, like for example single nodes of a binary search tree or large arrays. We can also relate such heap allocated structures to entire data structures of the analyzed programs that are built from dynamically

allocated objects. Such data structures are for example linked lists consisting of several node objects. What relation we choose depends on the program we want to analyze. For example, it may for some applications be advantageous to consider a linked list structure to be a single heap allocated structure. Consider for example a program working on several, disjoint linked lists. Simply putting the objects of each list consecutively in memory and separating the different lists in the cache by choosing memory locations such that objects of different lists always map to different cache sets, may already constitute a good solution. I.e, a solution aimed at maximizing cache hits and minimizing cache misses due to cache evictions from accesses to objects of different structures. For other applications we may want to consider each single program object to be a single heap allocated structure. Consider for example an application performing an in-situ transformation of one of its internal data structures. Considering the source and the target structure each a single object, we have no opportunity for objects to share memory locations. Considering each (program) object of these structures to be a single heap allocated object, instead, enables objects to share memory locations.

Consequently, a memory mapping $m$ is a function that maps each heap allocated object to a memory address:

$$m = \bigcup_{o_{i,j} \in \mathcal{O}} \{o_{i,j} \mapsto a_{i,j}\}$$

Where $o_{i,j} \in \mathcal{O}$ is a heap allocated object, $a_{i,j}$ its memory address, and $(i, j) \in \mathcal{I} \times \mathcal{J}_i$ an index for elements of $\mathcal{O}$, the set of all heap allocated objects. Furthermore, $\mathcal{I}$ denotes an index set for the set of heap allocated structures and, for all $i \in \mathcal{I}$, $\mathcal{J}_i$ an index set for the set of (heap allocated) objects contained in structure $i$ or rather $\circ_{j=0}^{|\mathcal{J}_i|} \left(o_{i,j}\right)$ where $\circ$ denotes the concatenation of heap allocated objects.

**Problem Definition** Given a program $P$, we strive for a memory mapping that allows for a minimal WCET bound on $P$ that uses as little memory as possible. Or more formally, we strive for a WCET bound of

$$\min_{m \in M^\star} \{\text{WCET}(P, m)\}$$

where $M^\star$ is the set of all valid memory mappings, with the side condition to use as little memory as possible.

**An IPET Approach to Compute a Program's WCET** How can we compute WCET$(P, m)$, the execution time of the program's longest execution path? As proposed by Li and Malik, the longest execution path of a program can be computed by implicit path enumeration techniques (IPET) [LM95]. Their approach formulates the problem of finding the longest execution path as an integer linear program (ILP). Assume we have the control-flow graph $G(P) = (V, E, s, e)$ of a program $P$, where $V$ is the set of nodes, i.e., the basic blocks of $P$ and $E$ the set of edges between those nodes representing possible program flow. The start node of $P$ is denoted by $s$, its end node by $e$. Furthermore, assume loop bounds $b_k$ for the loops contained in $P$ to be also statically available. Then an ILP according to Li and Malik's approach is constructed as follows. First, we introduce two types of counter variables: $x_i$—the execution frequency of basic block $B_i$—for counting the number of executions of basic block $B_i \in V$ and $y_j$ for storing the number of traversals of edge $E_j \in E$. We can then represent and describe the possible control flow by stating that the start and end node of the control-flow graph are executed exactly once. This is modeled by the constraints generated according to Equation (3.1). Equations (3.2) and (3.3) generate constraints to model that each node is executed as often as the control flow enters and leaves the node, respectively. And finally, Equation (3.4) incorporates loop bounds into our ILP representation of the program's control flow. For each loop $l$ with an (upper) iteration bound of $b_l$, this equation adds a constraint ensuring that each outgoing edge of the first block $b$ within the loop is taken at most as often as the sum over all ingoing edges to $b$ times the loop bound $b_l$.

$$x_i = 1 \quad \text{if} \quad B_i = s \vee B_i = e \tag{3.1}$$

$$\sum_{j \in \mathcal{J}} y_j = x_k \quad \text{where} \quad \mathcal{J} = \{j \mid E_j = (\cdot, B_k)\} \tag{3.2}$$

$$\sum_{j \in \mathcal{J}} y_j = x_k \quad \text{where} \quad \mathcal{J} = \{j \mid E_j = (B_k, \cdot)\} \tag{3.3}$$

$$y_l \leq b_l \cdot \left( \sum_{j \in \mathcal{J}} y_j \right) \quad \text{where} \quad \mathcal{J} = \left\{ j \mid E_j \text{ is loop entry edge to } l \right\} \tag{3.4}$$

The WCET bound is then obtained using the objective function:

$$\max \sum_{i \in \{i \mid B_i \in V\}} (x_i \cdot c_i^m)$$

where $c_i^m$ is an upper bound on the WCET of basic block $B_i$ for a given memory mapping $m$.

**WCET-optimal Memory Mapping** Combining a computation of WCET($P, m$) as discussed above with our desired result from the problem definition yields a new objective function. This function incorporates different memory mappings into Li and Malik's approach. Given this new objective function, a WCET-optimal memory mapping yields a WCET bound equal to

$$\min_{m \in M^\star} \left\{ \max \sum_{i \in \{i \mid B_i \in V\}} (x_i \cdot c_i^m) \right\} \tag{3.5}$$

Unfortunately, our new problem does not constitute an ILP anymore. We propose to simply approximate solutions to Equation (3.5) as follows. Initially, we start with a memory mapping $m$ that uses minimal memory.We then compute the WCET of the program for this memory mapping $m$ using an ILP as described earlier. Subsequently, we improve the memory mapping of our current solution with respect to the WCET by selecting the basic block $B_i$ whose penalty due to conflict misses has the greatest contribution to the overall WCET bound and modifying $m$ such that $c_i^m$ is minimized. This last step is then repeated until no further improvements can be achieved.

Algorithm 1 implements this strategy as a hill climbing algorithm. Internally, this algorithm relies on methods to compute a bound on the WCET of a program using an IPET approach ($\text{WCET}_{\text{IPET}}()$), to compute an initial memory optimal mapping (*mem_opt*()), and to compute a block optimal mapping for a given basic block (*block_opt*()). To enable our hill climbing algorithm to escape local maxima, we allow for a certain number of side steps. A side step means that the algorithm selects an equally good or even worse mapping if no improved mapping can be found during an iteration. The maximum number of allowed side steps should be at least the number of program blocks in order to allow the optimization of each block. In our experiments, we set the maximum number of side steps to $2 \cdot |V|$.

**Algorithm 1:** Hill climbing algorithm to approximate a WCET-optimal memory mapping.

**Data**: functions $\text{WCET}_{\text{IPET}}()$, *mem_opt*(), and *block_opt*(); program $P$; integer *sideSteps* number of allowed side steps
**Result**: approximation to WCET-optimal memory mapping for dynamically allocated objects of $P$

$mapping_{best} \leftarrow mem\_opt()$;
$mapping_{curr} \leftarrow mapping_{best}$;
$skip \leftarrow \{\}$;
**while** $sideSteps > 0$ **do**
  calculate $\text{WCET}_{\text{IPET}}(P, mapping_{curr})$;
  select basic block $b$ with largest WCET contribution due to conflict misses such that $b \notin skip$ or return if no such block exists;
  $mapping_{tmp} \leftarrow block\_opt(mapping_{curr}, b)$;
  $skip \leftarrow skip \cup \{b\}$;
  **if** $WCET_{IPET}(P, mapping_{tmp}) < WCET_{IPET}(P, mapping_{curr})$ **then**
    $mapping_{curr} \leftarrow mapping_{tmp}$;
    **if** $WCET_{IPET}(P, mapping_{tmp}) < WCET_{IPET}(P, mapping_{best})$ **then**
      $mapping_{best} \leftarrow mapping_{tmp}$;
    **end**
  **else**
    $sideSteps \leftarrow sideSteps - 1$;
    $mapping_{curr} \leftarrow mapping_{tmp}$;
  **end**
**end**

An ILP formulation to implement $\text{WCET}_{\text{IPET}}()$ has already been proposed. The remaining memory mappings, *mem_opt*() and *block_opt*(), can also be computed by ILPs. We give generating equations for suitable ILP formulations in the following paragraphs.

**Memory Optimal Mapping (*mem_opt*())** Let $a_{i,j}$ be an integer variable of an ILP. Furthermore, let $a_{i,j}$ store the memory address of the $j^{\text{th}}$ heap allocated object of heap allocated structure $i$. Again, memory address does not refer to the physical address, but rather the $\left(a_{i,j}\right)^{\text{th}}$ memory block by partitioning the memory into blocks of the size of a cache line. For all $i, j, i', j'$ such that the $j^{\text{th}}$ object of structure $i$ may be allocated contemporaneously with the $j'^{\text{th}}$ object of structure $i'$, we add two constraints:

$$a_{i',j'} + 1 - a_{i,j} - b_{i,j,i',j'} \cdot C \leq 0 \tag{3.6}$$

$$a_{i',j'} - \left(a_{i,j} + 1\right) + \left(1 - b_{i,j,i',j'}\right) \cdot C \geq 0 \tag{3.7}$$

where the $b_{a,b,c,d}$ are auxiliary binary variables and $C$ a constant larger than the value any expression within the ILP can take. Note that such a $C$ may easily be computed statically. These constraints serve to ensure that parts of heap allocated structures allocated contemporaneously do not reside at the same memory address. Consider the case that $a_{i',j'} = a_{i,j}$ (which we want to exclude in a solution). In this case, the first of the two equations can be simplified to $1 - b_{i,j,i',j'} \cdot C \leq 0$ and the ILP solver must set $b_{i,j,i',j'}$ to 1 in order to satisfy the equation. This, however, makes the second equation, which can then be simplified to $-1 \geq 0$, unsatisfiable. As intended, both equations can only be satisfied in case $a_{i',j'} \neq a_{i,j}$.

To not break with the program semantics and change its behavior, we need to ensure a consecutive placement of parts of the same heap allocated structure. To this end, we add further constraints for all $i$, $j$ according to the following equation:

$$a_{i,j+1} = a_{i,j} + 1 \tag{3.8}$$

Let $\bar{a}$ denote the largest memory address used in the computed memory mapping. A correct value of $\bar{a}$ is ensured by adding the following constraints for all $i$, $j$:

$$a_{i,j} < \bar{a} \tag{3.9}$$

Finally, we need to provide an objective function for the ILP. As we want to minimize our memory consumption, we can simply minimize the largest address in use, i.e., $\bar{a}$:

$$\min \bar{a} \tag{3.10}$$

**Block Optimal Mapping (*block_opt*())** We can modify the ILP used to compute a memory usage optimal mapping in the previous paragraph to compute a block-optimal mapping as follows. As we defined block-optimal with respect to potential conflict cache misses, we start with a computation of the cache sets to which memory blocks are mapped that are accessed in the basic block for which we want to optimize the mapping.

Let $\#cs$ denote the number of cache sets and $cs_{a,i,j}$ denote binary variables set to 1 if and only if the $j^{\text{th}}$ block of heap allocated structure $i$ is mapped to cache set $a$. Equations (3.11) to (3.22) generate a set of constraints that set these $cs_{a,i,j}$ accordingly.

However, in order to do so, also several auxiliary variables are needed. Let $o_{i,j}$ denote the $j^{\text{th}}$ block of heap allocated structure $i$. Integer variable $a_{i,j}$, again, stores the memory address of $o_{i,j}$. Equations (3.11) to (3.14) set $cs_{i,j}$ to the cache set to which $o_{i,j}$ is mapped. The auxiliary variable $n_{i,j}$ also used in these equations stores the result of an integer division of the address of $o_{i,j}$ by $\#cs$. We generate constraints according to these equations for all $i$, $j$ such that block $j$ of structure $i$ is accessed in the considered basic block.

Equations (3.15) to (3.22) are used to generate constraints for all $a, i, j$, such that $a$ is a cache set and block $j$ of structure $i$ is accessed in the considered basic block. These constraints set further auxiliary variables. Namely, $y_{a,i,j}$ which is set to store the result of a computation of $|a - cs_{i,j}|$. Furthermore, variables $ltz_{a,i,j}$ are set to 0 if and only if $a - cs_{i,j} < 0$ holds.

Finally, we set all $cs_{a,i,j}$ by the constraints generated from Equations (3.21) and (3.22).

$$a_{i,j} - n_{i,j} \cdot \#cs - cs_{i,j} = 0 \tag{3.11}$$

$$n_{i,j} \geq 0 \tag{3.12}$$

$$cs_{i,j} \geq 0 \tag{3.13}$$

$$cs_{i,j} - \#cs + 1 \leq 0 \tag{3.14}$$

$$
\begin{aligned}
a - cs_{i,j} - ltz_{a,i,j} \cdot C &\leq 0 && (3.15)\\
a - cs_{i,j} + \left(1 - ltz_{a,i,j}\right) \cdot C &\geq 0 && (3.16)\\
a - cs_{i,j} &\leq y_{a,i,j} && (3.17)\\
-\left(a - cs_{i,j}\right) &\leq y_{a,i,j} && (3.18)\\
a - cs_{i,j} + \left(1 - ltz_{a,i,j}\right) \cdot C &\geq y_{a,i,j} && (3.19)\\
-\left(a - cs_{i,j}\right) + ltz_{a,i,j} \cdot C &\geq y_{a,i,j} && (3.20)
\end{aligned}
$$

$$
\begin{aligned}
y_{a,i,j} - C \cdot \left(1 - cs_{a,i,j}\right) &\leq 0 && (3.21)\\
y_{a,i,j} - \left(1 - cs_{a,i,j}\right) &\geq 0 && (3.22)
\end{aligned}
$$

We compute the number of potential cache misses $p_a$ due to cache conflicts resulting from accessing cache set $a$ using the following constraints for all cache sets $a$. With $b_a$ being auxiliary binary variables and $k$ denoting the associativity of the cache:

$$
\begin{aligned}
\left(\sum_{i,j} cs_{a,i,j}\right) - k - b_a \cdot C &\leq 0 && (3.23)\\
\left(\sum_{i,j} cs_{a,i,j}\right) - k + (1 - b_a) \cdot C &\geq 0 && (3.24)\\
0 &\leq p_a && (3.25)\\
\left(\sum_{i,j} cs_{a,i,j}\right) - (1 - b_a) \cdot C &\leq p_a && (3.26)
\end{aligned}
$$

As we aim to minimize the number of conflict misses that may occur during execution of the considered basic block, we replace our ILP's objective function by:

$$
\min\ C \cdot \left(\sum_{0 \leq a < k} p_a\right) + \bar{a} \tag{3.27}
$$

Where, again, we use a very large constant $C$ to weigh a reduction of potential conflict misses higher than any reduction in memory consumption.

**A Heuristic Approach** However, once the number of cache sets increases, the ILP to compute block optimal mappings becomes intractable; even independently of the number of heap objects. Preliminary experiments suggest that for target hardware with 256 or more cache sets solutions to the ILPs cannot be computed in reasonable time anymore. Furthermore, computing block optimal mappings using ILP formulations also introduces severe complexity issues for programs with a large number of heap allocated memory blocks.

Hence, in order to allow for an analysis of programs with more allocations as well as programs for a more complex target hardware with more cache sets, we propose to replace the block optimal ILP by a heuristic algorithm. Consider a simulated annealing [KGV83] algorithm as sketched in Algorithm 2 as an alternative implementation of *block_opt*(). Simulated annealing is a generic, probabilistic heuristics to find a good approximation to a global optimum by a non-exhaustive search. Intuitively, it mimics the cooling process of some material. The *temperature* corresponds to the probability to select a worse solution over the current one in order to escape local maxima. This probability decreases over time, like the temperature of a cooling material.

In this algorithm, we call $N(m)$, where $m$ is a memory mapping, the *neighborhood of m*. A neighborhood of a memory mapping $m$ is defined as the set of all memory mappings, such that the address of a single structure $s$ differs from its address in $m$ by exactly 1. Formally speaking, for a memory mapping

$$m = \bigcup_{o_{i,j} \in O} \{o_{i,j} \mapsto a_{i,j}\}$$

we define the neighborhood $N(m)$ as

$$N(m) = \left\{ m \oplus \left\{ o_{i,j} \mapsto a'_{i,j} \right\} \mid \left| a_{i,j} - a'_{i,j} \right| = 1 \right\}$$

The evaluation function *eval*() to determine the costs of a memory mapping $m$ is defined as

$$eval(m) = C^2 \cdot memoryConflicts(m) + C \cdot potentialConflictMisses(m) + maxAddr(m)$$

The first component ensures that memory mappings with *fewer* memory conflicts are preferred. Hence, a mapping with zero memory conflicts will be selected as the

**Algorithm 2:** Simulated annealing algorithm to compute a block optimal mapping.

**Data**: $Temperature_{MAX}$, $Temperature_{MIN}$, $Cooling\_Ratio$
**Result**: approximation of a memory mapping with minimal number of potential conflict misses in program block $b$

$restarts \leftarrow 0$;
$mapping_{best} \leftarrow mem\_opt()$;
**while** $restarts < RESTARTS$ **do**
  $mapping_{curr} \leftarrow mem\_opt()$;
  $Temperature \leftarrow Temperature_{MAX}$;
  **while** $Temperature > Temperature_{MIN}$ **do**
    compute the set $N(mapping_{curr})$ of neighboring mappings of $mapping_{curr}$;
    **foreach** $mapping_{tmp} \in N(mapping_{curr})$ **do**
      **if** $eval\left(mapping_{tmp}\right) < eval\left(mapping_{best}\right)$ **then**
        $mapping_{best} \leftarrow mapping_{tmp}$;
      **end**
      set $mapping_{curr}$ to $mapping_{tmp}$ with a probability $P_{set}(m_1, m_2)$ of $P_{set}\left(mapping_{curr}, mapping_{tmp}\right) = e^{\frac{eval\left(mapping_{curr}\right) - eval\left(mapping_{tmp}\right)}{Temperature}}$;
    **end**
    $Temperature \leftarrow Temperature \cdot Cooling\_Ratio$;
  **end**
  $restarts \leftarrow restarts + 1$;
**end**

algorithm can always use more memory locations. The number of potential conflict misses is then minimized by the second component. Among memory mappings with no memory conflicts and the same number of conflict misses, component three ensures that a memory mapping with minimal memory consumption is selected. This third component is used to ensure that in any selected mapping, there is an upper bound on the highest address assigned to any heap allocated object of $|\mathcal{O}|$.

As maximal temperature, we use $Temperature_{MAX} = |\mathcal{O}|^2$. The intention is to have $Temperature_{MAX}$ large enough to allow for almost random behavior at the beginning of each restart, converging to a local optimum. The current cooling ratio is $1.2^{-|\mathcal{I}| \cdot restarts}$, where $|\mathcal{I}|$ denotes the number of heap allocated structures.

Although a memory optimal mapping can be computed by our ILP formulation for most programs, *mem_opt*() can be easily replaced by a similar simulated annealing heuristic.

**Applicability** The approach described so far relies on the static availability of tight, numerical bounds on all program loops and requested allocation sizes. While this is an appropriate assumption for a large class of hard real-time applications, there already exist algorithms to derive parametric WCETs [AHLW08, BEL11, AAN11]. I.e., techniques that allow for parametric upper bounds on an application's WCET in case only parametric loop bounds are available. Unfortunately, our approach is, per construction, not applicable to this extended class of applications for which precise timing analysis is already available. It is, of course, desirable to enable dynamic memory allocation for these applications, too.

Furthermore, even our heuristic approach scales rather poorly with an increasing number of heap allocated objects. Table 3.1, shows an excerpt of the benchmark results presented in [HR09]. The target hardware was modeled after the PowerPC MPC603e. The PowerPC CPU utilizes separated 4-way set-associative instruction and data caches of 16 KB, each. The cache architecture consists of 128 cache sets with a cache line size of 32 bytes. These benchmarks analyze a program performing an in-situ transformation of a data structure.

Suppose said program transforms a data structure $A$ consisting of $n$ objects where each object occupies $l$ memory blocks. The target data structure, $B$, conse-

| Program | Target Hardware | Memory-/Block-Optimal Algorithm | Analysis Time (min/max/**avg**) ms |
|---|---|---|---|
| Structure-copy(3) | PPC603e | ILP/ILP | 23/79/**43.2** |
| Structure-copy(3) | PPC603e | ILP/Simulated Annealing | 23/89/**43.6** |
| Structure-copy(100) | PPC603e | Simulated Annealing/Simulated Annealing | 9,025/9,629/**9,345.1** |
| Structure-copy(500) | PPC603e | Simulated Annealing/Simulated Annealing | 260,401/276,797/**268,932.7** |

Table 3.1: Running times of example analyses each executed 10 times on a 2.66 GHz Core 2 Duo with 2 GB RAM. Excerpt of the benchmark results given in [HR09].

quently consists of $n$ objects, too. In $B$, however, each element occupies $k$ memory blocks. When transforming one data structure into another, dynamic memory allocation can easily lead to a memory consumption smaller than the sum of the memory needed to hold both structures by freeing each component of the first data structure as soon as its contents are copied to the second structure. Assuming that $k \geq l$, one can show that we need just $n \cdot k + l$ memory blocks for a program performing an in-situ transformation as described. In all instances, our algorithms computed solutions that use exactly $n \cdot k + l$ memory blocks.

While the computed allocation schemes are very promising with respect to memory consumption, the running times of the analyses themselves are less promising. Consider the last two entries where a data structure consisting of 100 and 500 elements, respectively, is copied. While the number of heap allocated objects increases by a factor of 5, the analysis' execution time increases by a factor of roughly 28.78 (on average). A similar effect can be observed by comparing entries two and three. Increasing the number of heap allocated objects from 3 to 100 results in a 214.34 fold increase of the running time of the analysis (on average).

The next section proposes a different approach that is applicable to both classes of real-time applications: those for which numerical bounds on loop iterations and requested allocation sizes are available as well as those for which only parametric bounds are available.

## 3.3 Precomputing Allocation Schemes for Programs with Parametric Loop Bounds and Allocation Sizes

While the previously proposed algorithms take into account the possible control flow of a program, our proposed algorithm to cope with parametric allocation bounds reduces a program to its *memory allocation behavior*. A formal description of a program's memory allocation behavior is consequently the only input data to this algorithm.

**Memory Allocation Behavior** To formally describe a program's memory allocation behavior, we start by collecting all *allocation sites*. An allocation site is simply an occurrence of `malloc` within the program. Let $M$ denote the set of all allocation sites.

By assumption, we statically know how often each allocation site may be reached during program execution: as loop and recursion bounds are known, at least parametrically. Let $P$ denote the set of parametric loop and recursion bounds, $L$ a possibly empty set of constraints on the parameters in $P$. We can then introduce a further set

$$U = \bigcup_{m \in M} \{u_m\}$$

where $u_m \in \mathbb{N} \cup P$ is an upper bound on how often allocation site $m$ may be reached, i.e., how often this function call to `malloc` may be invoked.

We furthermore assume an at least parametric bound on the requested allocation sizes. For each allocation site $m$, we construct a function $f_m$ such that $f_m(i)$ evaluates to the size requested at the $i^{\text{th}}$ invocation of `malloc` at allocation site $m$. These sizes may be overapproximated by intervals.

Consequently, we can construct a set of functions describing the allocation requests for all allocation sites as:

$$A = \bigcup_{m \in M} \left\{ f_m : \mathbb{N}^{\leq u_m \in U} \mapsto \mathcal{I}_m \right\}$$

where $\mathcal{I}_m$ is a set of intervals.

The set

$$R = \left\{(m, i) \mid m \in M \wedge i \in \mathbb{N}^{\leq u_m \in U}\right\}$$

contains all allocation requests that may occur during program execution.

**Optimization Goals** Any precomputed allocation scheme needs to be feasible. I.e., objects with overlapping lifetimes must not be mapped to overlapping memory locations. As with our previously discussed algorithms, we assume liveness information to be available. For our algorithm, we encode this information in a conflict function

$$C : 2^R \mapsto \{0, 1\}$$

that evaluates to 1 if and only if its argument contains requests for at least two memory blocks with potentially overlapping lifetimes.

Furthermore, when precomputing static memory addresses for originally dynamically allocated memory blocks, we do not want to ignore cache set mappings completely. In our previous approach, we aim for a cache-set mapping with a minimal number of potential conflict misses. While this is a reasonable heuristics, there are potential issues with this approach. As we just considered *potential* conflict misses, there is a risk that our algorithm removes potential, but in actual program executions never occurring conflict misses in order to keep or even introduce potential and actually occurring misses. Furthermore, with this approach, it is always unclear, whether a subsequent timing (or rather cache) analysis may even profit from the optimization. In the worst-case scenario, our optimization of potential conflict misses may, hence, actually decrease cache performance and analyzability or predictability.

In theory, there are two ways how to incorporate an optimization of the cache-set mapping into our algorithm. We may strive for cache-set mappings leading to further improved predictability of the program. This, however, requires us to be aware of many details of the subsequent cache analysis applied to the transformed program. Basically, to be able to decide which cache-set mapping may enable a subsequent cache analysis to classify the largest number of memory accesses as hits or misses calls for knowing the exact analysis algorithm. With such an approach, we also have to be careful not to decrease actual cache performance

by just considering predictability. The second option would be to just strive for good cache performance in order to reduce the risk that the statically precomputed addresses decrease program performance. In order not to rely on assumptions about subsequently applied analysis techniques or to be restricted to specific analyses, we favor the second option.

Unfortunately, under the assumption that $P \neq NP$, one cannot efficiently approximate an optimal placement of objects in memory that reduces the number of cache misses; not even up to a very liberal approximation ratio [PR02, PR05]. However, Chilimbi et al. showed that simply trying to place objects that are likely to be accessed contemporaneously next to each other in memory achieves significant increases in performance [CHL00]. To incorporate this simple, but efficient heuristics into our algorithm, we construct a bias function

$$\mathcal{B} : (R \times R) \mapsto \{0, 1\}$$

such that $\mathcal{B}(r_1, r_2)$ evaluates to 1 if and only if the block for which memory is requested in $r_1$ is likely to be accessed shortly prior to the one for which memory is requested in $r_2$.

But how can we statically obtain information about what objects will be accessed contemporaneously during program execution? Chilimbi's work relied on the user to provide this information. While this yields the most precise information in most cases, we may also approximate object access behavior using shape analysis [SRW02] in order to not rely on user interaction. We say that two objects $o_1$ and $o_2$ are likely to be accessed contemporaneously if there exist field pointers between $o_1$ and $o_2$. A third, more efficient, but potentially less precise way to gather this information is to apply an efficient data structure analysis [LA03] together with the heuristics that objects organized in the same data structure are likely to be accessed contemporaneously. We will elaborate on the issue of obtaining useful data regarding which objects are likely to be accessed contemporaneously in more detail in Section 3.4. For now, assume this information to be available to our algorithm.

| Symbol | Formal Definition | Intended Meaning |
|---|---|---|
| $M$ | | set of allocation sites |
| $P$ | | set of parametric loop/recursion bounds |
| $L$ | | set of constraints on elements of $P$ |
| $U$ | $U = \bigcup_{m \in M} \{u_m\}$ where $u_m \in (\mathbb{N} \cup P)$ | set of upper bounds on invocations of allocation sites |
| $A$ | $A = \bigcup_{m \in M} \left\{ f_m : \mathbb{N}^{\leq u_m \in U} \mapsto \mathcal{I}_m \right\}$ where $\mathcal{I}_m$ is a set of intervals | set of functions mapping invocations of allocation sites to (the requested) sizes |
| $R$ | $R = \left\{ (m, i) \mid m \in M \wedge i \in \mathbb{N}^{\leq u_m \in U} \right\}$ | set of all allocation requests (of a program) |
| $C$ | $C : 2^R \mapsto \{0, 1\}$ | conflict function, capturing liveness information |
| $\mathcal{B}$ | $\mathcal{B} : (R \times R) \mapsto \{0, 1\}$ | bias function to guide cache set placement according to Chilimbi's heuristics |

Table 3.2: Definitions used in our approach to precompute a static memory allocation scheme for a given dynamic memory allocation scheme.

Table 3.2 summarizes the definitions introduced in this section so far.

**Formal Problem Definition** Given a formal description of a program's memory allocation behavior and the discussed optimization goals, an allocation problem is a six-tuple

$$(M, U, L, A, C, \mathcal{B})$$

A (static) allocation scheme is a feasible solution to an allocation problem of the form

$$\bigcup_{r \in R} \{(r, addr_r)\}$$

where $addr_r$ denotes the precomputed starting address of the memory block requested by $r$.

An optimal allocation scheme is a feasible solution to an allocation problem such that (1) there is no other feasible solution with smaller memory consumption. And (2), considering the set of all feasible solutions with minimum memory

consumption, no solution exists that places more blocks for which the bias function $\mathcal{B}$ evaluates to 1 in memory next to each other.

Unfortunately, finding such optimal solutions is still at least NP-hard, despite our simple heuristics for reasonable cache-set mappings.

**Theorem 3.1** (NP-Hardness of the Computation of Optimal Allocation Schemes)**.** *Computing optimal solutions to an allocation problem is at least NP-hard.*

*Proof.* Let $(V, E, k)$ be a given instance of the $k$-colorability problem for a graph $G = (V, E)$. Generate the allocation problem

$$K = (V, \{1\}, \{\}, \{f : \{1\} \mapsto [1, 1]\}, C : R \times R \mapsto \{0, 1\}, \mathcal{B} : R \times R \mapsto \{0\})$$

where $\mathcal{B}$ maps all arguments to 0 and $C$ is defined such that

$$C(S) = 1 \Leftrightarrow \exists v_1, v_2.(v_1, v_2) \in E \wedge \{v_1, v_2\} \subseteq S$$

This transformation can be done in polynomial time and can be used to solve the $k$-colorability problem for $G$ as follows. Find an optimal allocation for $K$ and check whether less than or equal to $k$ memory addresses are used, in which case $G$ is $k$-colorable by associating each memory location with a (different) color. □

Consequently, we choose a heuristic approach to finding *good* solutions, rather than optimal solutions. As striving for optimal solutions would render our technique only applicable to very small applications.

**Problem/Input Normalization** The first phase of our proposed algorithm consists of a normalization of its input. In this initial phase, the algorithm transforms its input $I$ into a normalized input $I'$, where

$$I = (M, U, L, A, C, \mathcal{B})$$

as follows. For sake of readability, in the following, we denote a request for a block of dynamically allocated memory simply by an *allocated memory block*. For its normalization of $I$, the algorithm uses the bias function $\mathcal{B}$ to identify maximal ranges, i.e., concatenations, of allocated memory blocks that should be placed adjacent in memory. These ranges are subsequently split again to yield

new memory blocks such that the sizes of the resulting *normalized memory blocks* are multiples of the size of a cache line. However, the tailing normalized block resulting from splitting such a maximal range may be of smaller size. Also, splitting must not occur within the bounds of an allocated memory block. I.e., each normalized memory block of $I'$ consists of $n \geq 1$ complete, i.e., unsplit, allocated blocks from the original problem $I$.

This transformation yields a new, normalized allocation problem $I'$ for a parametric set of normalized blocks where

$$I' = (M', U', L', A', C')$$

In this normalized problem, $M'$ denotes the set of *abstract allocation sites*, while the functions collected in $A'$ map to the sizes of the single normalized blocks of these abstract allocation sites. An abstract allocation site, consequently, is a sequence of consecutive allocation requests. $U' \subset \mathbb{N} \cup P'$ and $L'$ are the accordingly updated constraints on loop/recursion bounds and parameters.

We gain two advantages from this transformation step. Firstly, the number of memory blocks considered by the algorithm is in general reduced, leading to a smaller problem instance. Secondly and more importantly, the bias function is consumed by this transformation. Consequently, the algorithm does not need to respect further constraints introduced by this function.

But is this normalization a reasonable exploitation of $\mathcal{B}$ and Chilimbi's heuristics? Chilimbi's heuristics is based on the following two assumptions regarding general program properties. First, spatial locality of memory accesses (see Section 2.3) strongly exists, i.e., addresses are more likely to be accessed if addresses near to them have already been accessed recently. Second, accessed objects *share* memory blocks and consequently cache lines when these shared blocks are cached. Hence, increased cache performance is attributed to (parts of) an accessed object being already cached as a result of recently accessing the object next to it in memory. These cache hits, however, are explicitly preserved by our normalization. Objects likely to be accessed contemporaneously are concatenated during the identification of maximal sequences of memory blocks. When these concatenations are split again, we only split at cache set boundaries. Hence, a potential cache hit due to spatial locality can never be precluded by our splitting strategy.

**Memory Block Chunks** Given the normalized input $I'$, the concrete, numerical values of the parameters $P'$ are unknown at design time. Hence, the resulting allocation scheme is parameterized in $P'$. To enable the algorithm to cope with this, we introduce the concept of *memory block chunks*.

A memory block chunk, or simply chunk, is a relative placement of memory blocks from different abstract allocation sites of $M'$ in such a way that there is no conflict within a chunk itself. Conflict, again, denotes a situation where two contemporaneously live objects overlap in memory. Chunks are then placed sequentially in memory; potentially repeatedly such that the memory blocks of each abstract allocation site $m \in M'$ occur exactly $u_m$ times. Formally, a chunk is a set of triples $(m, i, o)$, with the intended meaning that the $i^{\text{th}}$ request from abstract allocation site $m$ is located within the chunk at relative position $o$.

An allocation scheme, i.e., a solution to an allocation problem, consists of (potentially several types of) chunks together with the number of occurrences of each chunk. This number of occurrences is in general parametric. We distinguish two kinds of chunks: singleton chunks and repetitive chunks. A singleton chunk is a chunk that is generated exactly once, while multiple instances of repetitive chunks are generated.

By introducing these chunks, we reduce the computation of a solution to an allocation problem to finding an appropriate set of chunks that, when concatenated in any order, yield a feasible allocation scheme.

**Memory Chunks for an In-Situ Copy** Let us consider the example from the previously discussed approach again, where an in-situ transformation was performed by the analyzed application. Assume this to be the only operation of this application and no further heap allocated objects to be present. Furthermore, assume that we want to transform a singly-linked list of objects allocated at allocation site $M_s$ into a doubly-linked list of objects allocated at $M_d$. This transformation shall be performed using only a minimum amount of memory. Let the number of elements of the list be determined by a parameter $p$. Assume, the singly-linked list is traversed once and on this traversal the visited elements are copied to newly allocated elements of the doubly-linked list. Then, the $i^{\text{th}}$ element of the singly-linked list has a conflict with the $j^{\text{th}}$ element of the doubly-linked

if and only if $j \leq i$. For simplicity, assume the sizes of all list elements to be a multiple of the cache line size.

What kinds of memory block chunks would we expect our algorithm to compute? A possible allocation scheme with minimal memory consumption is given in Figure 3.2(a). This allocation scheme consists of three memory block chunks $c_1$, $c_2$, and $c_3$ where

$$c_1 = \{(M_d, 1, 0)\} \wedge c_2 = \{(M_d, i, 0), (M_s, i-1, 0)\} \wedge c_3 = \{(M_s, p, 0)\}$$

Both, $c_1$ and $c_3$ are repeated once, each. Memory block chunk $c_2$ is repeated $(p-1)$ times with variable $i \in [2; p]$. Formally, we get the following solution $S$ to our allocation problem

$$S = \{\{(M_d, 1, 0)\}, \{(M_s, p, 0)\}\} \cup \bigcup_{i \in [2,p]} \{(M_d, i, 0), (M_s, i-1, 0)\}$$

To yield an addressing scheme, we can concatenate chunks (possibly generated from chunk repetitions) in any order and map the resulting concatenation to memory locations. Let $S$ be a chunk set and

$$S^\star = s_1 \circ s_2 \circ \ldots \circ s_n \,.\, s_i \in S$$

an allocation scheme, i.e., a concatenation of the chunks contained in $S$. We extract a memory mapping from $S^\star$ by assigning memory addresses as follows. We first construct a function $addr_c : S \to \mathbb{N}$ mapping memory block chunks to (relative) memory addresses such that

$$addr_c(s) = \begin{cases} 0 & s = s_1 \\ addr_c(s_{r-1}) + |s_{r-1}| & \text{otherwise} \end{cases}$$

where the size of a memory block chunk is defined as

$$|s| = \max\{f_m(i) \in A' \mid (m, i, \cdot) \in s\}$$

Finally, a function $addr : R' \to \mathbb{N}$ assigns a (relative) target address to an (abstract) request $(m, i)$ in the straightforward manner

$$addr(m, i) = addr_c(s) + o \;,\; (m, i, o) \in s$$

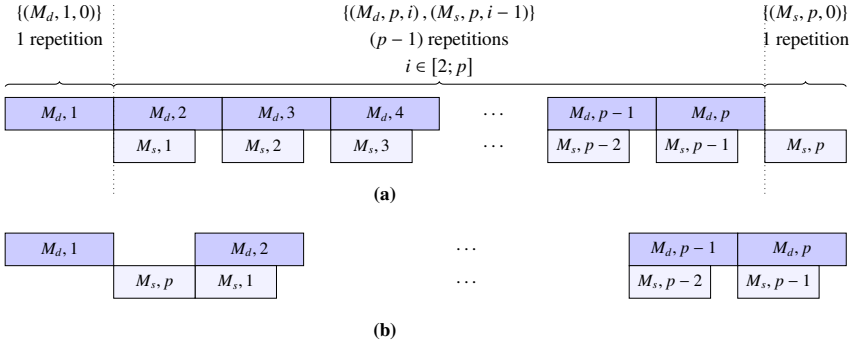



Figure 3.2: Allocation schemes for an in-situ list-copy with minimum memory consumption generated from the chunk set $\{\{(M_d, 1, 0)\}, \{(M_s, p, 0)\}\} \cup \bigcup_{i\in[2,p]} \{(M_d, i, 0), (M_s, i-1, 0)\}$.

Note that an allocation scheme corresponding to a given chunk set is not necessarily unique. We could also have concatenated the chunks as depicted in Figure 3.2(b), yielding an alternative, but still memory optimal allocation scheme.

**Computing Memory Block Chunks** But how can we efficiently compute such memory block chunk sets? The algorithm we propose to compute these sets of chunks for a given allocation problem works as follows. The algorithm maintains a workset of unprocessed requests for normalized blocks, i.e., requests not yet assigned to a chunk. We initialize this workset with the set of all abstract requests, i.e., with

$$R' = \left\{(m, i) \mid m \in M' \wedge i \in \mathbb{N}^{\leq u_m \in U'}\right\}$$

While the workset is not empty, the algorithm creates singleton chunks followed by sequences of repetitive chunks to remove requests from its workset. Algorithm 3 gives the pseudo code for this main routine of our algorithm.

The function *createChunk()* creates new memory block chunks and adds normalized blocks to these until (1) no further blocks are requested for a given abstract allocation site or (2) no further blocks can be added without either causing a

**Algorithm 3:** Algorithm to compute a suitable set of memory block chunks to yield reasonable allocation schemes.

```
Data: Problem specification I' = (M', U', L', A', C')
Result: Allocation scheme as a set of chunks
workset = Set of (abstract) requests R' obtained from M', U' and A';
while workset ≠ ∅ do
    createChunk(workset, true); // first chunk—unrolling
    removeProcessedRequests();
    if workset = ∅ then break;
    createChunk(workset, false); // create repetitive chunk
    computeRepetitions(); // repeat last chunk
    removeProcessedRequests();
end
```

conflict or (3) adding the block would exceed the size of the chunk. The order in which blocks are added is either given by the problem specification (in the case of singleton chunks) or in decreasing order of block sizes (in the case of repetitive chunks). This order also determines the size of a chunk. Algorithm 4 gives the pseudo code for this function.

While the function *sortByRequestSize()* is self-explanatory, we discuss the function *addRequestToChunk()* in more detail. This function adds requests for normalized blocks to a given memory block chunk in the following way. If the chunk is empty, then the request is always added at the first position (offset 0) of the chunk and the size of the memory block chunk is set to the size of this request. Subsequent blocks, i.e., requests for blocks added to a non-empty chunk, are temporarily placed at position $pos = 0$. In case this does not cause conflicts, the request is fixed there and the algorithm returns *true*. While conflicts do occur, the request is shifted to the next position $pos + 1$, until either all conflicts are solved and the requested block is fixed at this position or there is no space left in the chunk. In the latter case, the request is not added and the function returns *false* to notify its caller accordingly. Algorithm 5 gives the pseudo code for this operation.

**Algorithm 4:** Pseudo code for function *createChunk()*.

```
Data: Set of abstract requests R', boolean isSingleton
Result: Memory block chunk
if ¬IsSingleton then sortByRequestSize(R')
for (m, i) ∈ R' do
    boolean added = true;
    while i < u_m ∧ added do
        added = addRequestToChunk();
        if added then i = i + 1;
    end
end
```

**Algorithm 5:** Pseudo code for function *addRequestToChunk()*.

```
Data: Request r
Result: boolean added
if chunk.isEmpty() then
    addAtZero(r);
    return true;
end
int pos = 0;
while pos ≤ ChunkSize − sizeOf(r) do
    If ¬conflictInChunk() return true;
    pos++;
end
return false;
```

The problem specification may contain several non-numerical parameters. Also the sizes of requested blocks can be parametric. Hence, not all conditionals within the above functions may be computed directly. Only the set of parameter constraints $L$ given as part of the problem specification may be used to decide these conditions. If the set $L$ is, however, not sufficient to allow for deciding conditionals, we need to split the specification depending on the various outcomes.

For instance, a conditional

$$\text{if } p < q$$

leads to a split resulting in the following two problem specifications:

$$S = (M, U, L \cup \{(p < q\}, A, C, \mathcal{B})$$

and

$$S = (M, U, L \cup \{(p \geq q\}, A, C, \mathcal{B})$$

Hence, each time the set of restrictions on parameters, $L$, does not contain enough information to decide whether a conditional *cond* is satisfied, we replace the current allocation problem $S$ by two new allocation problems, $S_t$ and $S_f$. In $S_t$, we set $L$ to $L \cup \{cond\}$, while in $S_f$, $L$ is set to to $L \cup \{\neg cond\}$.

**Applicability** How well does this algorithm do in practice? Let us consider the in-situ copy example once more. This time, we use more realistic values and omit the assumption that the sizes of list elements are multiples of the cache line size. We also set the sizes of list elements to the same concrete values that we use in the evaluation of the algorithms assuming numerical bounds, in order to compare the new approach to our former one. The memory allocation behavior of such a program copying a singly-linked list to a doubly-linked list can be formalized as

$$(\{m_s, m_d\}, \{p_s, p_d\}, \{p_s = p_d\}, \{f_{m_s} : [p_s] \mapsto [8, 8], f_{m_d} : [p_d] \mapsto [12, 12]\}, C_{lc}, \mathcal{B}_{lc})$$

With the intended meaning that there are two allocation sites, one for the elements of the singly-linked list ($m_s$), one for those of the doubly-linked one ($m_d$). Each site can be reached at most $p_s$ and $p_d$ times, respectively. We know that $p_s = p_d =: p$ as all elements are copied and store this constraint in $L$. All elements of the singly-linked list are of size 8 bytes, those of the doubly-linked list of size 12 bytes. $C_{lc}$

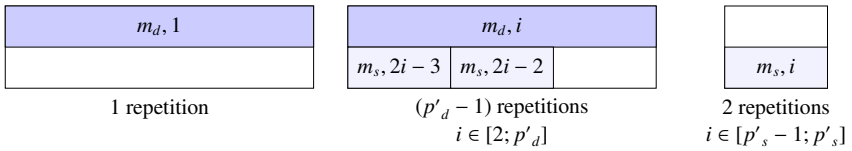



Figure 3.3: Allocation chunks for normalized in-situ list copy

is constructed, such that there are conflicts between list elements that are in-use at the same time. With $\mathcal{B}_{lc}$ constructed such that elements adjacent within a list are to be put adjacent in memory, our algorithm computes a set of memory block chunks as depicted in Figure 3.3. This normalization yields blocks of size 96 bytes and 32 bytes for the doubly- and singly-linked list, composed of 8 and 4 objects of the original lists, respectively.

Comparing this result to the allocation scheme given in Figure 3.2 quickly reveals that our algorithm did not compute a memory optimal allocation scheme for this problem. Our solution has two tailing memory block chunks consisting of a normalized block, each, together containing the tailing 8 elements of the singly-linked list. Instead of a singleton chunk containing just the tailing element of this list. Furthermore, the heading singleton chunk containing the head of the doubly-linked list contains now 8 elements of the doubly linked list instead of one. This is, however, not due to shortcomings of the algorithm. It is rather a result from the algorithm trying to preserve a maximal number of cache hits due to memory accesses following the principle of spatial locality, according to Chilimbi's heuristics. Our bias function $\mathcal{B}$ tells the algorithm that consecutive elements of the lists are likely to be contemporarily accessed. Hence, it strives to have consecutive elements overlap in cache lines. Which is why it always keeps 8 elements of the doubly-linked lists together, as $8 \cdot 12 = 96$ is the least common multiple of the size of an element (12 bytes) and the cache line size (32 bytes). Analogously, 4 elements of the singly-linked list of size 8 bytes, each, are always kept together. The different number of elements per abstract request also changes the conflicts. The second to last abstract request for singly-linked list elements (containing the first four of the last eight elements) now has a conflict with the last

abstract request for doubly-linked list elements (containing the last eight elements) and hence they cannot share a memory location anymore.

Assume we construct $\mathcal{B}_{lc}$ such that it always evaluates to 0 for all inputs, i.e., no bias is given, in order to prevent bias disabling the algorithm to compute memory optimal solutions. Then, our algorithm computes a memory optimal set of memory block chunks; at the price of potentially preventing cache hits during program execution.

Finally consider a reverse list-copy from doubly-linked to singly-linked elements

$$\Big(\{m_s, m_d\}, \{p\}, \{\}, \{f_{m_s} : [p_s] \mapsto [q, q], f_{m_d} : [p_d] \mapsto [12, 12]\}, C_{lc}, 2^R \mapsto \{0\}\Big)$$

with parametric sizes to demonstrate how the algorithm copes with incomplete information about requested sizes. At allocation site $m_s$ blocks of size $q$ bytes are requested, at site $m_d$ 12 byte blocks. Let $4 < q \leq 12$ bytes be statically known.

On this example, our algorithm computes the solutions depicted in Figure 3.4. Again, memory consumption of the solutions computed by our new algorithm is very promising.

Scalability was observed to not be a problem in any of the benchmarks performed [HA10]. Although in theory, splits due to incomplete information bear the potential for very high analysis times; or rather a large set of solutions. Furthermore, our parametric algorithm can also be applied to a broader class of programs than our previous approach. Hence, the limitations of the algorithms described in the previous section seem to be overcome.

However, we also note some disadvantages of this alternative approach. The main drawback is that we do not have a real notion of optimality regarding the computations of the algorithm. While in our first approach, we—at least implicitly within an ILP formulation—have a definition of what is optimal and strive to reach an optimum, our algorithm to solve parametric problem instances works differently. This algorithm simply constructs an allocation scheme according to heuristics that have proven useful. We can therefore not provide any guarantees about the quality of the computed solutions.

Furthermore, in order to yield good results, this algorithm requires very detailed information about the program for which an allocation scheme is to be computed.

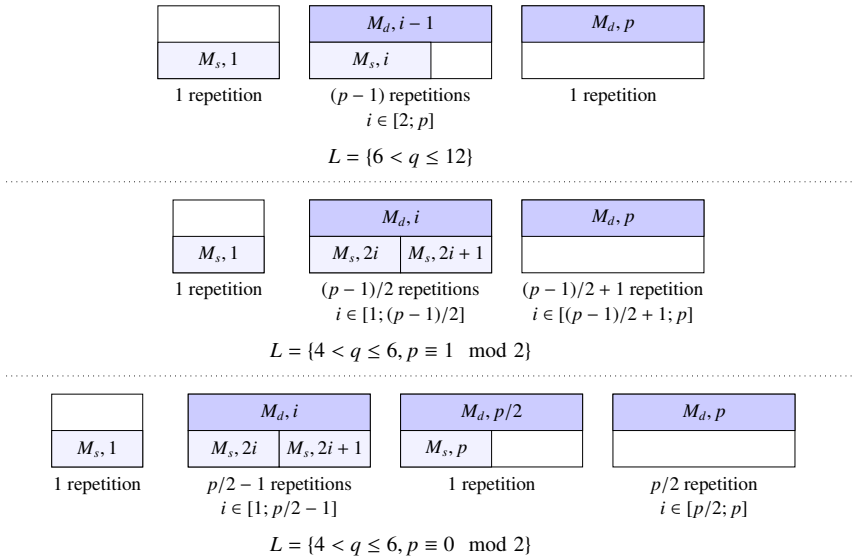



Figure 3.4: Computed memory block chunks for the reversed in-situ list-copy example with incomplete information.

Hence, there is a much higher dependence on a precise static (pre)analysis to derive the input for the parametric algorithm than there is for the numeric algorithms.

The following section summarizes our research aimed at constructing a static analysis to derive sufficiently precise information to enable our parametric approach to yield good results.

## 3.4 Allocation Site Aware Shape Analysis

The algorithms discussed in the previous section all rely on the availability of precise information about a program's allocation behaviors or at least the lifespans of allocated objects. In [Her10], we propose an extended, allocation-site aware shape analysis to statically derive this information. An extended shape analysis as proposed was subsequently implemented and evaluated [Leg13]. This section summarizes our proposal and elaborates on the applicability and precision of such an analysis.

**Shape Analysis** Shape analysis, generally, denotes a static program analysis that determines the *shape* of—or invariants that hold for—the heap and the interlinked structures built within at the different program points. Shape analysis techniques rely on a sophisticated heap abstraction that abstracts the in general infinite set of possible heap shapes to a finite one, but still preserves enough information to show that invariants do hold.

Most recent approaches to shape analysis rely either on separation logic [Rey02] to express inferred properties of structures arising on the heap [CR08, DOY06], or they model the heap by three-valued logical structures [SRW02]. The commonality of all these approaches is that they are *storeless*. I.e., they only model *what* heap structures may arise, not *where*, i.e., at what memory addresses, they reside on the heap, nor *where* and *when*, i.e., at what allocation site and which invocation thereof, they were allocated. This is, however, not a shortcoming of these analyses. For the current field of applications for shape analyses like checking data structure invariants [SRW02, CR08] and memory safety or even verifying partial program correctness [LARSW00], information about where and when heap objects were

allocated is not required. Hence, abstracting from allocation sites in order to increase performance is a reasonable, obvious design choice.

We propose to extend the framework for parametric shape analysis via three-valued logic proposed in [SRW02]. As to not lose focus of this chapter, we will just very briefly sketch this approach to shape analysis in the next paragraph. For a detailed discussion of shape analysis via three-valued logic, we refer to [SRW02].

**Shape Analysis via Three-Valued Logic** This shape analysis technique uses two-valued logical structures to describe concrete heap states. Each heap allocated object is represented by a logical individual, each pointer variable by a unary predicate that evaluates to *true* if and only if its argument is the individual representing the heap object to which the variable points. Field pointers referencing one heap object from another are analogously modeled by binary predicates. I.e., for each field $f$, we define a predicate $f(a, b)$ that evaluates to *true* if and only if $a$ is a logical individual representing a structure whose $f$ field points to the structure modeled by logical individual $b$.

In addition, so-called *instrumentation predicates* are employed to increase the precision or the performance of the analysis. Instrumentation predicates, basically, record information derived from other predicates and are defined in terms of a formula possibly including other (instrumentation) predicates. However, such a definition must of course not become mutually recursive. By explicitly storing the value to which a formula $\varphi$ evaluates within a structure in an instrumentation predicate, it is sometimes possible to extract more precise information from that structure than that obtained by simply (re-)evaluating $\varphi$.

Properties of the heap are formulated as logical formulæ. Such properties can then be easily checked to hold at a given heap state by evaluating their defining formulæ on the logical structure describing this heap state.

Effects of program statements on the heap state are captured by predicate-update formulæ that state how predicates are updated to yield a structure describing the heap state after the execution of these program statements.

Figure 3.5 gives an example for a shape graph. The figure depicts the graphical representation of a logical structure that describes three heap allocated objects organized in a singly-linked list. The single logical individuals are depicted as

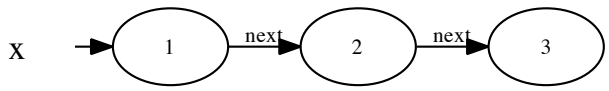



Figure 3.5: A shape graph depicting three heap objects organized in a singly linked list.

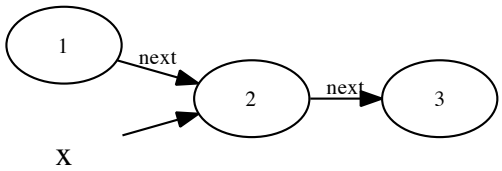



Figure 3.6: The resulting shape graph after advancing the pointer variable $x$ in the structure depicted in Figure3.5.

circles, predicates evaluating to *true* as arrows. Predicates evaluating to *false* are not drawn.

Applying the effects of the program statement

$$x = x{\rightarrow}next;$$

modeled by predicate-update formulæ

$$\begin{aligned} x(v) &\leftarrow \exists u.x(u) \wedge next(u,v) \\ next(u,v) &\leftarrow next(u,v) \end{aligned}$$

yields a structure as depicted in Figure 3.6.

A shape analysis of a given program can then be implemented as a fixed point computation collecting for each program point the set of logical structures describing all heap states that may arise at this point. For the starting point of the program, an initial heap description is required. However, an unbounded number of concrete heap descriptions may arise at program points. Therefore, abstract heap descriptions are introduced that use three-valued logical structures that can themselves represent a possibly infinite number of concrete two-valued logical

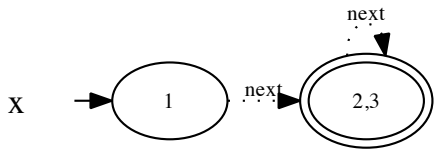



Figure 3.7: Abstract shape graph embedding (in particular) the structure from Figure 3.5.

structures. A concrete logical structure is abstracted by partitioning individuals into equivalence classes such that all individuals within one class yield the same truth values for a predefined set $\mathcal{A}$ of *abstraction predicates*. The individuals of the abstracted structures correspond to these equivalence classes. Abstract individuals that may represent more than one concrete individual are called *summary nodes*. Predicates not in $\mathcal{A}$ need to be reevaluated and may evaluate to the indefinite truth value $1/2$ in case not all concrete individuals summarized by the abstract individual evaluate to the same definite truth value.

Abstracting the structure from Figure 3.5 under $\mathcal{A} = \{x\}$ results in the three-valued logical structure depicted in Figure 3.7. Dotted arrows represent predicates evaluating to $1/2$ and summary nodes are drawn doubly circled.

Modeling the effects of program statements on abstract heap descriptions is done using the same update formulæ as in the concrete, two-valued setting. These formulæ are now simply evaluated using three-valued logic instead of two-valued logic. However, to increase precision, before applying update formulæ the *relevant* parts of the structure are concretized (*focus* or partial concretization). As focusing may generate contradicting or less precise structures, after application of the update formula, the resulting structures are *coerced* into more precise structures and contradicting structures are completely eliminated.

**Allocation-Site Aware Shape Analysis via Three-Valued Logic** How can we add allocation-site awareness to such an analysis? We propose to start by simply associating with each heap object *where* and *when* it was allocated. The number of allocation sites, i.e., calls to `malloc`, is statically known and for most

programs very small. Hence, to model *where* an object was allocated, it suffices to introduce additional unary predicates

$$alloc_m \in \{alloc_{m'} \mid m' \text{ is an allocation site}\} \subset \mathcal{A}$$

such that $alloc_m(u) = 1$ if and only if $u$ was allocated at program location $m$. Note that these predicates are defined as abstraction predicates.

Furthermore, to model *when* the object was allocated, we construct a function

$$t^\natural : \mathcal{U} \mapsto \mathbb{N}$$

that maps individuals of a concrete structure to invocations of an allocation site. In an abstract structure, we map to intervals of possible invocations

$$t : \mathcal{U} \mapsto \mathbb{I}$$

where the set of intervals is defined as

$$\mathbb{I} = \{[l, u] \,|\, l \in \mathbb{N} \wedge u \in \mathbb{N} \wedge l \leq u\}$$

Analogously, we add functions $s^\natural$ and $s$ to associate heap objects with their (requested) sizes.

Summarization of two individuals $v_1$ and $v_2$ is adapted as follows. Let the new summary node be $v_{sm}$, then

$$t(v_{sm}) = t(v_1) \sqcup t(v_2)$$

and

$$s(v_{sm}) = s(v_1) \sqcup s(v_2)$$

where

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min\{l_1, l_2\}, \max\{u_1, u_2\}]$$

Note that this least upper bound operator for intervals, the whole domain, rather, may be very imprecise. However, a first evaluation of the proposed analysis evidences no precision issues [Leg13]. Still, alternative, more precise domains may be used at this point. Simply (but expensively) using a power set domain (over

integers) as well as more sophisticated domains as an octagon domain [Min06] and interval polyhedra [CMWC09] are potential alternatives.

Furthermore note that this additional information modifies the implementations of the focus and coerce functions. As these operations are not relevant at this point, we omit a detailed discussion here. A discussion on how both functions are adapted can, however, be found in [Leg13].

The logical predicates are reevaluated as in the stateless framework.

**Precision & Performance of Such an Allocation-Site Aware Shape Analysis** We can further improve the results from such an allocation-site aware shape analysis. In a real-time setting, shape analysis can be performed almost arbitrarily precisely. Why is that so? Of course, as in the general setting, we can add instrumentation predicates to further increase precision. But we can, for real-time applications, also deactivate abstraction, i.e., the summarization of individuals, completely as no unbounded structures may arise as we statically know all loop and recursion bounds. However, abstract heap structures are still desirable as they may lead to significantly shorter analysis times.

The following set of (instrumentation) predicates and additional, precision increasing techniques have shown good tradeoffs between the precision and complexity of shape analyses; for both stateless and allocation-site aware analyses. First, in order to separate different data structures, a predicate $r_x(v)$ modeling reachability from program variables can be used. This predicate has already been shown to significantly increase the precision of general (stateless) shape analyses [SRW02]. We define for all pointer variables $x$ a reachability predicate as

$$r_x(v) := \exists u \,.\, x(u) \wedge fr(u, v)^*$$

where $x$ and $fr$ are predicates corresponding to pointer variables and field references, respectively. Second, we propose to not simply remove deallocated objects from shape graphs, but to mark such objects as freed. This can easily be achieved by a unary predicate $deallocated(v)$. Third, we propose to further increase the precision of partial concretization with respect to numeric intervals by allowing the analysis to mark predicates modeling field references with superscripts $^<$ and $^>$. These superscripts $^<$ and $^>$ indicate that if and only if the predicate evaluates to

*true*, then both arguments to the predicate are allocated directly after (in case of $^{<}$) or before (in case of $^{>}$) each other at the same allocation site.

Furthermore, in order to keep the number of different shape graphs per program location small, we do want an embedding of structures that differ only in numerical values used as interval bounds. To achieve this, we, basically, want to substitute numerical interval boundaries by variables. In our current implementation, this is implemented as a post-processing step after the fixed-point iteration finished. Hence, this step does not increase performance by reducing the number of structures during the fixed-point iteration. It does, however, produce more readable and more compact analysis results. To a certain degree, implementing this *variable abstraction* as a post-processing step is also a necessity. As, firstly, we cannot correlate interval boundaries based solely on a single structure arising during the fixed-point computation, since matching values could then be a pure coincident. And secondly, embedding structures too early may also decrease precision. During the evaluation of our allocation-site aware shape analysis, we observed that always embedding all structures that may be embedded after an iteration of the fixed-point computations does significantly hinder the detection of new, potential annotations as the exact invocation for a new node is lost too early [Leg13]. Hence, for each program point, we consider all shape graphs that have the same canonical abstraction and try to consolidate them into a single structure by using variables instead of differing numerical values to represent interval boundaries. If the same variable can be used in all structures to express the same boundaries, just with possibly different values, then it becomes more likely that these boundaries are really correlated [Leg13].

**Applicability** How does the result of an allocation-site aware shape analysis help in automatically finding *good* formal descriptions of an application's allocation behavior? A useful preanalysis allows an at least semi-automatic construction of an allocation problem instance of the form

$$(M, U, L, A, C, \mathcal{B})$$

$M$ can be directly extracted from the program code or, in our proposed shape analysis, is available as a subset of the abstraction predicates. In fact, our current

implementation automatically generates the corresponding predicates from the program code [Leg13]. $U$ and $L$ are either provided by the user or a static loop bound analysis [ESG+07, MBCS08, Hon09] and are input to our allocation-site aware shape analysis, too. $A$ is constructed from $M$, $L$, and the requested sizes. The size of a heap object, i.e., requested memory block, is explicitly stored in the shape graphs. The functions $C$ and $\mathcal{B}$ can be extracted from an allocation-site aware shape analysis as follows. $C$ evaluates to 1 if and only if representatives of at least two elements of its argument set are present in the same shape graph and none is marked as deallocated. $\mathcal{B}(r_1, r_2)$ evaluates to 1 if and only if there exists a field-pointer predicate evaluating to true for the individuals representing $r_1$ to $r_2$.

Please note that an allocation-site aware shape analysis as described in this subsection, while tailored directly at analyzing a program's allocation behavior, can also be employed to other fields of application. In [Her10], we already outlined the applicability of such an analysis to perform a data structure analysis [LA03] or an escape analysis [WR99].

**Evaluation** How does our proposed allocation-site aware shape analysis perform on our running example, the in-situ transformation? Consider the simplified C function given in Figure 3.1. Figure 3.8 gives the analysis results for the most general case for the three most relevant program locations. Those results are obtained in the evaluation elaborated on in [Leg13]. In Figure 3.8, the shape graph given in row **input structure** describes the initial heap state (at program location 1) and is provided as input to the analysis. Row **loop structure** describes a possible structure occurring at the end of the while loop (end of program location 14). This structure embeds most potential structures computed during the fixed-point iteration via our proposed variable abstraction. The structures not embedded within the depicted one constitute edge cases (first and last iterations). Examining this structure reveals the relevant liveness information to construct a conflict function as the one manually derived in [HA10]. Finally, row **final structure** depicts the analysis result for the function's final program location (at the beginning of line 17). From the computed shape graphs, the bias function from [HA10] can also be (re-)constructed automatically.

Listing 3.1: Simplified In-Situ List Copy Function

```
struct dll_el * copy( struct sll_el * src ) {                          1
    struct dll_el * result;                                            2
    struct dll_el * e;                                                 3
    ...                                                                4
    while ( src != NULL ) { /* loop bound exactly 256 */                5
            struct dll_el * tmp = malloc(...);                         6
            if ( result == null )                                      7
                    result = tmp;                                      8
            else                                                       9
                    e->next = tmp;                                     10
            e = tmp;                                                   11
            tmp = src;                                                 12
            src = tmp->next;                                           13
            free(tmp);                                                 14
    }                                                                  15
    ...                                                                16
    return result;                                                     17
}                                                                      18
```

In [Leg13], our proposed analysis is also tested on the `Dijkstra` benchmark program(s) (see Section 5.4, page 153). Again, the information that is extractable from the resulting shape graphs matches the information that we derived *manually* in [HA10].

## 3.5 Conclusions

Statically precomputing memory addresses and thus being able to remove the dynamic memory allocator completely gives many advantages. Obviously, without any need for calls to `malloc` and `free`, we do not have to worry about the preciseness of overapproximations of their execution times and cache effects. The resulting programs will be faster and also more predictable with static memory allocations.

Even with respect to memory consumption, a static allocation scheme may outperform dynamic allocation. If an application does not bear potential for memory reuse, static allocation always performs better as dynamic memory allocation

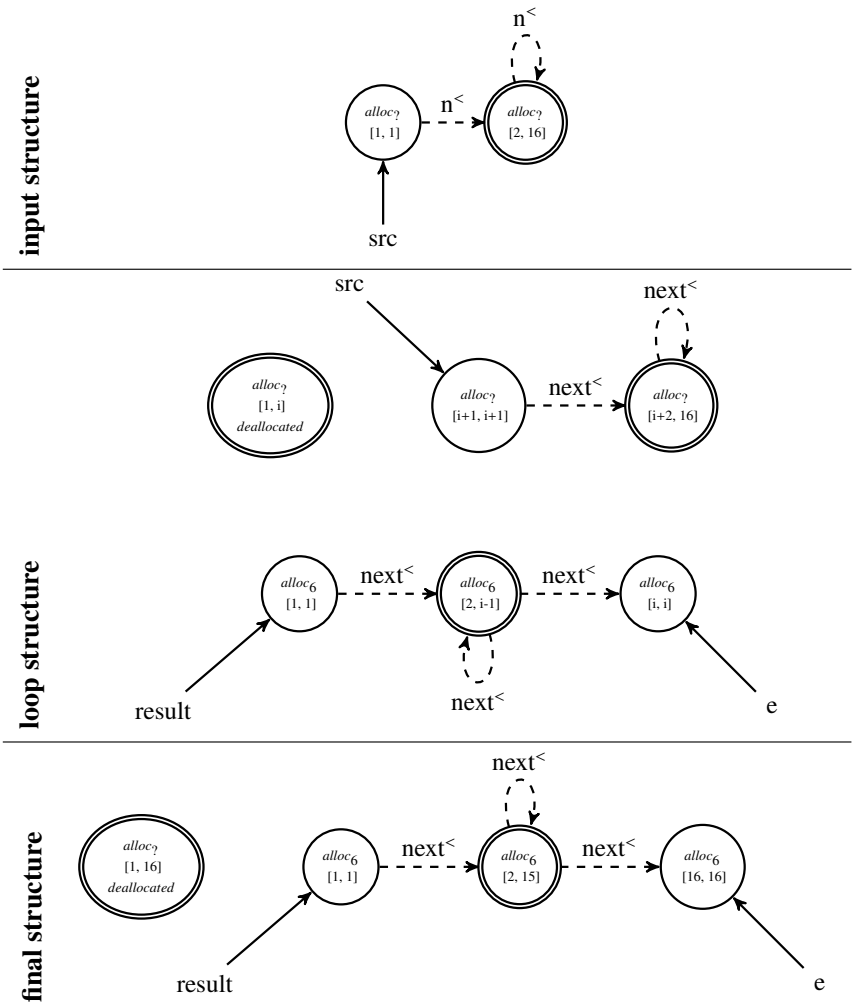



Figure 3.8: Analysis results for the in-situ list copy example.

always incurs a memory overhead for management information. However, for programs that can highly reuse memory, the memory performance of a static precomputation of memory addresses depends on the quality of the precomputed conflict function. We can differentiate two classes of applications here. Those for which no good conflict function can be constructed because statically we cannot decide which objects' lifetimes overlap. And those, for which this information may statically be available, but the employed analysis to derive the conflict function is too imprecise. For both classes, a static precomputation may need significantly more memory than is required with a dynamic allocation scheme.

In conclusion, static precomputation of memory addresses for otherwise dynamically allocated objects is the technique of choice if a static analysis is able to compute a reasonably good conflict function. From our experiments, our parametric algorithm generally outperforms the numeric ones as it normally yields a better cross-benefit ratio of analysis times vs. memory consumption of the computed allocation scheme.

# 4 Predictable Cache-Aware Memory Allocation

> Problems worthy of attack prove their worth by hitting back.
>
> PIET HEIN
> *Scientist, mathematician, inventor, designer, author, and poet (1905–1996)*

## 4.1 Chapter Overview

This chapter elaborates on the construction of a class of predictable, cache-aware memory allocators. Section 4.2 describes our main proposal how such an allocator can be constructed and what techniques can be employed to counteract memory fragmentation and increase overall performance. This section significantly extends and improves our algorithm presented in [HBHR11]. Section 4.3 discusses an alternative interface to our allocator that is better suited for a relational cache analysis as discussed in Section 2.4.2. In Section 4.4, we summarize work on `PRADA` [Hau12, HH13], an alternative, more lightweight approach to achieve cache-awareness and predictability in a dynamic memory allocator.

## 4.2 CAMA—A Cache-Aware Memory Allocator

What sets a predictable, cache-aware dynamic memory allocator apart from its general purpose counterparts? Ogasawara identified the following design goals that an allocator must meet in order to be considered sufficiently predictable for utilization in real-time systems: a WCET of (de-)allocations in $O(1)$ and touching only few cache lines during (de-)allocation operations together with a still reasonable memory efficiency [Oga95]. Ogasawara defined reasonable memory efficiency very loosely as performing better than a binary buddy system on random allocation traces. To effectively support a subsequent cache analysis, constant execution times and only a small set of possibly affected cache lines are in general not sufficient as discussed in Chapters 2.4.1 and 2.4.2. We therefore extend the list of design goals that a predictable, cache-aware allocator must meet as follows. The allocator shall also be able to provide guarantees about the cache-set mapping of the memory blocks it returns to satisfy allocation requests. Furthermore, more detailed knowledge has to be statically available about the cache-set mapping of memory blocks that may be read or written during the allocator's operations. Hence, in the design of CAMA, we aimed at constant execution times for allocation and deallocation requests, the possibility to guide the allocator with respect to which cache set the returned memory addresses are mapped to, and the absence of statically unpredictable influences on the cache, i.e., cache pollution, by (de-)allocation operations. We added an additional parameter to allocation requests, so that the memory allocation function now has two parameters: the requested block size and the cache set to which the block's memory address shall be mapped.

**How to Manage Free Memory Blocks with Constant-Time Allocation and Deallocation in Mind** CAMA uses a segregated-fit allocation algorithm like Ogasawara's `Half-Fit`. Segregation is done on multiple-levels to incorporate cache mapping information and to decrease internal fragmentation by using finer grained size classes. The latter has originally been proposed by Masmano et al. for their Two-Level Segregated Fit (`TLSF`) [MRCR04, MRR+08]. Consequently, CAMA organizes the free memory blocks currently managed by the allocator in

segregated free-lists, where a single segregated list consists of all memory blocks within the same size class and whose (starting) memory addresses are mapped to the same cache set.

Formally, we associate with each free memory block a tuple $(addr, size)$, where $addr$ is the starting address of the free block and $size$ is its size (in bytes). Let $\mathcal{B}$ denote the set of all such tuples associated with the free blocks currently managed by our allocator. Furthermore, let $S_{k,i,j}$ denote the set of tuples associated with free blocks whose start addresses are mapped to cache set $k$ and whose size is in the interval $I_{i,j}$. We define these intervals as

$$I_{i,j} = \left[ 2^i + \frac{j}{j_{max}+1} \cdot 2^i ; 2^i + \frac{j+1}{j_{max}+1} \cdot 2^i \right)$$

where $i$ and $j$ are index variables ranging from $i_{min}$ to $i_{max}$ and 0 to $j_{max}$, respectively. These intervals constitute the size classes of our allocator. Note that they are constructed using the same two-level approach to building size classes that is used with `TLSF`. I.e., the first level (the $i$ index) *logically generates* the series of intervals containing all numbers between two succeeding powers of two. While the second level (the $j$ index) partitions these exponentially growing intervals into a fixed number of $(j_{max}+1)$ intervals of equal size. Figure 4.1 illustrates the partitioning of the natural numbers by these intervals for $i_{min} = 2$, $i_{max} = 6$, and $j_{max} = 1$.

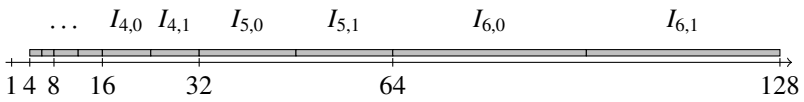



Figure 4.1: Illustration of $\biguplus_{2 \le i \le 6 \wedge 0 \le j \le 1} I_{i,j}$ as a partition of $[4, 128]$.

We observe that the $S_{k,i,j}$ are pairwise disjoint and formally defined as

$$S_{k,i,j} = \left\{ (addr, size) \in \mathcal{B} \,\middle\|\, \left\lfloor \frac{addr}{size_{cline}} \right\rfloor \equiv k \mod sets \wedge size \in I_{i,j} \right\}$$

where $size_{cline}$ denotes the size of a cache line and $sets$ the number of cache sets.

If we enforce a minimum size for managed memory blocks and choose $i_{min}$ appropriately, the set of all $S_{k,i,j}$ is a partition on $\mathcal{B}$, i.e.,

$$\biguplus_{\substack{0 \le k < sets \\ i_{min} \le i \le i_{max} \\ 0 \le j \le j_{max}}} S_{k,i,j} = \mathcal{B}.$$

We can associate a segregated list with each set $S_{k,i,j}$ and put all memory blocks associated with tuples in $S_{k,i,j}$ into this segregated list. Satisfying an allocation request for a block of size *size* whose memory address is mapped to cache set $k'$ is reduced to determining the smallest set $S_{k,i,j}$ with which only blocks large enough to satisfy the request are associated. The allocator can then return any block from the segregated list associated with this set. This set can be determined in constant time by computing its index triple $(k, i, j)$ as

$$\begin{aligned} k &= k' \\ i &= \lfloor \log_2(size) \rfloor \\ j &= \left\lceil \frac{\left(size - 2^i\right)(j_{max} + 1)}{2^i} \right\rceil \end{aligned}$$

Please note that due to the rounding of $j$, we may compute an invalid third index component of $j_{max} + 1$. In that case, we get the desired result by setting $j$ to 0 and $i$ to $i + 1$.

For deallocation, we also have to determine the appropriate segregated list for the newly freed memory block. Remember that this list depends solely on the size of the block and the cache-set mapping of its starting address. While the latter could be derived directly from the pointer passed to the deallocation procedure, the size information would have to be otherwise supplied. Alternatively, there is also the option to store an explicit pointer to the appropriate free list at a block header preceding the managed memory blocks. This alleviates the allocator from any address computations to determine the associated free lists, while still requiring the same amount of management overhead.

**How to Implement Splitting and Merging Techniques in a Predictable, Cache-Aware Manner** To counteract external fragmentation, we further need

to incorporate splitting and merging capabilities into CAMA. In order to be able to split larger blocks into smaller blocks at allocation time to satisfy requests for smaller blocks using a larger block (splitting), we proceed as follows. We introduce for each cache set $k$ a bit vector

$$v_k \in \{0,1\}^{(i_{max}-i_{min}+1)(j_{max}+1)}$$

such that the $n^{th}$ component of $v_k$ is 1 if and only if

$$S_{k,i',j'} \neq \emptyset$$

where $i' \coloneqq \lfloor n/(j_{max}+1) \rfloor$ and $j' \coloneqq n \mod (j_{max}+1)$.

Our adapted allocation procedure still starts with computing an address triple $(k,i,j)$ upon a request for $\left(size \in I_{i,j}\right)$ bytes starting in cache set $k$. However, in a second step, the allocator scans $v_k$ to find the non-empty segregated list which is associated with the smallest size class still large enough to satisfy the requested block size, assuming such a list exists. Technically, we look for the first bit set to 1, starting from the $(i \cdot (j_{max}+1)+j)^{th}$ bit of $v_k$. Assume this bit to be the $n'^{th}$ component of $v_k$. Any block from the list containing the free blocks associated with the set

$$S_{k,\lfloor n'/(j_{max}+1)\rfloor,(n' \bmod (j_{max}+1))}$$

can, per construction of $v_k$ and $S_{k,\cdot,\cdot}$, be used to satisfy the request. Furthermore, in the current state of the allocator, this list contains the smallest blocks large enough to satisfy this request.

CAMA splits blocks from this list if the blocks have a minimum size large enough such that after splitting a block $b \in S_{k,\lfloor n'/(j_{max}+1)\rfloor,(n' \bmod (j_{max}+1))}$ into two blocks $b_l$ and $b_r$ it holds that

1. $b_l$ is suitable to satisfy the original request, i.e., the tuple $(addr, size)$ associated with $b_l$ would be an element of $S_{k,i,j}$ according to our partitioning and

2. the size of $b_r$ is at least the minimum size for managed memory blocks, i.e., given the tuple $(addr', size')$ associated with $b_r$, it holds that $size' \geq 2^{i_{min}}$.

Otherwise, i.e., when the size criteria for $b_r$ cannot be satisfied, CAMA does not split, but simply returns $b$. While the latter case returns a larger block than requested to satisfy the allocation request and thus causes internal fragmentation, this is still preferable over potentially much larger external fragmentation, when new memory has to be obtained in order to generate new memory blocks to satisfy the request with a better fitting memory block.

But is allocation still a constant-time operation when splitting requires scanning some bit vector? Obviously, we can statically determine the size of those bit vectors: each bit vector contains one bit per size class. Hence, at allocation $(j_{max} + 1) \cdot (i_{max} - i_{min})$ bits have to be read in the worst case. This gives us a constant upper bound on the number of read operations necessary to read the whole bit sequence. In our prototype implementation with 48 size classes per cache set (12 power-of-two classes, each split into 4 linearly increasing subclasses), at least two and at most three 32 bit words have to be read. This number of additional read operations to some bitmap in order to *scan* all relevant free lists in constant time for a suitable free block is very similar to that of `TLSF`. `TLSF` reads one (best case) or three (worst case) 32 bit words at allocation time to identify the smallest non-empty size class with blocks large enough to satisfy the current request.

After splitting, we need to insert $b_r$ into the appropriate free list. Technically, we set a back pointer as a block header of $b_r$ as discussed above. Consider organizing our free lists in the usual way, i.e., consisting of the free memory blocks themselves with each free block containing pointers to its predecessor and successor within the free list. Adding $b_r$ to a free list would generate three more memory accesses for adjusting these links and setting $b_r$ as the new first block in the list. This sums up to potentially four accesses. Statically, we can in general never know whether these additional memory accesses occur during an allocation. We may, of course, narrow down the number of affected cache sets from those accesses in case we split a larger than requested block starting in the requested cache set. But what happens if an allocation requires to increase the managed memory and hence splitting a memory block with an unknown cache-set mapping of its start address? Unfortunately, in this case, no precise cache-set mapping for any of these four accesses can be statically determined. Note that the second, new free block created in this scenario is located left of the split block used to satisfy the request

and its cache-set mapping is completely unknown. Hence, a subsequent cache analysis would have to cope with four *unknown* cache accesses.

A cache-aware memory allocator, however, should strive to support subsequent cache analyses by providing strong guarantees regarding which cache sets may be accessed during allocation operations. To be able to provide such guarantees, we do not organize the free memory blocks directly in the segregated free-lists as is common practice. Instead, we build CAMA's segregated free lists over so-called *descriptors*, small management units logically built-up as depicted in Figure 4.2.

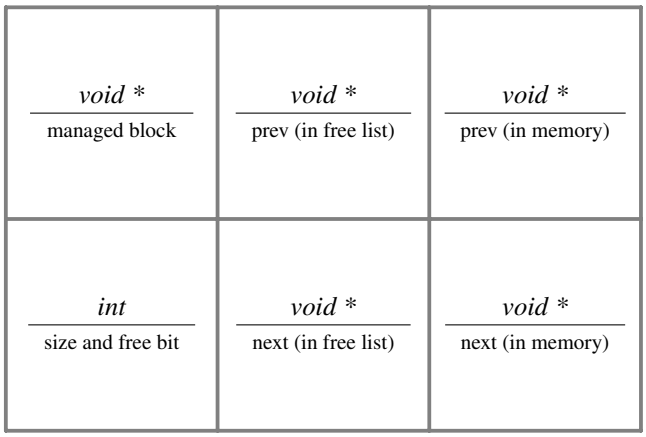



Figure 4.2: Layout of a descriptor.

As shown, a descriptor stores the following information. A pointer to the memory block for whose management it is used as well as the size of this block. Those two entries can be used to compute the address triple $(k, i, j)$ to identify the appropriate free list into which to insert the block's descriptor when the memory block is deallocated. Additionally, one bit of the size entry is used as a *free bit* indicating whether the memory block referred to by this descriptor is currently free, i.e., deallocated, or in-use, i.e., allocated. A descriptor further stores pointers to the descriptors of the memory blocks physically adjacent to the memory block associated with the given descriptor. Finally, we want to organize descriptors denoting currently free memory blocks in doubly-linked free lists.

Therefore, a descriptor stores pointers to its pre- and successors in these free lists. CAMA guarantees to place descriptors exclusively at memory locations mapped to a predefined range of cache sets. Thus, accesses to descriptors always result in accesses to a statically known, limited part of the cache. Descriptors are themselves relatively small memory blocks. Therefore, it seems appropriate to allocate them in bulks: in *descriptor areas* that span the whole part of a cache page mapped to the cache-set range destined for descriptors. This approach reduces the block headers of memory blocks managed by CAMA to a single pointer to its descriptor, no matter if the block is currently allocated, i.e., in-use or free for allocation.

External fragmentation may be further counteracted by an *inverse* operation to splitting: the merging of consecutive free blocks into a single large free memory block. This *merging* is performed at deallocation time in order to satisfy requests for larger blocks later on. As noted earlier, each in-use block stores a pointer to its descriptor block. Furthermore, a descriptor contains pointers to the memory blocks residing in memory locations adjacent to the memory location in which the block managed by this descriptor resides. If a memory block is deallocated, we can hence easily check whether its left-adjacent and/or right-adjacent neighboring blocks are currently free. To determine whether a memory block is currently in-use or free, we use the additional free bit within the size entry of the block's descriptor. If one or both adjacent blocks are currently free, we merge these blocks into a single free block. This merging can be done by updating entries in descriptors and consequently requires only accesses to statically known cache areas. As the memory blocks are merged into one block, one descriptor is used to manage this new block, while the descriptor of the second memory block is not needed anymore.

Hence, storing management information in descriptors instead of at the free blocks themselves enables us to implement splitting and merging operations in a predictable, cache-aware manner. However, this may come at the price of increasing fragmentation.

**Counteracting Internal Fragmentation From Descriptor Overhead** Obviously, descriptors are bigger and hence cause higher internal fragmentation than

*traditional* headers. The ratio is six entries for a descriptor versus one to two entries for a header used by a buddy system and TLSF, respectively. For larger memory blocks, this increase in management overhead is negligible. But what about very small blocks of just a few bytes themselves? One could adjust the minimum size for managed blocks such that a reasonable ratio of block size to management overhead is achieved. However, given the four times larger management overhead by using descriptors, this could put CAMA at huge disadvantage when compared to other allocators. This is especially true since requests for small memory blocks are the majority of all allocation requests for a significant number of programs [WJNB95]. Another approach would be to disable splitting and merging for very small blocks completely. In that case, such blocks would not need descriptors, but could be easily managed while in-use with a simple back pointer to their appropriate free list when deallocated. If not in-use, such a block could be organized in its free list by storing the necessary pointers in its (payload) body. While splitting small blocks would not occur anyways, this approach, however, would also prevent the merging of small blocks. Merging consecutive free small blocks to yield a larger block with an increased probability of being useful in the future, in turn, is often desirable.

To enable splitting and merging for all block sizes without incurring any of the problems mentioned above, we propose the following approach. Intuitively, we want small blocks to *share* a common single descriptor responsible for managing the whole group of small memory blocks. The smaller the block sizes are within a group that shares a descriptor, the bigger that group has to be—in terms of memory blocks organized within. To achieve this effect, we introduce so-called *area blocks*: consecutive memory ranges divided into a fixed number of equally-sized blocks. An area block is managed by a single descriptor. The number of memory blocks building-up an area block is determined by the size threshold $size_{bmin}$ that regulates when a memory block is large enough to have its own, exclusive descriptor. For blocks of size $s < size_{bmin}$, areas of $k(s)$ blocks of $s$ bytes, each, are created, where $k(s)$ is the smallest number such that $k(s) \cdot s$ constitutes a size large enough to be eligible for an exclusive descriptor. I.e.,

$$k(s) = \min \{n \in \mathbb{N} \mid n \cdot s \geq size_{bmin}\}$$

However, this reduced internal fragmentation comes at the price of reduced cache predictability. For small blocks sharing a descriptor, we cannot guarantee a single, concrete cache set to which their starting memory addresses are mapped. Instead, we can guarantee only an interval of cache sets containing all sets to which starting locations of the blocks grouped in an area are mapped.

**Counteracting External Fragmentation Caused By Descriptor Placement**

External fragmentation may be increased by using descriptors as well—even much more severely than internal fragmentation. Consider a naïve implementation of a dynamic memory allocator as outlined so far. Assume the allocator does only reuse descriptors but not recycle descriptor areas, i.e., it never merges free descriptor areas with adjacent free memory blocks. Such an allocator is prone to generate states as sketched in Figure 4.3.

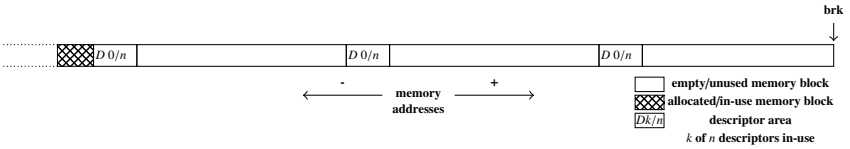



Figure 4.3: Empty descriptor areas fragmenting the managed memory range.

In such a state, a request for a very large (or for a moderately large block that starts in a cache set closely preceding the cache set range used for descriptors) forces the allocator to increase the managed memory range, although, enough unused memory would be available.

Even with an intelligent recycling for descriptor areas, as long as the only restriction on descriptor placement is that their memory locations must be mapped to a given range of cache sets, *bad* descriptor placement may cause additional external fragmentation. Consider, for example, an allocator state as sketched in Figure 4.4 and an open allocation request for $s$ bytes starting at a memory location mapped to cache set $k$.

In the depicted state sketch, a single, badly placed descriptor prevents that a free memory block of suitable length starting at a location mapped to the desired

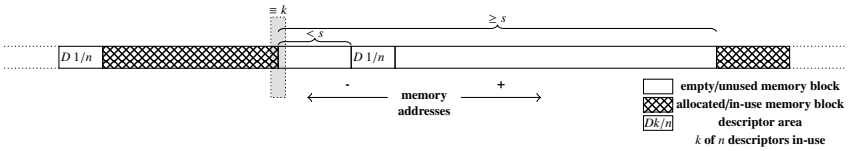



Figure 4.4: Allocator state where descriptor placement may be causing additional external fragmentation.

cache set (gray-shaded area) is available. While moving this descriptor to the left depicted descriptor area would solve this problem, moving descriptors at allocation time is not desirable. Technically, we would either need a search over all descriptor areas to find a free descriptor slot to which to move a badly placed descriptor. Or we need to keep pointers to all only partially used descriptor areas. And what if not moving one, but two or even more descriptors enables the allocator to satisfy a pending request using just its currently managed memory range? Obviously, allowing descriptor relocations at allocation time would severely impair timing predictability of this operation.

To circumvent or at least counteract these problems, we propose to implement a descriptor placement policy that strives to prevent such situations from developing. This can be achieved by requiring the descriptor placement policy to guarantee the following invariants.

IV1 No two consecutive free memory blocks of any kind exist. I.e., a free descriptor area is merged with adjacent free memory blocks whenever possible.

IV2 At most one descriptor area is not fully used, i.e., contains currently unused descriptors.

IV3 Each descriptor area contains the descriptor by which the area itself is managed.

IV1 ensures that free descriptor areas are recycled, thus situations as depicted in Figure 4.3 cannot occur. IV2 minimizes the number of descriptor areas used

at any time. Consequently, the probability of having a descriptor area *blocking* an allocation as depicted in Figure 4.4 is decreased. The impact of IV3 on fragmentation is less strong compared to IV1 and IV2. It prevents states as sketched in Figure 4.5 where an additional, actually unnecessary descriptor area is maintained due to a cycle where one area stores the descriptor to manage the other and vice versa. However, IV3 allows for a more efficient way to implement

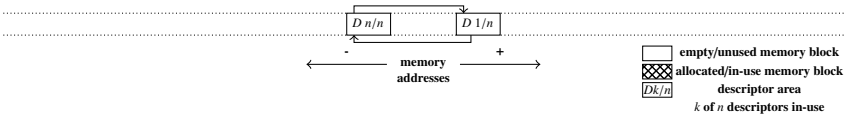



Figure 4.5: Unfavorable placement of descriptors resulting in the allocation of an additional descriptor area.

our allocator. Without IV3, the situation that a descriptor area is managed by a descriptor it stores itself would constitute a special case that both allocator interfaces, malloc and free, would have to consider. Maintaining IV3 as an invariant makes this *self-management* the only valid case, allowing for more efficient and clearer implementations. Furthermore, for the free operation, we can use IV3 to guarantee fewer descriptor relocations in the worst case. Which consequently yields a smaller WCET for this operation as well as a more restricted effect on the respective cache state. Thus, it increases overall predictability.

How can we maintain these invariants and what are the worst-case allocator-states during allocation and deallocation operations?

Maintaining IV1 through IV3 when processing an allocation request is fairly straightforward. In the worst-case, a suitable memory block needs to be extracted from splitting a memory block newly requested from the operating system. In this case, at most four additional descriptors are needed to manage the four new blocks created. As one of those blocks is a descriptor area, its descriptor will reside within that newly created area to maintain IV3. Figure 4.6 illustrates this case.

As no free blocks are created, IV1 will still hold if it held before the split operation. Given that IV2 held before, we either have a descriptor area that still

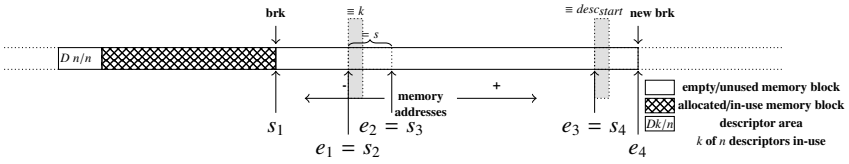



Figure 4.6: Allocator state where an allocation for $s$ bytes starting from an address mapped to cache set $k$ causes a maximal number of split operations. Four new memory blocks will be created in order to satisfy such a request while keeping invariants IV1 to IV3 intact. Each new block $i$ spans from $s_i$ to $e_i$.

has free descriptors we can use. Or all descriptor areas are fully used (as depicted in Figure 4.6). In the latter case, we create a new descriptor area from which to allocate the required descriptors for the newly created memory blocks. In case a descriptor area exists with unused descriptors left, we allocate as many of the descriptors we require from there to keep IV2 intact. If we cannot allocate all required descriptors from an existing area, we still have to create a new area from which to allocate the remaining descriptors. Creating new descriptor areas is trivially implemented in such a way that their management descriptor resides within the area itself, thus maintaining IV3.

Figure 4.7 depicts an allocator state that may yield a worst-case behavior when processing a deallocation request.

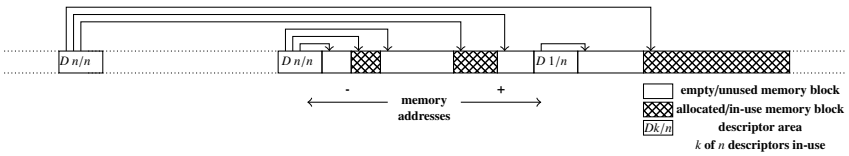



Figure 4.7: An allocator state where a deallocation of either of the two leftmost in-use blocks results in the worst-case number of four merge operations.

A worst-case situation arises when either one of the two left-most in-use blocks gets deallocated. In both cases, we end up with three adjacent free blocks: the currently freed block and its two already free neighbors. To maintain IV1, CAMA has to merge those three blocks into a single free block. IV1 also guarantees that three is the maximum number of free adjacent blocks after a block becomes free: as two adjacent free blocks will be directly merged into a single free block, free blocks must always be separated by a non-free block. When such a non-free block becomes free, the largest sequence of adjacent free blocks that may be created encompasses three blocks. However, merging three blocks also frees two descriptors. This in turn may render a descriptor area completely empty; either directly or after relocating used descriptors to maintain IV2. In our example, the rightmost descriptor area will be completely depleted once the descriptor it holds is relocated. Consequently, another merge of three free blocks is triggered. Leading to the worst case of two merges of three adjacent blocks, each. Comparing CAMA to `TLSF` with its similar merging strategy, we observe that CAMA needs exactly the double amount of merge operations in the worst case: four instead of two. This is exactly what one would expect, given that both allocators maintain that a maximal sequence of free adjacent blocks is at most of length one. But a deallocation in CAMA may free two memory blocks: the block to be deallocated itself and a depleted descriptor area; instead of just the memory block itself with `TLSF`. Also note that we implicitly assume a minimum descriptor area size of 3 descriptors per area to guarantee that at most one descriptor area can get freed during a deallocation.

**Counteracting Incomplete Memory Use** An implementation of an allocator as described so far would unfortunately introduce *incomplete memory use* as a third source of memory fragmentation. Originally, the term incomplete memory use was coined to describe `Half-Fit`'s inability to find free blocks larger than the base size of their respective free list to serve requests for blocks of sizes larger than this base size; even if they are just one byte larger. This was due to the allocation routine starting its search at the smallest size class of which all blocks are at least of the requested size, i.e., the next bigger size class than the one in which the requested size lies. As discussed in Chapter 2, simply rounding

all requests and hence block sizes to the next size class solves this problem and transforms incomplete memory use into (also highly predictable) internal fragmentation. In CAMA's case, however, incomplete memory use is caused by the lookup mechanism *starting at* the requested cache set. I.e., only size classes whose free blocks have a starting address with the given cache set mapping are considered. Larger blocks whose starting addresses are mapped to another cache set, but nonetheless contain a subblock with the requested cache set mapping and sufficient size cannot be found.

With regard to CAMA's worst-case memory consumption, we can exploit this incomplete memory use—as we did in the discussion of `Half-Fit`—to hide free, suitable memory blocks from the allocator. And hence construct a (de-)allocation sequence that forces the allocator to use an unbounded amount of memory for a fixed, constant amount of live memory. Even worse, we can even show that incomplete memory use due to restricting the allocator's lookup mechanism to free blocks starting at a given cache set causes severe memory waste for common (de-)allocation patterns as well as real-life programs [HH13] (see also Chapter 5 for a more detailed analysis of the sources of fragmentation for different allocators and (de-)allocation behaviors).

With incomplete memory use bearing the potential to cause a prohibitively large memory consumption for many (de-)allocation sequences, we propose the following adaption of CAMA's allocation routine. In situations where CAMA fails to find a suitable, free memory block starting at the requested set to satisfy an allocation request, instead of directly getting more memory from the underlying operating system, we propose to try a constant-time search for a free block that can be split to yield a suitable one with the requested cache set mapping.

To allow for a constant-time search, we let the allocator manage a second bit sequence where, again, a set bit indicates that the free list of the corresponding size class is non-empty. When scanning this sequence, we are interested in only the larger size classes, but associated to any cache set. As we can freely choose what we consider a *larger size class*, we can restrict the size of this bit vector such that the additional runtime costs for reading this sequence do not overly increase the WCET of CAMA's allocation routine. We also want an ordering on the bits such that picking the first block from the free

list associated with the first set bit that we encounter constitutes a reasonable choice.

Intuitively, as we are allowed to just check a single free block for its capacity to fulfill a given request, we strive to increase the probability for this check to turn out positive. We do so by trying to select the largest free block with a starting address mapped to a cache set *smaller*, but as close as possible, to the requested set.

Formally, we build a suitable bit sequence $v_\star$ as follows.

$$v_\star = \left\langle fb_{i,j} \right\rangle, i_{lmin} \leq i \leq i_{max} \wedge 0 \leq j \leq j_{max} + 1$$

where we use $i_{lmin}$ to formally define the term *large size class*: a size class $I_{i,j}$ is considered to be a large size class if and only if $i \geq i_{lmin}$. Furthermore, each subsequence $fb_{i,j} \in \{0,1\}^{sets}$ contains one bit per cache set $k$ (in descending order) set to 1 if and only if the corresponding free list is non-empty. I.e., if the $n^{th}$ bit of $fb_{i,j}$ is set to 1, then $S_{sets-1-n,i,j} \neq \emptyset$ holds.

When an allocation request for *size* bytes cannot be satisfied from a free list of blocks with starting addresses mapped to the requested cache set, $v_\star$ is scanned in the intuitive manner. The subsequences $fb_{i,j}$ with $size \in I_{i,j}$ are read until a set bit is found. This bit is associated with a non-empty free list containing the largest currently free blocks. Furthermore, if several such free lists exist, due to the reverse ordering on the bits of the $fb_{\cdot,\cdot}$, the first set bit corresponds to the free list containing blocks whose starting address is mapped to a cache set closest to the requested set. The first block of the associated free list is then considered for satisfying the pending request.

Setting $i_{lmin}$ such that every size class greater or equal to $size_{bmin}$ is considered during the search for a suitable free block, eliminates incomplete memory use completely (assuming block sizes are rounded and size classes degenerated to sets of equally sized blocks). However, the smaller $i_{lmin}$ is chosen, the higher the worst-case execution time of the allocator's allocation routine. For an architecture with 128 cache sets, 4 additional bytes per considered size class need to be read in the worst case. But the probability that a found block can be used to satisfy a request decreases with decreasing size classes. Hence, it may often be desirable to choose a more restrictive $i_{lmin}$ in order to find a reasonable trade-off between

worst-case execution times and memory waste due to incomplete memory use.

**Summarizing our Proposed Cache-Aware Memory Allocator CAMA** On a higher level of abstraction, what do we end up constructing so far? In summary, CAMA constitutes a constant-time multi-level segregated-list allocator with an indirect in-heap management of memory. Free memory blocks are organized in segregated free-lists which are addressed using a multi-level addressing scheme. The first layer groups free lists according to the cache sets the free memory blocks start in. While two more layers build size classes to group memory blocks according to their block sizes. Allocation and deallocation is mostly reduced to computing the address of a suitable free-list to allocate a block from and deallocate a block to, respectively. Such a computation can be performed in constant time. Furthermore, CAMA implements splitting and merging operations to counteract fragmentation. These are also constant time operations with a tightly bounded number of memory accesses directed to a memory area mapped to a statically strictly bounded portion of the cache. This restriction of accesses to a given part of the cache only is enabled by CAMA's indirect block managing scheme using descriptors with a statically predefined cache set mapping to manage memory blocks.

How well does such an allocator meet the criteria for a dynamic memory allocator suitable for hard real-time systems that we established at the beginning of this chapter?

**Constant-Time (De-)Allocation Routines** Even in the worst case, all internal procedures proposed for usage in a predictable allocator (free-list lookups, splitting at allocation, merging at deallocation) can be tightly bounded by a constant number of operations. We refrain from giving concrete worst-case execution times in processor cycles for procedures as they greatly vary depending on the target platform and CAMA's configuration. WCET bounds also depend significantly on the implementation of the allocator, further decreasing the information value of concrete processor cycles. In consequence, we cannot be sure whether the complexity of the algorithms or a too unpredictable implementation thereof is responsible for a concrete WCET bound. Furthermore, the worst-case execution

times of procedures are in general not compositional, i.e., we cannot simply add up the execution times of single subprocedures (bit vector scans, index tuple computations, and so on) to yield the worst-case execution time of an entire (de-)allocation.

The following listing gives a summary that may also serve as a worst-case count of operations for the single procedures.

1. At allocations

    a) Free-list lookup: In the worst case, all free lists associated with the requested cache-set mapping are empty. In this case, we need to compute the index triple and, during the first search phase, read at most $2 + \lfloor (j_{max} + 1)\,(i_{max} - i_{min})/\,32 \rfloor$ words (of 32 bits) to *verify* that no free block with a suitable starting address exists.

    If CAMA is configured to perform a subsequent worst-fit approach to find a biggest block that may be split to yield an appropriate block to satisfy the request, we need to scan another bit vector. In the worst case, this second round will need to read another, additional bit vector of $\frac{sets}{32} \cdot (j_{max} + 1)\,(i_{max} - i_{lmin})$ words.

    b) Splitting of a larger than required block: In the worst-case, the second search found a block that needs to be split into three blocks, where the middle block is suitable to satisfy the original request. Hence, we need to write three descriptors and, again in the worst-case, create a new descriptor area including writing a fourth and fifth descriptor to manage said area as well as a newly created memory gap left of that area.

2. At deallocations

    a) Checking free-bits of memory-adjacent blocks: To check whether the newly freed block's adjacent memory blocks are also free and a merge is required, we need to read two entries of the newly freed block's descriptor (linked from the (header of) the block itself) as well as the size fields of the descriptors of the adjacent blocks.

b) Merging the maximal number of adjacent free blocks: In the worst case, the newly freed block is located between two already free blocks which results in two merge operations to create one new, continuous free block spanning these three blocks. Furthermore, after merging these, we free two descriptors which may lead to a completely free descriptor area that again may be located between two currently free memory blocks. Which again results in two merge operations. Note that merging cannot cascade due to invariants IV1 to IV3 coupled with the minimum size of the descriptor areas of larger than three.

Please note that, while we argue against the informative value of concrete execution times, we have derived such values for specific architectures and parameters before [HBHR11, HH13].

In [HBHR11], we derived WCET bounds considering as target platform the PowerPC MPC603e. This hardware is widely used in embedded systems. It possesses separate data and instruction caches with 128 cache sets each and a cache line size of 32 bytes. The instantiation of CAMA that we considered did not use area blocks, so the WCET bounds we obtained give a good impression on the (execution time) costs of using descriptors and a three-level lookup mechanism (as opposed to TLSF's two-level approach). For allocation requests, we were able to derive upper bounds of 9,935 cycles and 16,260 cycles for CAMA and TLSF, respectively, using aiT. This result is a bit surprising, given that CAMA's algorithm is more complex than that of TLSF. By simply implementing TLSF's internal computation of logarithms in a more predictably way, we were able to reduce TLSF's bound to 13,026 cycles. This, of course, strengthens our argument against the usefulness of processor cycles at this point. Furthermore, free operations can be bounded by 6,891 and 5,703 cycles for CAMA and TLSF, respectively.

The WCET bounds given in [HH13] compare the allocation routine of TLSF to that of CAMA with area blocks and a small second bit search to counteract incomplete memory use of 128 considered free lists. The considered target platform is again the PowerPC MPC603e. Figure 4.8 summarizes the results given in [HH13] in two scatterplots. As we can see in those plots, upper bounds on the execution times are, in this case, very similar for both allocators. Despite

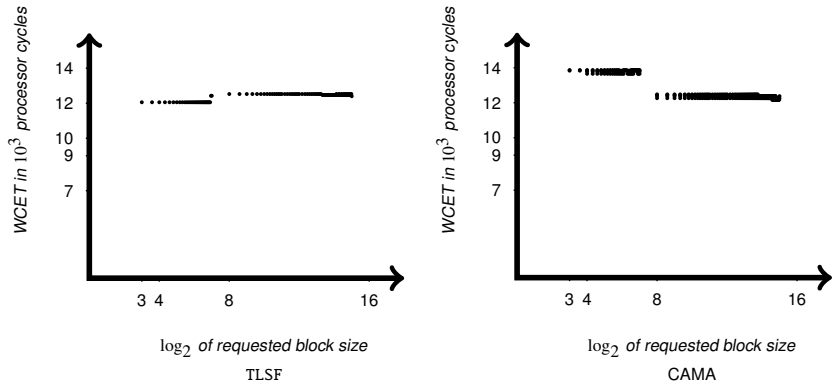



Figure 4.8: WCET bounds of `TLSF` and `CAMA` (using areas for small blocks and a 128 bit bad-fit fallback).

the significantly more complex algorithms of CAMA (compared to `TLSF`), the provable bound on its WCET rises to just 13,867 cycles. Additionally, we could decrease the WCET bound of `TLSF`'s allocation routine to 12,533 cycles with a newer, more precise version of `aiT`. Again, we may still largely overestimate the WCET of `TLSF` due to a less predictable programming style used in the implementation of `TLSF`. Note that the *jump* in the derivable WCET bounds for CAMA for the different requested sizes is attributed to the more complex management of smaller blocks in areas.

**Predictable Cache Behavior** The main goal, allowing to guide dynamic allocations with respect to the cache-set mapping of allocated blocks, is achieved by construction: CAMA's allocation routine takes the desired cache-set mapping as an additional parameter. Furthermore, its real-time properties with respect to providing constant-time (de-)allocation routines is evidenced in the previous paragraph. But how well does CAMA do with respect to its overall cache predictability, i.e., the predictability of the effects of CAMA's internal routines on the cache state? CAMA's allocation routine may access its internal bit sequences, its free list table, and descriptors. All accesses to descriptors are directed to a small,

statically known number of cache sets. We may also store CAMA's bit sequences and free list table at memory locations mapped to the same cache sets. In this case, when invoking the allocation routine, only the small set of cache sets to which descriptors are mapped may be affected. Furthermore, the number of pairwise different accesses to these sets (w.r.t. cache lines) may then, at least for current hardware, potentially exceed the number of cache lines. Hence, simply dropping information about these cache sets in the current cache state is an easy, precise enough way to update the cache state when our allocator's allocation routine is invoked. CAMA's deallocation routine may only access descriptors and its free list table and can consequently be handled the same way.

**Memory Consumption** As already discussed in Chapter 2.2, static worst-case bounds on the memory consumption of a dynamic memory allocator are bound to be rather discouraging. For CAMA, such a bound is heavily dependent on whether or not the allocator is configured to be prone to incomplete memory use. If a configuration is chosen in which incomplete memory use may occur, CAMA's worst-case memory consumption is unbounded. Nonetheless, such a configuration may in practice be a good choice if one can prove that the applications using CAMA do not exhibit (de-)allocation sequences that cause overly high memory waste due to incomplete memory use.

If we configure the allocator such that incomplete memory use cannot occur, CAMA selects blocks either to a best-fit (a suitable free block is found in the free lists associated with the requested cache set) or bad-fit (a suitable, large block is found that *spans* over the requested cache set) strategy. Hence, external fragmentation can be bounded by the bounds known for these strategies: $\mathcal{M} \cdot m$, where $\mathcal{M}$ is the sum of the requested memory and $m$ the maximal size of an allocation.

Overhead for descriptors as well as the predefined cache-set mapping can be attributed to internal fragmentation. In the worst case, a block of size $s$ with a starting address mapped to a cache set to which descriptors are mapped is requested in a state where no free descriptors are available. In this case, we may need to allocate $s + size_{way}$ bytes to create such a block, plus a whole cache page for a new descriptor block area with the correct cache-set mapping.

Overall, memory usage to satisfy such a request for $s$ bytes sums up to $s + 2 \cdot size_{way}$ bytes. Hence, maximal internal fragmentation for requests for a total of $\mathcal{M}$ bytes is bounded by $\left(1 + 2 \cdot size_{way}\right) \cdot \mathcal{M}$. And the overall memory consumption including external and internal fragmentation is consequently bounded by $\left(1 + 2 \cdot size_{way}\right) \cdot \mathcal{M} \cdot m$, where $m$ is the maximal size of an allocation. The additional memory need for CAMA's descriptors as well as *memory holes* due to the enforced cache-set mapping of dynamically allocated blocks to which the allocator must adhere are both attributed to internal fragmentation.

Comparing the upper bounds on CAMA's and TLSF's memory consumption nicely reflects their respective design choices. As shown, TLSF's worst-case memory consumption is bounded by

$$\underbrace{\frac{\mathcal{J}+1}{\mathcal{J}}}_{\text{upper bound on internal fragmentation per block}} \cdot \underbrace{\underbrace{\mathcal{M}}_{\text{upper bound on memory blocks}} \cdot (m-2)}_{\text{upper bound on memory usage incl. external fragmentation}}$$

The first factor bounds internal fragmentation dependent on the number of second-level size classes. This factor covers memory waste due to rounded block sizes. The second and third factor bound the worst-case memory use due to maximal external fragmentation. TLSF selects blocks (of rounded sizes) to a best-fit strategy. Hence, we can sharpen the trivial general bound on external fragmentation of $\mathcal{M} \cdot m$ to the bound on external fragmentation for best-fit allocators, which is $\mathcal{M} \cdot (m - 2)$.

In CAMA's case, we showed a bound of:

$$\underbrace{\left(1 + 2 \cdot size_{way}\right)}_{\text{upper bound on internal fragmentation per block}} \cdot \underbrace{\underbrace{\mathcal{M}}_{\text{upper bound on memory blocks}} \cdot m}_{\text{upper bound on memory usage incl. external fragmentation}}$$

Again, the first factor bounds internal fragmentation. In CAMA's case, this factor is significantly larger to cover for memory wasted to adhere to the given cache-set mapping and to store descriptors. As in TLSF's case, the second and third factor bound the worst-case memory use due to maximal external fragmentation. CAMA, however, employs either a best-fit (a suitable block with the requested cache-set mapping exists) or a bad-fit (a block with a suitable cache-set mapping needs to be

created first) strategy to selecting free blocks. Hence, the general bound of $\mathcal{M} \cdot m$ cannot be sharpened in CAMA's case.

**Concluding Remarks** In conclusion, we consider our design goals for a cache-aware, predictable memory allocator to be met by CAMA. However, we do also consider CAMA to be more of a set of cache-aware techniques than an allocator that can be used out-of-the-box. The more suitable CAMA is configured for a given application, i.e., the more appropriate one sets size classes, sizes of descriptor areas, depth of bit vector scans, and so on, the higher is its predictability and the better its performance.

**A Classification of CAMA According to Wilson et al.** In [WJNB95], Wilson et al. propose to classify allocators according to a three-level distinction of *strategies*, *policies*, and *mechanisms*. The authors see an allocator as a set of strategies to decrease memory fragmentation that exploit regularities of programs, or rather that attempt to avoid pitfalls stemming from those regularities. A policy is an implementable decision procedure determined by a strategy. A mechanism is a set of algorithms and data structures that actually implement a policy. In those terms, CAMA follows three strategies.

1. *Avoid searching for suitable blocks*, which aims at constant response times rather than low memory usage.

2. *Never cause unpredictable memory accesses*, which aims at cache predictability.

3. *Avoid letting small, long-lived blocks prevent the reclaiming of larger, contiguous free memory areas*, which aims at low memory use.

The corresponding policies concretize these strategies into implementable procedures.

1. *Sort free blocks into reasonably small equivalence classes such that a given allocation may be satisfied by any block from this class, rendering unnecessary any (further) searching within an equivalence class* and *manage*

*such classes in a way that allows for a constant-time mapping of allocator requests to suitable classes*.

2. *Organize equivalence classes in such a way that the cache set mapping of all potentially accessed data fields is statically known*.

3. *Always take the first block from the smallest, non-empty class that contains only blocks that are guaranteed to be large enough to satisfy a request* and *always split blocks when a larger than needed block was used*.

The following mechanisms implement these policies.

1. Reasonably sized equivalence classes are built by our three-leveled approach. Blocks of the same class are organized in a singly-linked list. The segregated, singly-linked lists are organized in a multidimensional array such that the suitable list for a given (de-)allocation can be easily computed and accessed in constant-time.

2. A statically known cache set mapping of accessed data fields is achieved by employing descriptors. I.e., an indirect management of free blocks instead of the traditional direct management at the blocks themselves.

3. CAMA strives to always take a smallest block to satisfy requests including internal *requests* for descriptors by employing a good fit strategy and our proposed descriptor placement strategies, respectively.

## 4.3 An Alternative Approach to Guiding Cache-Set Mappings of Allocations: `RelCAMA`

CAMA modifies the standard interface to the memory allocator by adding an additional argument to the allocation routine: the cache set to which the starting address of the returned memory block shall be mapped. This design choice was clearly motivated by aiming to support a subsequent cache analysis with as precise information about cache set mappings as possible. However, as discussed in

Section 2.4.2, Hahn and Grund show that for a precise cache analysis, knowing the relations between accessed memory blocks is sufficient.

Our implementation of CAMA currently features an alternative allocation interface aimed at supporting relational cache analyses. This alternative interface has the following function signature

```
void* carelmalloc(size_t size, enum relation_t rel, ...);
```

Invoking this allocation routines returns a pointer to a free memory block of at least `size` bytes with a starting address with relation `rel` to all addresses in the null-pointer terminated list of pointers following the first two arguments (`...`).

What relations can be requested? In the lattice of relations used by Hahn and Grund (see Figure 2.6, page 59), the most precise and hence desirable relations are the *same block*, *same set but different block*, and *different set* relations. For the first two relations, the *same block* and *same set but different block*, an allocator can in general not guarantee to find such a block in constant time. Mapping newly allocated memory to a given block may simply not be possible due to the block already being full. Finding a block with the same cache set mapping as a given block, but not being the same block, can be implemented as a constant time operation, but would further add to the complexity of CAMA. The *different set* relation, however, may be guaranteed as such a block can always be created assuming more memory can be obtained from the underlying operating system. Besides this relation, also the *same set* relation can be generally guaranteed by an allocator and is consequently supported by CAMA.

While this interface also breaks with the standard aim of keeping caches transparent to the application (as do other, existing cache-conscious allocators), we note several advantages of the relational allocation interface. Although both arguments (fixed cache set or fixed relation) can be determined by a static preanalysis or via simple heuristics [HRW08], when done manually, good relations may be easier to find for a programmer than good fixed cache sets. From a more technical point of view, just having to adhere to a fixed relation instead of a fixed cache set mapping imposes fewer restrictions on the blocks the allocator may return to fulfill an allocation request.

Many adjustments of CAMA's allocation strategy are conceivable and may

potentially further decrease CAMA's memory consumption. While the *same set* relation is as restrictive as a fixed cache set, the *different set* relations may allow the allocator to choose from a large set of cache sets to which a returned block must be mapped. In this case, the allocator can switch its allocation strategy to the following heuristics that may often decrease memory consumption in practice:

- *Always return a block from a cache set with the highest number of currently free blocks.*

  This strategy aims at counteracting imbalances in the request frequencies of the different sets which may otherwise lead to an increased memory consumption.

- *Always return a best fitting block by choosing the cache set accordingly.*

  This strategy aims to (further) mimic a best-fit strategy which has been shown to often yield the lowest memory consumption in practice as discussed in Section 2.2.

## 4.4 An Alternative Approach: PRADA

Compared to other dynamic memory allocators, CAMA's algorithms are rather complex, mainly due to its indirect management of blocks. This complexity prevents simple and efficient implementations. At least, this puts CAMA at a disadvantage when looking at best-case or average-case execution times. This also raises the question, whether there are simpler ways to cache-aware, predictable dynamic memory allocation that do not rely on a complex indirect memory management.

In [Hau12], PRADA, an alternative approach to predictable and cache-aware dynamic memory allocation is described. PRADA uses the same three-level approach to set up segregated free lists to manage free memory blocks as CAMA. Hence, constant time allocation and deallocation routines are achieved in the same manner. Unlike CAMA, PRADA builds those free lists in the standard, straightforward way: using the free blocks themselves to store pointers to the free blocks preceding

and succeeding the current block within its free lists. Still, split and merge operations may need to access those data fields. As it is statically impossible to precisely derive when splits and merges are executed, storing information directly at the free blocks themselves leads to unpredictable cache accesses, corroding information about the cache state. To circumvent this problem, PRADA defers all actions, i.e., split and merge operations, which would introduce unpredictable behavior when always immediately executed. Instead of a guaranteed immediate execution of such operations, PRADA stores them as *action requests* first. The actions themselves are executed during subsequent (de-)allocations directed to exactly the cache set those operations need to access. Hence, PRADA can guarantee that during (de-)allocations internal actions access only the cache set to which the (de-)allocated memory block is mapped.

While we omit a detailed description of the PRADA algorithm and refer to [Hau12] for more details on that topic, we still highlight the differences between the two allocators in the remainder of this section.

CAMA maintains one descriptor per managed memory block to enable cache-predictable split and merge operations for this block. In case such an operation is never applied to a block, this block's descriptor can be considered an unnecessary management overhead. PRADA, however, creates its internal action request only in cases in which an action actually can be applied. Hence, there is less potential for unnecessary management overhead in this allocator. However, the number of requests that PRADA can contemporaneously store is limited to a small, statically fixed number. If at any point the maximal number of action requests is reached, further requests are simply dropped and never executed. This has the obvious disadvantage that for certain request sequence splitting and merging gets deactivated, although memory usage may significantly profit from those operations. Consider for example a program that performs an in-situ copy of one of its data structures. Splitting and merging may enable efficient memory use in such a scenario. This is achieved either by allowing the program to merge and reuse memory locations from the source structure to hold elements of the destination structure, in case a smaller data structure is transformed into a larger one. Or splitting free memory locations from the source structure to hold elements of the destination structure, in cases where the source structure's elements are of a larger type than the destina-

tion's. If this example program, however, fills up its request queue before its data structure transformation, these opportunities to efficiently reuse memory are lost. Still, for other request sequences, PRADA's delayed application of operations may, however, be beneficial. Consider the following (de-)allocation sequence

$$\left(\alpha_s^x \alpha_s^y \delta^x \delta^y\right)^+$$

If the memory blocks served in the two allocations are adjacent in memory, a non-deferred merge will result in an unnecessary merge operation after the second deallocation and, in consequence, an additional split at the following allocation (when the sequence is repeated).

Hence, PRADA's overall performance highly depends on a suitable choice of the number of requests that can be contemporaneously stored.

To justify PRADA's approach and expect it to perform reasonably well in practice even with only a constant, fixed number of contemporaneous split and merge requests, one needs to make two assumptions regarding regularities in real-life programs. Namely that real-life programs do not feature sequences that lead to the dropping of beneficial actions and that there are programs in which merges are often followed by split operations of the same block. In the latter case, allocator performance will increase when these merges were deferred until they become obsolete and are consequently dropped. The latter regularity, at least, has been often observed [WJNB95].

In terms of the classification proposed by Wilson et al., PRADA is motivated by the same strategies as CAMA. However, the policy designed to implement the *Never cause unpredictable memory accesses*-strategy is different. PRADA's policy for ensuring the absence of unpredictable memory accesses, or better accesses to memory locations with a statically unknown cache set mapping can be summarized as:

- *Defer memory accesses until they are directed to memory locations with a statically known cache-set mapping; even if this entails splitting an operation into several micro steps or possibly never executing an operation at all.*

**Concluding Remarks** CAMA's indirect memory management via descriptors is, arguably, the greatest drawback of our approach as it makes implementations rather complex. PRADA's additional management overhead, its action requests, are, in contrast, rather trivial to implement. Action requests do also not influence external fragmentation (we evidence the influence of CAMA's descriptor areas on overall external fragmentation in Chapter 5).

Does this mean PRADA may be the better, more promising approach? Besides bearing the risk to drop useful split and merge operations, we note another source for overly high memory consumption with PRADA. Unlike in CAMA, with PRADA there is no obvious way to counteract incomplete memory use. Unfortunately, incomplete memory use occurs rather often and drastically in allocators using our proposed free list structure [HH13, Hau12] (see also Chapter 5). Hence, in general, CAMA must be considered the more robust and more promising approach after all.

In order to get a better impression on PRADA's memory performance, we included two instantiations of the allocator in our evaluations. We discuss our findings in Chapter 5.

# 5 Experimental Evaluation

> The fundamental principle of science, the definition almost, is this: the sole test of the validity of any idea is experiment.
>
> — Richard P. Feynman
> *Physicist (1918–1988)*

## 5.1 Chapter Overview

In Section 2.2, we discussed the general issue that for all dynamic memory allocators there exists a worst-case (de-)allocation sequence that forces the allocator to use an overly large amount of memory. This amount of memory is often close to the absolute maximum for allocators without incomplete memory use of $I \cdot \mathcal{M} \cdot m$. Again, $\mathcal{M}$ denotes the sum of all requests, $m$ the largest block size manageable by the allocator, and $I$ captures memory waste due to internal fragmentation. The resulting worst-case bounds on an allocator's memory consumption have in practice been shown to be overly pessimistic, as real-life programs do in general not feature these worst-case (de-)allocation sequences. CAMA, obviously, does not make an exception to this end. Hence, in order to argue CAMA's usefulness, we have to

provide sufficient evidence that for real-life programs—or better: (de-)allocation sequences that may occur in practice—a reasonable memory consumption is achieved.

This chapter reports in detail on the benchmarks and measurements we performed in order to show that CAMA does indeed achieve predictability without incurring a prohibitively large memory overhead. The chapter is structured as follows. Section 5.2 elaborates on our choice of allocators and metrics used to compare CAMA against. Section 5.3 evaluates these allocators on synthetic, randomized (de-)allocations sequences that model typical behaviors expected of real-time applications. In Section 5.4, a small set of existing real-time applications is used to benchmark our set of allocators.

## 5.2 Allocators and Metrics Used in our Benchmarks

We used the following (instantiations of) dynamic memory allocators and metrics in our comparisons.

1. CAMA (two instantiations, with and without counteracting incomplete memory use by a bad-fit fallback mechanism),
2. `PRADA` (two instantiations, one configuration with splitting and merging disabled),
3. Cache-Set-Guided Address-Ordered Sequential-Fit Allocators with best-fit, first-fit, and worst-fit policies,
4. `TLSF`,
5. `DLMalloc`,
6. the sum over all allocation requests ($\mathcal{M}$), and
7. the maximally live memory.

The following paragraphs elaborate on the choices of allocators used in our comparisons. They furthermore discuss what we intent to evidence by these different allocators or configurations. In those paragraphs, we also give the detailed configuration of each allocator that we used in our measurements.

**CAMA** We used two instantiations of CAMA: one instantiation where incomplete memory use is counteracted by a subsequent bad-fit approach if CAMA's standard best-fit approach fails (denoted CAMA) as well as an instantiation where the allocator relies on its best-fit approach only (denoted by CAMA`0`). For the bad-fit approach, all size classes containing blocks of sizes greater than or equal to 1024 bytes are considered.

The instantiation of our proposed allocator that we denote just CAMA constitutes a reasonable tradeoff between (worst-case) execution times and memory usage. CAMA`0` evidences the severity of incomplete memory use resulting from our proposed best-fit lookup mechanism that considers only free blocks with a starting address mapped to the requested cache set and ignoring splittable, larger blocks with a different cache-set mapping but spanning a suitable memory range. The detailed configurations of those two allocators are given in Table 5.1 and Table 5.2, respectively.

**PRADA** We also added two versions of `PRADA` to our benchmarks. `PRADA32` is configured to store a maximum of 32 contemporaneous requests per cache set (for splitting/merge operations plus 32 so-called top-level requests), `PRADA0` cannot store any requests. By setting the number of requests to 0, we aim to get some insights into the importance of being able to split and merge to counteract high memory fragmentation. The selection of 32 requests per cache set seems rather arbitrary at first. However, this number is motivated by two observations. In previous benchmarks, we observed that 32 contemporaneous requests are sufficient in all test cases such that no request is ever dropped [HH13, Hau12]. Furthermore, this selection allows for an efficient implementation of the `PRADA`'s lookup routine for a pending request. We can use the 32 bits of a 4 byte word to encode whether a reserved request slot currently stores a pending request.

| Parameter | Value | Remarks |
|---|---|---|
| Cache sets (*sets*) | 128 sets | |
| Cache line size | 32 bytes | |
| Cache way/page size | 4096 bytes | |
| $i_{min}$ | 2 | |
| $i_{max}$ | 21 | |
| First-level size classes | 19 | |
| Second-level size classes | 4 | |
| Size classes | 76 | 3 to 5 words are read during the best-fit phase |
| $size_{min}$ | 160 bytes | Blocks larger than this value are managed by an own, exclusive descriptor. |
| $size_{lmin}$ | 1024 bytes | All size classes with blocks larger than this value are considered during the bad-fit phase. May lead to 3 words per set to be read. |
| Descriptors per area | 13 | 320 bytes per cache way can be used for descriptors. |

Table 5.1: Configuration of CAMA used for benchmarking.

| Parameter | Value | Remarks |
|---|---|---|
| Cache sets (*sets*) | 128 sets | |
| Cache line size | 32 bytes | |
| Cache way/page size | 4096 bytes | |
| $i_{min}$ | 2 | |
| $i_{max}$ | 21 | |
| First-level size classes | 19 | |
| Second-level size classes | 4 | |
| Size classes | 76 | 3 to 5 words are read during the best-fit phase. |
| $size_{min}$ | 160 bytes | Blocks larger than this value are managed by an own, exclusive descriptor. |
| $size_{lmin}$ | $\infty$ | Disables the bad-fit phase. |
| Descriptors per area | 13 | 320 bytes per cache way can be used for descriptors. |

Table 5.2: Configuration of CAMA0 used for benchmarking.

By comparing CAMA to PRADA32, we cannot isolate and compare the memory costs of using an indirect block management scheme with descriptors to those of using a direct managing scheme with delayed split and merge operations. Changes in memory usage due to a different degree of incomplete memory use caused by the presence of descriptor areas may be much more significant and superpose the changes due to an indirect memory management. We observed this effect in previous benchmarks [HH13, Hau12]. We can, however, evidence whether a simpler algorithm like PRADA that can be implemented much more efficiently than CAMA's indirect memory organization is sufficient for typical (de-)allocation sequences occurring in (hard) real-time systems. Furthermore, we can get insights about the degree of memory waste due to incomplete memory use from comparing the instantiations of PRADA to our cache-set-guided address-ordered sequential-fit allocators.

The detailed configurations of the two instantiations of PRADA are given in Table 5.3 and Table 5.4, respectively.

**Cache-Set-Guided Sequential-Fit Allocators** To isolate the memory costs for adhering to a statically predefined cache-set mapping of dynamically allocated data, we implemented three simple linear-time allocators. These allocators provide the same allocation interface with an additional cache-set argument as CAMA and PRADA. Technically, these allocators maintain a single free list in which all managed free blocks are stored. This list is ordered with respect to the starting addresses of the memory blocks. Hence, deallocation takes linear time (in the number of managed blocks), but merge operation can be efficiently implemented as constant-time operations. Upon an allocation request, these algorithms sequentially search their respective free lists to find a block that can be split to yield a large enough (sub-)block with the requested cache-set mapping. The instantiation denoted `aoff` always selects the first such block to satisfy the allocation request. Hence, `aoff` implements a sequential first-fit algorithm. `aobf`, in contrast, selects the smallest block that may be split to yield a suitable block to satisfy the request; `aowf` selects the largest such block. Hence, they implement sequential best-fit and worst-fit algorithms, respectively. If several smallest or largest blocks exist, both algorithms select the one with the smallest memory address.

| Parameter | Value | Remarks |
|---|---|---|
| Cache sets (*sets*) | 128 sets | |
| Cache line size | 32 bytes | |
| Cache page/way size | 4096 bytes | |
| $i_{min}$ | 5 | This allocator returns cache-set aligned blocks and always rounds block sizes to multiples of the line size ($2^5$ bytes). |
| $i_{max}$ | 24 | Set to 24 to result in a free-list table of the same size as CAMA's. |
| First-level size classes | 19 | |
| Second-level size classes | 4 | |
| Size classes | 76 | 3 to 5 words are read during the best-fit phase. |
| Max. no. of requests | 32 | |
| Max. no. of top-level requests | 32 | |

Table 5.3: Configuration of `PRADA32` used for benchmarking.

| Parameter | Value | Remarks |
|---|---|---|
| Cache sets (*sets*) | 128 sets | |
| Cache line size | 32 bytes | |
| Cache page/way size | 4096 bytes | |
| $i_{min}$ | 5 | This allocator returns cache-set aligned blocks and always rounds block sizes to multiples of the line size ($2^5$ bytes). |
| $i_{max}$ | 24 | Set to 24 to result in a free-list table of the same size as CAMA's. |
| First-level size classes | 19 | |
| Second-level size classes | 4 | |
| Size classes | 76 | 3 to 5 words are read during the best-fit phase |
| Max. no. of requests | 0 | Disables splitting and merging. |
| Max. no. of top-level requests | 0 | Disables splitting and merging. |

Table 5.4: Configuration of `PRADA0` used for benchmarking.

**Other Allocators and Metrics** In order to put the memory performances of our cache-set guided allocators into context, we also included well-known and widely-used allocators as well as two general metrics.

`TLSF` represents the current state-of-the-art of non-cache-aware, real-time dynamic memory allocators. In a similar evaluation of the memory consumption of `TLSF`, `Half-Fit`, and a binary buddy allocator, `TLSF` has been shown to perform strictly better than the other two [MRR+08]. For the test cases used in [MRR+08], memory waste was roughly ten times higher with `Half-Fit` and five times higher with a binary buddy, respectively, than with `TLSF`. On average, `TLSF` incurred a 9.73% rise in memory consumption in those benchmarks due to internal and external fragmentation. We use `TLSF` in its latest version, version 2.4.6, in its standard configuration with 24 first-level and 32 (4 bytes) second-level size classes.

`DLMalloc` denotes Doug Lea's memory allocator. This allocator represents the current state-of-the-art of general purpose dynamic memory allocators. In the evaluations reported on in [MRR+08], Doug Lea's allocator performed strictly better than the real-time allocators (`TLSF`, `Half-Fit`, and the binary buddy). On average, internal and external fragmentation summed up to a 8.77% increase in memory consumption in those benchmarks. Doug Lea's allocator is used in version 2.8.4 in our evaluation.

Furthermore, $\mathcal{M}$, again, denotes the sum over all requests blocks. We additionally recorded the maximal live memory for each test case. This value, denoted `Oracle`, is rather theoretical in nature. An allocator aiming to always allocate just the maximal live memory would have to (1) know all future requests, (2) be allowed to revoke placement decisions, i.e., move allocated objects in memory, and (3) would not be allowed to incur any management (memory) overhead. Obviously, requirements (1) and (3) are practically not achievable, and requirement (2) is incompatible with the C semantics.

## 5.3 Memory Performance for Random (De-)Allocation Sequences

For our evaluation, we generated several synthetic, randomized (de-)allocation traces that we consider representative for real-time applications. We agree with Wilson et al. that the regularities existing in real programs are not well enough understood to model them formally and perform probabilistic analyses that are directly applicable to real program behavior [WJNB95]. Nonetheless, this is still the best we can do as we lack representative real-life programs. With our synthetic traces, we follow the *typical* (de-)allocation patterns of programs discussed in [WJNB95]. The following paragraphs elaborate on those patterns and explain our instantiations of these patterns.

**The *ramp* pattern** The simplest (de-)allocation pattern consists of just a sequence of allocations. I.e., a program following this pattern allocates a certain amount of memory, performs its actual computations, and finally deallocates all dynamically allocated objects without any further allocation nor computation (or it might just terminate once its actual computations are completed). This pattern is known as a *ramp*. For our evaluation, we generated two synthetic traces implementing this pattern. The trace denoted *ramp small* consists of a sequence of 100,000 allocation requests. Each request is for one of five size classes and requests either 8, 12, 16, 20, or 24 bytes. The sequence is generated such that the numbers of requests for the respective size classes follow a binomial distribution. Figure 5.1 depicts the concrete frequencies of occurrences of requests per size class. With this instantiation of a ramp pattern, we strive to incorporate the often observed behavior of programs to just request a very limited number of different sizes [WJNB95]. Hence, our 5 size classes only. Furthermore, we assume that those size classes correspond to the different sizes of object types used within an application. Therefore, we chose those classes to be rather small as we would expect objects in a hard real-time application to be rather small, too. The binomial distribution is used to model that the different object types are created with a different frequency.

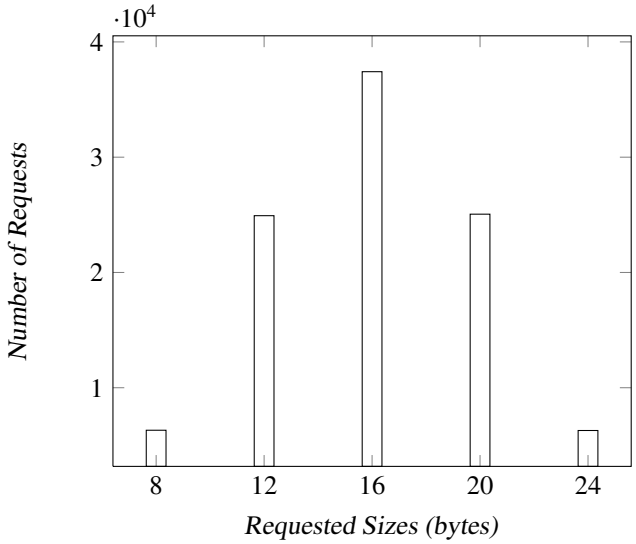



Figure 5.1: Bar plot of allocation requests issued in *ramp small*.

In the *ramp large* test case, we generated a smaller sequence of allocation requests but for larger sizes. Figure 5.1 depicts the concrete frequencies of occurrences of requests per size class. The 10,000 requests we generated adhere to a normal distribution with a mean of 256 words, i.e., 1024 bytes, and a 4 byte alignment. Figure 5.2 gives the scatter plot of the concrete frequencies of occurrences of requests per size.

**The *peak* pattern** Another typical (de-)allocation pattern is to allocate a certain amount of memory, perform some computations, deallocate all objects again, and repeat those steps. This behavior can often be observed in reactive systems. A reactive system typically produces an output upon receiving stimuli from within or outside the system. During the computation of a single task to produce such an output, objects might be allocated that become obsolete and are consequently deallocated once the output upon a given stimulus is computed. Upon the next stimulus, objects are allocated again and this procedure repeats. Again, we gener-

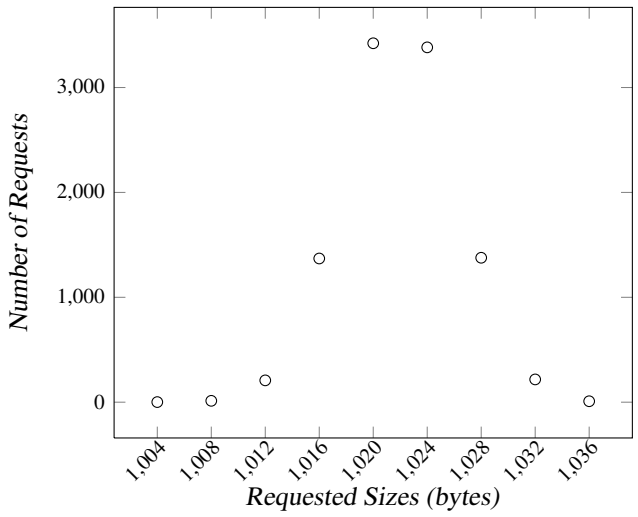



Figure 5.2: Scatter plot of allocation requests issued in *ramp large*.

ated two synthetic, randomized (de-)allocation traces modeling such a behavior. For the trace *peak small*, we generated and concatenated 100 subsequences (or rounds) consisting of 1,000 allocations for small objects with a randomized life time. Small objects were again of sizes 8, 12, 16, 20, or 24 bytes and selected to a binomial distribution. To determine an allocation's life span, we generate a random number adhering to an exponential distribution with parameter $\lambda = \frac{3}{1{,}000}$. I.e., the expected value of an object's life span is one third of the number of allocations per round. Our second peak trace, *peak large*, is set-up in the same way. We only changed the random number generator for requested sizes to the same normal distribution used for the *ramp large* test case.

As we would expect, from a certain point on, the number of rounds does not influence memory consumption any further. However, the overall requested memory does and we may of course generate any arbitrary ratio of memory requested vs. memory used by any allocator. We give the memory consumption after 100 rounds. Please also note that this pattern is typical for reactive systems that may virtually run forever.

**The *plateau* pattern** The third (de-)allocation pattern that we consider is a combination of the ramp and peak pattern. This pattern, known as a *plateau*, starts with a sequence of allocations (a ramp pattern) followed by sequences of allocations with subsequent deallocations (a peak pattern) on top of the allocations from the beginning. Aiming to capture another typical behavior of a real-time system, we generate two *plateau* patterns as follows. Both our sequences, *plateau small* and *plateau large* start with 100 allocations for small and large objects, respectively. To generate random requests for this part of the (de-)allocation sequences, we, again, reuse our random number generators producing binomially distributed numbers of sizes 8, 12, 16, 20, or 24 (*plateau small*) and normally distributed numbers with a mean value of 1024 (*plateau large*). For the second part of the *plateau* patterns, we use the processes for generating a synthetic, randomized (de-)allocation sequence used for the *peak* patterns. For our *plateau* patterns, we generate a *peak* with 100 rounds with 500 allocations, each. Once again, those allocations are for sizes of 8, 12, 16, 20, or 24 and according to a binomial distribution (*plateau small*) and for normally distributed sizes with a mean of 1024 (*plateau large*), respectively.

**Heuristics for Choosing Cache-Set Arguments** For the cache-aware dynamic memory allocators, we need to provide a cache-set argument. This additional cache set parameter tells the allocator the cache set to which the returned memory address shall be mapped. In real-life applications, this parameter is either set by a timing analysis while (pre-)analyzing the program or by the programmer himself. Obviously, choosing unsuitable cache set arguments will significantly increase the memory consumption. Computing the optimal cache set arguments may, however, lead to overly optimistic results. We therefore set the cache set arguments according to very simple heuristics as the average programmer would probably do. We chose the same heuristics for all our six benchmark sequences. This heuristics simply cycles through the cache sets that are guaranteed to never contain a descriptor. Hence, we end up with roughly the same number of requests per cache set. Furthermore, this heuristics reduces placement conflicts between memory blocks that contain descriptor areas and blocks that contain memory blocks allocated and accessed by the application.

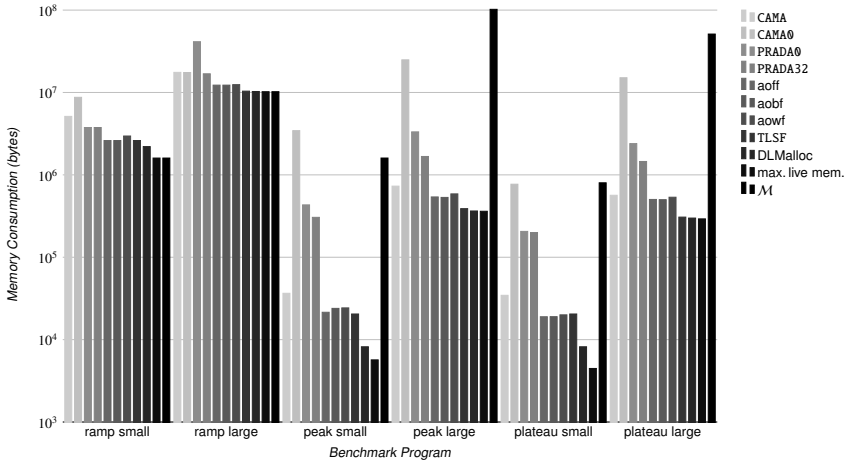



Figure 5.3: Memory consumption of different allocators for the (de-)allocation sequences of our synthetic, randomized test cases.

**Benchmark Results** Figure 5.3 shows the memory consumption of the different allocators for our synthetic allocation sequences. The exact values are given in Table 5.5.

Comparing CAMA to `TLSF`, we observe a noticeable increase in memory consumption for adding cache awareness.

Consider the test case `ramp small`. The increased memory consumption of CAMA does not seem to stem from the predefined cache-set mapping to which the allocator must adhere. This is evidenced by our cache-aware sequential fit allocators that perform very close to `TLSF`, with the worst-fit allocator performing noticeably worse. This may imply that worst-fit strategies perform comparably bad on this particular allocation sequence. We also note that CAMA often resorts to a bad-fit approach (approximating a worst-fit) in this test case as evidenced by the much higher memory consumption of `CAMA0` for which we disabled a worst-fit fallback. More importantly, however, this allocation sequence consists exclusively of very small requests. This is a particularly bad case for CAMA. Although these blocks are grouped in areas, each cache page uses its entire

| | ramp small | ramp large | peak small | peak large | plateau small | plateau large |
|---|---|---|---|---|---|---|
| CAMA | 5,128,544 | 17,551,456 | 36,544 | 730,112 | 34,528 | 565,568 |
| % fragmentation | 220.426 | 71.729 | 543.380 | 101.842 | 672.784 | 93.494 |
| CAMA0 | 8,737,728 | 17,473,600 | 3,440,992 | 24,895,840 | 770,400 | 15,098,208 |
| % fragmentation | 445.924 | 70.967 | 60,480.845 | 6,782.551 | 17,142.614 | 5,065.454 |
| PRADA0 | 3,758,144 | 41,332,864 | 432,192 | 3,325,920 | 206,016 | 2,396,640 |
| % fragmentation | 134.805 | 304.413 | 7,509.014 | 819.463 | 4,510.922 | 719.947 |
| PRADA | 3,758,144 | 16,859,136 | 305,216 | 1,672,768 | 199,520 | 1,454,016 |
| % fragmentation | 134.805 | 64.955 | 5,273.521 | 362.443 | 4,365.533 | 397.453 |
| aoff | 2,609,812 | 12,283,780 | 21,496 | 539,716 | 18,968 | 503,200 |
| % fragmentation | 63.058 | 20.188 | 278.451 | 49.207 | 324.530 | 72.157 |
| aobf | 2,609,816 | 12,279,388 | 23,960 | 532,580 | 19000 | 499,620 |
| % fragmentation | 63.058 | 20.145 | 321.831 | 47.234 | 325.246 | 70.932 |
| aowf | 2,963,188 | 12,447,844 | 24,284 | 587,748 | 19,992 | 535,520 |
| % fragmentation | 85.137 | 21.793 | 327.535 | 62.485 | 347.449 | 83.214 |
| TLSF | 2,611,200 | 10,383,360 | 20,480 | 389,120 | 20,480 | 307,200 |
| % fragmentation | 63.145 | 1.594 | 260.563 | 7.573 | 358.371 | 5.100 |
| DLMalloc | 2,203,648 | 10,280,960 | 8,192 | 364,544 | 8,192 | 299,008 |
| % fragmentation | 37.682 | 0.592 | 44.225 | 0.780 | 83.348 | 2.298 |
| Oracle | 1,600,540 | 10,220,464 | 5,680 | 361,724 | 4,468 | 292,292 |
| $\mathcal{M}$ | 1,600,540 | 10,220,464 | 1,600,992 | 102,201,168 | 800,980 | 51,203,048 |

Table 5.5: Memory consumption (in bytes) and percentage of fragmentation of the different allocators for our synthetic (de-)allocation traces.

descriptor area to manage the (payload) areas of the same cache page. CAMA is consequently forced to produce the maximal number of descriptor areas for its currently allocated memory range. This results in the worst possible ratio of descriptor overhead to usable (by the program) memory. The descriptor areas also incur additional external fragmentation.

In `ramp large`, the allocation sequence allows for a better ratio of descriptor areas to managed memory blocks. We consequently observe a significant decrease in CAMA's memory fragmentation. However, the predefined cache-set mapping does raise memory demands in this case, putting CAMA at a disadvantage again.

These observations are confirmed by the remaining four test cases: for sequences containing many small allocations, CAMA suffers from a bad ratio of descriptor areas to managed blocks as well as increased external fragmentation from those descriptor blocks. On sequences with mainly large requests, CAMA performs reasonably well. Please note that in those test cases, the predefined cache-set mapping alone raises memory demands already well above `TLSF`'s overall memory consumption.

In conclusion, we consider CAMA's performance rather encouraging considering that, as already pointed out, the predefined mapping already increases memory demands. Furthermore, descriptors incur comparably high memory overhead and also raise external fragmentation.

What else can we conclude from those results? Comparing `PRADA` to `PRADA0`, our data evidences the usefulness of enabling split and merge operations in an allocator. CAMA`0`, `PRADA`, and `PRADA0` also prove that incomplete memory use is indeed a serious problem with our proposed lookup mechanism and managing of free lists. Without counteracting incomplete memory use, one always risks prohibitively large memory consumption. Nevertheless, the allocators performed still well below $\mathcal{M}$ and hence below their provable worst-case behavior. This may suggest that typical (de)allocation sequences do not provoke worst-case behavior in our cache-aware allocators.

## 5.4 Memory Performance for Real-Life Programs

To further evidence the validity of the memory performance observed with our randomized (de-)allocation traces, (de-)allocation traces of real-life programs are desirable. However, embedded systems currently avoid dynamic memory management due to the problems we discussed in Chapter 2.4. Still, a very small set of soft real-time applications using dynamic memory allocation is contained in the MiBench benchmark suite [GRE+01]. The MiBench suite itself consists of a set of embedded programs, considered to be representative for commercial applications. Unfortunately, only six test cases of this suite use dynamic memory allocation. These six test cases execute the programs `Susan`, `Patricia`, and `Dijkstra`, each on a set of small and large input data, respectively.

`Susan` was originally developed for recognizing corners and edges in magnetic resonance images of the brain. The software is, however, also used for image recognition in unmanned vehicles. The instantiation working on a small input image processes a black and white image of a rectangle, while in the large input data instance a complex picture is processed. Both test cases perform four allocation requests only. The requested sizes vary (three different sizes, each) and adhere to a ramp pattern. In total, 43,836 bytes (susan small) and 664,068 bytes (susan large) are requested, respectively.

`Dijkstra` constructs a graph (as a $100 \times 100$ adjacency matrix) and then computes 20 and 100 shortest paths, respectively, between pairs of nodes using repeated applications of Dijkstra's algorithm. The small and the large test case for `Dijkstra` differ only in the number of path computations and hence the number of (de-)allocations. The requested sizes are always the same (16 bytes), the (de-)allocation pattern is neither strictly following a peak, ramp or plateau pattern. It can probably be best described as a LIFO pattern (in the sense of last allocated, first deallocated) within a peak pattern showing no further observable regularities. The complete (de-)allocation sequences are of length 29,950 (dijkstra small) and 151,442 (dijkstra large), respectively.

`Patricia` uses patricia tries to construct routing tables. A patricia trie is a data structure used in place of full trees with very sparse leaf nodes. Patricia tries are often used to represent routing tables in network applications. The (de-)allocation

sequences produced by the two `Patricia` test cases adhere to a *ramp* pattern. Both sequences request only three sizes: 8, 12, and 20 bytes. Each size is requested 10,891 (`patricia small`) and 62,722 (`patricia large`) times, respectively. Hence, both test cases differ only in the total number of allocation requests.

For our benchmarking, we recorded the (de-)allocation sequences of the respective program executions and generated reduced programs that simply reproduce these sequences without the actual program computations. Again, the heuristics we used for setting the additional cache set argument for our cache-set guided memory allocators were intentionally kept very simple.

The heuristics used for the `Susan` test cases is based on the assumption that memory is never deallocated and just put consecutively in memory. The heuristics simulates this behavior and sets cache-set arguments to the cache set that the start addresses of allocated blocks are mapped to in its simulation. For `Dijkstra`, we aimed for an equal distribution over the cache sets not mapped to by descriptor areas; as we do with our synthetic test cases. To achieve this, we, again, simply cycle through the cache sets that are guaranteed to never contain a descriptor. In the `Patricia` test cases, we use the same heuristics as in the `Dijkstra` test cases.

Figure 5.4 shows the results of our memory performance measurements for those real-life (de-)allocation sequences. The exact values are given in Table 5.6.

On these real-life applications, CAMA performs noticeably better than what we would expect from the evaluation on synthetic benchmarks. For the `Susan` test cases, we provided a better cache-set mapping that does not increase memory demands. We therefore do only observe a slight increase in memory consumption compared to `TLSF` (`susan large`) or even none at all (`susan small`, where rounded allocations when obtaining fresh memory from the operating system put `TLSF` at a disadvantage).

In the `Dijkstra` test cases, our again very simple cache-set mapping puts the cache-aware allocators already at a disadvantage. Additionally, only very small blocks are requested, forcing CAMA to, again, cope with a bad-case sequence of allocations. Still, overall memory consumption is very reasonable for CAMA, especially when compared to the sum of requested memory ($\mathcal{M}$). These test cases also evidence the significant influence of splitting and merging on the overall memory consumption (see difference between `PRADA` and `PRADA0`).

| | susan small | susan large | dijkstra small | dijkstra large | patricia small | patricia large |
|---|---|---|---|---|---|---|
| CAMA | 48,288 | 676,736 | 17,024 | 17,024 | 1,503,584 | 8,654,912 |
| % fragmentation | 10.156 | 1.908 | 237.778 | 223.404 | 245.144 | 244.971 |
| CAMA0 | 92,532 | 716,884 | 359,988 | 1,596,980 | 4,730,420 | 26,623,540 |
| % fragmentation | 111.087 | 7.953 | 7,042.619 | 30,237.766 | 985.855 | 961.172 |
| PRADA0 | 157,696 | 782,048 | 1,148,544 | 1,148,544 | 78,602,912 | 452,203,808 |
| % fragmentation | 259.741 | 17.766 | 22,688.571 | 21,718.845 | 17,943.089 | 17,924.131 |
| PRADA | 157,696 | 782,048 | 181,696 | 245,760 | 1,229,472 | 6,268,160 |
| % fragmentation | 259.741 | 17.766 | 3,505,079 | 4,568.693 | 182.222 | 149.839 |
| aoff | 43,852 | 664,084 | 12,264 | 12,296 | 611,768 | 3,516,368 |
| % fragmentation | 0.037 | 0.002 | 143.333 | 133.587 | 40.430 | 40.157 |
| aobf | 43,852 | 664,084 | 12,264 | 12,296 | 611,768 | 3,516,368 |
| % fragmentation | 0.037 | 0.002 | 143.333 | 133.587 | 40.430 | 40.157 |
| aowf | 43,852 | 664,084 | 16,180 | 16,276 | 623,928 | 3,538,224 |
| % fragmentation | 0.037 | 0.002 | 221.032 | 209.195 | 43.221 | 41.028 |
| TLSF | 50,756 | 675,012 | 21,508 | 21,508 | 789,508 | 4,527,108 |
| % fragmentation | 15.786 | 1.648 | 326.746 | 368.587 | 81.229 | 80.443 |
| DLMalloc | 45,056 | 667,648 | 8,192 | 8,192 | 610,304 | 3,514,368 |
| % fragmentation | 2.783 | 0.539 | 62.540 | 55.623 | 40.094 | 40.077 |
| Oracle | 43,836 | 664,068 | 5,040 | 5,264 | 435,640 | 2,508,880 |
| $\mathcal{M}$ | 43,836 | 664,068 | 239,600 | 1,211,536 | 435,640 | 2,508,880 |

Table 5.6: Memory consumption (in bytes) and percentage of fragmentation of the different allocators for the (de-)allocation sequences of the MiBench test cases.

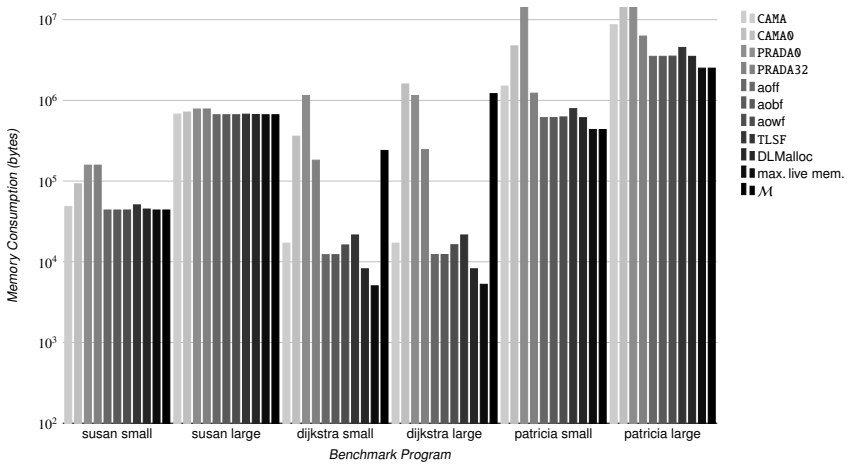



Figure 5.4: Memory consumption of different allocators for the (de-)allocation sequences of the MiBench test cases.

The `Patricia` test cases do not allow for memory reuse and request a very large number of very small blocks. Consequently, all allocators perform rather poorly.

Again, allocators prone to incomplete memory use show overly high memory consumption in all test cases.

**Precomputed Allocation Schemes for Real-Life Programs** In [HA10], we also evaluated our approach to a static precomputation of suitable memory addresses for a program's dynamic allocations, as discussed in Chapter 3.3, on this same set of benchmark programs (`Susan`, `Dijkstra`, and `Patricia`). We reproduce these results in the following paragraphs and compare the memory performance of a static precomputation to that of a dynamic memory allocation.

`Susan`'s allocation behavior can be formalized to

$$(M_s, U_s, \{\}, A_s, C_s, (R \times R) \mapsto \{0\})$$

where

$$\begin{aligned}
M_s &= \{1,2,3,4,5,6,7,8,9\} \\
U_s &= \bigcup_{m \in M_s} u_s = \{1\} \\
A_s &= \{f_1 : \{1 \mapsto x \cdot y\}, f_2 : \{1 \mapsto 516\}, f_3 : \{1 \mapsto (14+x)(14+y)\}, \\
&\quad f_4 : \{1 \mapsto 16\}, f_5 : \{1 \mapsto 4 \cdot x \cdot y\}, f_6 : \{1 \mapsto 4 \cdot x \cdot y\}, \\
&\quad f_7 : \{1 \mapsto 4 \cdot x \cdot y\}, f_8 : \{1 \mapsto x \cdot y\}, f_9 : \{1 \mapsto 4 \cdot x \cdot y\}\} \\
C_s(C) &= \begin{cases}
1 & \text{if } \{(1,1)\} \subseteq C \\
& \vee \{(2,1)\} \subseteq C \\
& \vee \{(7,1),(8,1)\} \subseteq C \\
& \vee \{(3,1),(4,1)\} \subseteq C \\
0 & \text{otherwise}
\end{cases}
\end{aligned}$$

Here, $x$ and $y$ are fixed parameters determining the size of the processed images. Our algorithm computed a set of memory block chunks as depicted in Figure 5.5.

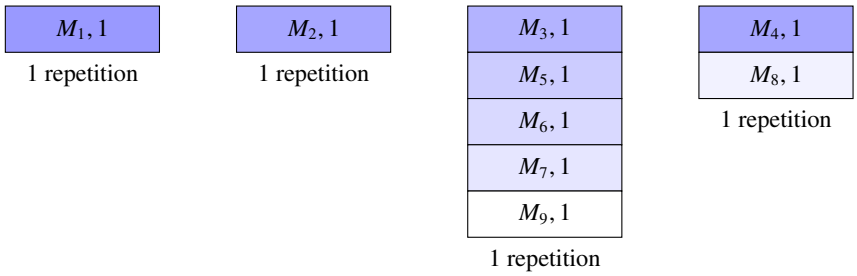



Figure 5.5: Allocation chunks for the `Susan` test case.

Transforming this chunk set yields static allocation schemes requiring $m$ memory, where $m$ is just the sum over the memory requirements for the four concatenated chunks, i.e.,

$$m = x \cdot y + 516 + \max\left\{(14 + x)\,(14 + y)\,, 4 \cdot x \cdot y, 512\right\} + \max\left\{8, x \cdot y\right\}$$

For our two instantiations of `Susan`, $x$ and $y$ are set to $x = 76 \wedge y = 95$ and $x = 384 \wedge y = 288$ for `susan small` and `susan large`, respectively. Hence, these schemes require 7,220 + 516 + 28,880 + 7,220 = 43,836 and 110,592 + 516 + 442,368 + 110,592 = 664,068 bytes of memory. We observe that our precomputation yields memory optimal allocation schemes in both cases.

The allocation behavior of the `Dijkstra` test cases can be described by

$$(M_d, U_d, \{\}, A_d, C_d, (R \times R) \mapsto \{0\})$$

where

$$\begin{aligned} M_d &= \{1\} \\ U_d &= \{u_1\} = \left\{n^2\right\} \\ A_d &= \left\{f_1 : \mathbb{N}^{\leq n^2} \mapsto [16, 16]\right\} \\ C_d(C) &= \begin{cases} 0 & \text{if } C = \{(1,1),(1,2)\} \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

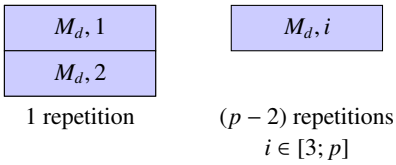



Figure 5.6: Allocation chunks for the `Dijkstra` test cases, where $p = n^2$.

Here, $n$ corresponds to the number of nodes of the constructed graph. Figure 5.6 depicts the computed set of memory block chunks for this allocation problem. The resulting allocation schemes from these chunks would require $(1+(100^2-2))\cdot 16 =$ 159,984 bytes of memory (as the 20 and 100 searches performed in the test cases are pairwise independent).

This application shows one limitation of precomputing memory addresses, namely that it requires precise liveness information regarding which allocated blocks are alive at the same time.

The `Patricia` benchmark programs are reduced to the following formalization of their allocation behavior:

$$\left(M_p, U_p, \{\}, A_p, C_p, \mathcal{B}_p, (R \times R) \mapsto \{0\}\right)$$

where

$$\begin{aligned} M_p &= \{1, 2, 3\} \\ U_p &= \{u\} \\ A_p &= \{f_1 : \mathbb{N}^{\leq I} \mapsto [20, 20], f_2 : \mathbb{N}^{\leq I} \mapsto [8, 8], f_3 : \mathbb{N}^{\leq I} \mapsto [12, 12]\} \\ C_p(C) &= 1 \end{aligned}$$

As we cannot safely determine that two allocated blocks are not contemporaneously in-use, our algorithm is not able to compute a better set of memory block chunks than the one given in Figure 5.7. However, in this case, all blocks overlap in their lifetimes, so this result cannot be improved.

Furthermore, as each allocation site always requests the same, statically known size, there is no uncertainty about the requested sizes. But what memory consump-

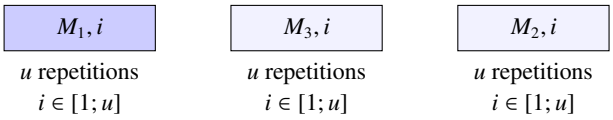



Figure 5.7: Allocation chunks for the `Patricia` test cases.

tion does our precomputed memory scheme exhibit in this case? The only variable in our set of chunks is the upper bound $u$ on the invocations of the allocation sites. This variable, however, simply corresponds to the number of inputs (i.e., UDP packets in this case) plus one header node that is always generated. The `Patricia` test cases process 10,891 and 62,722 such packets, yielding upper bounds of 32,673 and 188,166 for `patricia small` and `patricia large`, respectively. Hence, we generate schemes using 435,640 and 2,508,880 bytes of memory, respectively. Again, these schemes are memory optimal.

# 6

# Summary & Conclusions

> Reason is itself a matter of faith. It is an act of faith to assert that our thoughts have any relation to reality at all.
>
> G. K. Chesterton
> *Writer (1874–1936)*

We have discussed the challenges dynamic memory allocation imposes on static timing analysis. On modern embedded hardware, cache performance has a large influence on an application's timing behavior. Hence, real-time applications have to exhibit predictable cache behavior. Otherwise, tight bounds on the cache performance of an application and hence its execution times cannot be statically determined.

When precise information about an application's allocation behavior can be derived statically, precomputing a static allocation scheme to replace the application's dynamic memory allocation is the preferable option to circumvent the challenges we identified. We have developed algorithms to statically precompute suitable allocation schemes. We furthermore proposed a static analysis capable of deriving precise information about an application's allocation behavior.

In case this first approach is not applicable due to the absence of useful liveness information, we proposed techniques to build a predictable, cache-aware memory allocator by which to replace the standard allocator. CAMA provides predictable cache behavior in the sense that it (a) can be guided with respect to which cache set allocated memory is mapped to and it (b) guarantees to access only a statically known subset of cache sets while processing allocation and deallocation requests. Furthermore, CAMA's execution times can be tightly bounded.

TLSF is already successfully used within soft real-time systems. This indicates that the price paid for real-time behavior in terms of increased memory consumption is considered worth paying in order to be able to use dynamic memory allocation. Hard real-time systems, however, still rely exclusively on static memory allocation as the cache influence of dynamic memory allocation, even when using TLSF, introduces too much uncertainty about the cache performance. CAMA may provide guarantees about an application's cache behavior that may equally be worth the corresponding further increase in memory consumption. It therefore seems reasonable to believe that CAMA can enable the use of dynamic memory allocation within future, more complex hard real-time systems.

# 7 Future Work

> We learn something every day, and lots of times it's that what we learned the day before was wrong.
>
> Bill Vaughan
> *Columnist and author (1915–1977)*

This thesis proposes two approaches to enable dynamic memory allocations for programmers writing real-time applications: statically transforming dynamic into static allocation and employing a predictable dynamic memory allocator. Enabled by the latter approach, we identified two research directions worthy of further investigations.

**Lookup Mechanism for Predictable Allocators** As our evaluations show, the weak spot of our predictable allocators seems to be their lookup mechanism. I.e., their internal table of free lists that can only be efficiently searched when considering only free blocks starting at a given cache set. This, however, is shown to be often too restrictive: blocks that may be split to yield a suitable block to satisfy a request are overlooked because they start in a different cache set. While we circumvent this with an additional bounded (bit-) search over *promising* free lists that may start in any cache set, there might be an alternative, more elegant

solution. We also propose to allow non-constant lookup mechanisms, as long as they can be tightly bounded, to widen the field of techniques that may be employed for this lookup.

**Decreasing an Application's WCET** By employing our proposed predictable dynamic memory allocator, not just the statically provable bound on an application's WCET, but its actual WCET may be decreased. Given a set of representative real-time applications, the degree to which these applications' WCETs are reduced seems to be worth measuring. While currently a useful set of such applications is not available, this may change in the near future.

Please note that in [HBHR11], we show very encouraging results in that direction. In this paper, we demonstrate the impact of cache awareness on WCET bounds by analyzing a simplified task scheduler (see Listing 7.1).

Listing 7.1: The main loop body of a simplified task scheduler.

```
struct task_descr* lowPriority = low;                                          1
struct task_descr* highPriority = high;                                        2
for(i = 0; i < HUGE_LIST_SIZE; i++) {                                          3
    for(j = 0; j < SMALL_LIST_SIZE; j++) { // high prioritized tasks waiting?  4
        high = high->next;                                                     5
        ...                                                                    6
    }                                                                          7
    high = highPriority;                                                       8
    low = low->next; // next lower prioritized task waiting?                   9
    ...                                                                        10
}                                                                              11
low = lowPriority;                                                             12
```

For this application, we could manually annotate the statically available information about the cache-set mapping to which our dynamically allocated objects adhere. This scheduler manages two singly-linked lists composed of task descriptors with maximally 4 and 16 entries, respectively. The smaller list is used for high priority tasks, the other for all other tasks. CAMA is used to ensure that all high-priority objects map to a cache set different from the other objects. In this setup, analyzing the WCET of the program fragment given in Listing 7.1 using aiT yields

an upper bound on the WCET of 6,505 cycles on a PowerPC MPC603e. With a statically unknown cache-set mapping, however, only a WCET of 10,915 cycles can be guaranteed on the same hardware.

Note that the analyzed code does not contain any invocations of the memory allocator, so the gain in WCET guarantees is completely attributed to the analysis being able to exclude that objects of the lower prioritized list evict higher priority objects from the cache, which leads to the safe prediction of cache hits when traversing the higher priority list again. Also note that the use of CAMA does also decrease the *real* WCET, not just the provable WCET, as potential conflict misses are actually eliminated.

# List of Figures

# List of Tables

# Bibliography


[AAN11] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. Precise and Efficient Parametric Path Analysis. In Jan Vitek and Bjorn De Sutter, editors, *LCTES*, pages 141–150. ACM, 2011.

[AHLW08] Sebastian Altmeyer, Christian Humbert, Björn Lisper, and Reinhard Wilhelm. Parametric Timing Analysis for Complex Architectures. In *RTCSA*, pages 367–376. IEEE Computer Society, 2008.

[AZMM04] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[BEL11] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An Efficient Algorithm for Parametric WCET Calculation. *Journal of Systems Architecture - Embedded Systems Design*, 57(6):614–624, 2011.

[CC76] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *2nd International Symposium on Programming*, pages 106–130, April 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[CHL00] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making Pointer-Based Data Structures Cache Conscious. *IEEE Computer*, 33(12):67–74, 2000.

[CMWC09] Liquian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval Polyhedra: An Abstract Domain to Infer Interval Linear Relationships. In J. Palsberg and Z. Su, editors, *Proceedings of the sixteenth International Static Analysis Symposium (SAS'09)*, pages 309–325, Los Angeles, CA, USA, January 17–19 2009. LNCS 5673, Springer, Berlin.

[Com64] Webb T. Comfort. Multiword List Items. *Commun. ACM*, 7:357–362, June 1964.

[CP03] Calin Caşcaval and David A. Padua. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 150–159, New York, NY, USA, 2003. ACM.

[CPHL01] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact Analysis of the Cache Behavior of Nested Loops. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 286–297, New York, NY, USA, 2001. ACM.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational Inductive Shape Analysis. In *POPL '08*, New York, NY, USA, 2008. ACM.

[CS00] Siddhartha Chatterjee and Sandeep Sen. Cache-efficient Matrix Transposition. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 195–205, 2000.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A Local Shape Analysis Based on Separation Logic. In *Proceedings of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag.

[DS72] Peter J. Denning and Stuart C. Schwartz. Properties of the Working-Set Model. *Commun. ACM*, 15:191–198, March 1972.

[ESG+07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[Fer97] Christian Ferdinand. *Cache Behaviour Prediction for Real-Time Systems*. PhD thesis, Universität des Saarlandes, 1997.

[FHW+08] Christian Ferdinand, Reinhold Heckmann, Hans-Jörg Wolff, Oleg Parshin, Reinhard Wilhelm, and AbsInt Angewandte Informatik GmbH. Towards Model-Driven Development of Hard Real-Time Systems Integrating ASCET and aiT/StackAnalyzer, 2008.

[FMW97] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS)*, pages 37–46, June 1997.

[FMWA96] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache Behavior Prediction by Abstract Interpretation. In *Science of Computer Programming*, pages 52–66. Springer, 1996.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-TimeSystems. *Real-Time Systems*, 17:131–181, December 1999.

[GGU72] Michael R. Garey, Ronald L. Graham, and Jeffrey D. Ullman. Worst-case Analysis of Memory Allocation Algorithms. In *Proceedings of the fourth annual ACM symposium on Theory of computg*, STOC '72, pages 143–150, New York, NY, USA, 1972. ACM.

[GMM97] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324. ACM Press, 1997.

[GMM98] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, 1998.

[GRE+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *International Workshop on Workload Characterization, 2001. WWC-4.*, pages 3–14. IEEE, 2001.

[HA10] Jörg Herter and Sebastian Altmeyer. Precomputing Memory Locations for Parametric Allocations. In Björn Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 125–136. Austrian Computer Society, July 2010.

[Hah11] Sebastian Hahn. Towards Relational Cache Analysis. Bachelor's thesis, Saarland University, 2011.

[Hau12] Florian Haupenthal. PRADA: Predictable Allocations by Deferred Actions. Bachelor's thesis, Saarland University, 2012.

[HBHR11] Jörg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. CAMA: A Predictable Cache-Aware Memory Allocator. In *Pro-*

*ceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS '11)*. IEEE Computer Society, July 2011.

[Her10] Jörg Herter. Allocation-Site Aware Shape Analysis and Applications in Hard Real-Time Systems. In *Proceedings of JRWRTC*, November 2010.

[HFG04] Reinhold Heckmann, Christian Ferdinand, and AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Prediction by Static Program Analysis. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, pages 26–30. IEEE Computer Society, 2004.

[HG12] Sebastian Hahn and Daniel Grund. Relational Cache Analysis for Static Timing Analysis. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS '12)*, pages 102–111, July 2012.

[HH13] Florian Haupenthal and Jörg Herter. PRADA: Predictable Allocations by Deferred Actions. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 77–86, Dagstuhl, Germany, July 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Hir73] Daniel S. Hirschberg. A Class of Dynamic Memory Allocation Algorithms. *Commun. ACM*, 16(10):615–618, October 1973.

[Hon09] Olha Honcharova. Static Detection of Parametric Loop Bounds on C Code. Master's thesis, Universität des Saarlandes, February 2009.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Edition*. Morgan Kaufmann, 1996.

[HR09] Jörg Herter and Jan Reineke. Making Dynamic Memory Allocation Static To Support WCET Analyses. In *Proceedings of 9th Interna-*

*tional Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

[HRW08] Jörg Herter, Jan Reineke, and Reinhard Wilhelm. CAMA: Cache-Aware Memory Allocation for WCET Analysis. In Marco Caccamo, editor, *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, July 2008.

[ISO99] ISO. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, December 1999. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`.

[IT88] François Irigoin and Remi Triolet. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[KGV83] Scott Kirkpatrick, Charles Daniel Gelatt, Jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Kno65] Kenneth C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8:623–624, October 1965.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[LA03] Chris Lattner and Vikram Adve. Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis. Technical report, University of Illinois at Urbana-Champaign, 2003.

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting Static Analysis to Work for Verification: A Case Study. In *ISSTA*, 2000.

[Lea96] Doug Lea. A Memory Allocator. unix/mail, 1996.

[Leg13] Andreas Legrum. Extending TVLA to Allow Allocation-Site Aware Shape Analysis of Hard Real-Time Applications. Bachelor's thesis, Saarland University, 2013.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew L. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, RTSS '96, pages 254–, Washington, DC, USA, 1996. IEEE Computer Society.

[LS99] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[LTH02] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline Modeling for Timing Analysis. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 294–309, London, UK, 2002. Springer-Verlag.

[LW94] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *Computer*, 27(10):15–26, October 1994.

[MBCS08] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *RTCSA*, pages 161–166. IEEE Computer Society, 2008.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[MH93] Frank Mueller and David B. Whalley Marion Harmon. Predicting Instruction Cache Behavior. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1993.

[Min06] Antoine Miné. The Octagon Abstract Domain. *Higher Order Symbol. Comput.*, 19(1):31–100, March 2006.

[MRBC08] Miguel Masmano, Ismael Ripoll, Patricia Balbastre, and Alfons Crespo. A Constant-Time Dynamic Storage Allocator for Real-Time Systems. *Real-Time Systems*, 40(2):149–179, 2008.

[MRC03] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. Dynamic Storage Allocation for Real-Time Embedded Systems. In *Proceedings Work-In-Progress Session of the 24th IEEE Real-Time Systems Symposium*, 2003.

[MRCR04] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorges Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *16th Euromicro Conference on Real-Time Systems*, pages 79–88. IEEE, 2004.

[MRR+08] Miguel Masmano, Ismael Ripoll, Jorges Real, Alfons Crespo, and A.J. Wellings. Implementation of a Constant-Time Dynamic Storage Allocator. *Softw. Pract. Exper.*, 38(10):995–1026, August 2008.

[Oga95] Takeshi Ogasawara. An Algorithm with Constant Execution Time for Dynamic Storage Allocation. *Real-Time Computing Systems and Applications, International Workshop on*, 0:21, 1995.

[PN77] James L. Peterson and Theodore A. Norman. Buddy Systems. *Commun. ACM*, 20:421–431, June 1977.

[PR02] Erez Petrank and Dror Rawitz. The Hardness of Cache Conscious Data Placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 101–112, New York, NY, USA, 2002. ACM.

[PR05] Erez Petrank and Dror Rawitz. The Hardness of Cache Conscious Data Placement. *Nordic J. of Computing*, 12(3):275–307, June 2005.

[Ran69] Brian Randell. A Note on Storage Fragmentation and Program Segmentation. *Commun. ACM*, 12:365–ff., July 1969.

[Rei08] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008.

[Rey02] John Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. IEEE Computer Society, 2002.

[Rob71] John M. Robson. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *J. ACM*, 18(3):416–423, July 1971.

[Rob74] John M. Robson. Bounds for Some Functions Concerning Dynamic Storage Allocation. *J. ACM*, 21(3):491–499, July 1974.

[Rob77] John M. Robson. Worst Case Fragmentation of First Fit and Best Fit Storage Allocation Strategies. *Comput. J.*, 20(3):242–244, 1977.

[RT98] Gabriel Rivera and Chau-Wen Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, 1998.

[RwT98] Gabriel Rivera and Chau wen Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, pages 353–360. ACM press, 1998.

[RWT+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of the 6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[San04] Daniel Sandell. Evaluating Static Worst-Case Execution-Time Analysis for a Commercial Real-Time Operating System. Technical report, Department of Computer Science and Electronics, Mälardalen University, Västerås, Sweden, 2004.

[Seh05] Daniel Sehlberg. Static WCET Analysis of Task-Oriented Code for Construction Vehicles. Technical report, Department of Computer Science and Electronics, Mälardalen University, Västerås, Sweden, 2005.

[SP74] Kenneth K. Shen and James L. Peterson. A Weighted Buddy Method for Dynamic Storage Allocation. *Commun. ACM*, 17(10):558–562, October 1974.

[SPH+05] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jgu, and Guillaume Borios. Computing the Worst-Case Execution-Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[Tan06] Lili Tan. The Worst Case Execution Time Tool Challenge 2006. Technical Reports of WCET Tool Challenge 1, University Duisburg-Essen, December 2006.

[War84] Joe Warren. A Hierarchical Basis for Reordering Transformations. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 272–282, New York, NY, USA, 1984. ACM.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.

[WHW+97] Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, Washington, DC, USA, 1997. IEEE Computer Society.

[Wil06] Reinhard Wilhelm. Determining Bounds on Execution Times. In Richard Zurawski, editor, *Handbook on Embedded Systems*, chapter 14. CRC Press, 2006.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *IWMM*, pages 1–116. Springer-Verlag, 1995.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[Wol89] Michael J. Wolfe. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.

[WR99] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. *SIGPLAN Not.*, 34(10), 1999.