Cache Analysis in Presence of Pointer-Based Data Structures

Tomasz Dudziak
Institute of Computer Science
University of Wrocław, Poland
Email: tomasz.dudziak@gmail.com

Jörg Herter
Department of Computer Science
Saarland University, Germany
Email: jherter@cs.uni-sb.de

***Abstract*—Cache analysis plays a crucial part when analyzing the WCET of an application. This paper presents ongoing work aiming at a precise cache analysis in the presence of pointer-based, heap-allocated data structures. The proposed analysis achieves precision by augmenting its abstract cache states with information about the structure of the program's allocated objects as well as a short term access history of these objects. This additional information is derived from an adapted shape analysis that serves as an pre-analysis phase to our actual cache analysis.**

I. Introduction

Caches are ubiquitous in modern hardware. They are used to bridge the ever increasing gap between processor speeds and memory access times. However, for hard real-time applications caches introduce additional challenges. To give safe and precise estimations of the worst-case execution time (WCET) of a program, additional *cache analyses* [1] are needed to derive tight bounds on the cache performance. Such analyses need to be able to classify most memory accesses as cache hits—the accessed object will always be in the cache—or misses—the object will always be loaded from memory. Conservatively classifying each memory access as a cache miss is in general not an option as this would result in a prohibitively large overestimation of the WCET.

Current cache analyses perform poorly on programs that manipulate pointer-based data structures like linked lists or trees. Analyzing such programs is challenging for two reasons:

1) Pointer-based data structures are usually dynamically allocated. However, the cache set mapping of addresses returned by standard memory allocators are impossible to predict statically.
2) Traversing such structures involves accessing different memory addresses depending on the history of previous manipulations. Hence, a cache analysis would have to track the *shape* of data structures. This is a well known problem in static program analysis [2].

In this paper, we outline ongoing research aimed at developing an adapted cache analysis that is able to handle real-time programs manipulating pointer-based data structures. Section II elaborates on the challenges addressed by this work. Our proposed solution consists of several phases as depicted in Figure 1. In Section III, we discuss how *shape graphs* can be used to provide cache analyses with information about the program's data structures. Our analysis additionally relies on information about allocation sites. Section IV discusses what information is required and how it can be derived from the program or supplied via user annotations. In Section V, we extend an existing cache analysis to derive more precise cache behavior description, especially for programs using pointer-based data structures. In summary, the main contribution of this work is the combination of shape analysis with traditional cache analysis to increase the precision of the latter. To achieve this, we propose a specialized shape analysis and show how an adapted cache analysis can use the additional knowledge derived by this shape analysis to increase its own precision.

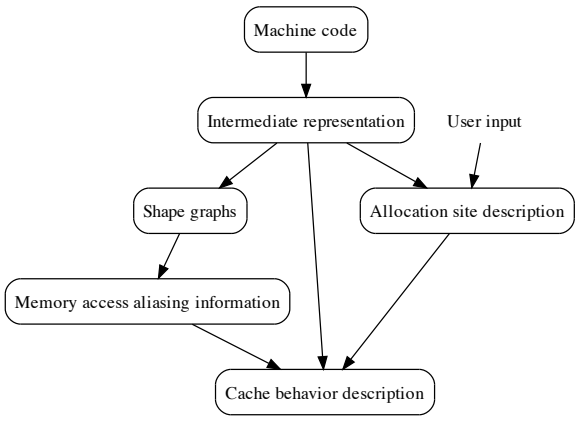

Fig. 1. High level design of the analysis

II. Dynamic Memory Allocation and WCET Analysis

General dynamic memory allocators strive to minimize memory fragmentation and (de-)allocation times. Fragmenta-

```
1  struct uip_fw_netif {
2      struct uip_fw_netif *next;
3      u16_t ipaddr[2];
4      u16_t netmask[2];
5      u8_t (* output)(void);
6  };
```

Listing 1: Network interfaces in uIP's packet forwarding subsystem form a linked list.

```
1  static struct uip_fw_netif *
2  find_netif(void)
3  {
4      struct uip_fw_netif *netif;
5
6      /* Walk through every network interface
7         to check for a match. */
8      for(netif = netifs;
9          netif != NULL;
10         netif = netif->next) {
11
12         if(ipaddr_maskcmp(BUF->destipaddr,
13                           netif->ipaddr,
14                           netif->netmask)) {
15
16             /* If there was a match, we break
17                the loop. */
18             return netif;
19         }
20     }
21
22     /* If no matching netif was found, we use
23        default netif. */
24     return defaultnetif;
25 }
```

Listing 2: Example routine from uIP's packet forwarding subsystem.

tion is counteracted by applying a *best fit* strategy that return the free memory block best fitting the current request. This involves a search over at least a subset of the free blocks currently managed by the allocator. Good response times are achieved by minimizing average execution times. Within real-time applications for which tight bounds on their WCETs need to be derived, these otherwise reasonable design decisions for memory allocators introduce serious problems:

1) WCET of the (de-)allocation procedure itself
   Optimized for good average case performance, the worst-case response time of allocators is often linear in the number of currently free memory blocks. This leads to overly pessimistic and hardware dependent bounds on the WCET of (de-)allocations.
2) Cache influence of (de-)allocation procedure itself
   Implementing a best fit strategy results in searching free blocks for the best candidate to return. This often involves accesses to meta data stored directly in the free blocks. We cannot statically decide which blocks will be considered during searches. Therefore, we cannot predict what blocks will be loaded and hence the influence of such a search on the cache by possibly evicting already cached data remains unknown.
3) Cache set mapping of dynamically allocated memory
   Memory addresses of dynamically allocated objects cannot be predicted statically. Thus, guaranteeing cache hits or misses for such objects is in general not possible.

As an example of an embedded program that employs pointer-based, dynamically-allocated data structures, consider uIP [3], a lightweight TCP/IP implementation. Its packet

Fig. 2. Example of a shape graph

forwarding subsystem manages available network interfaces in a linked list of `uip_fw_netif` structures (see Listing 1). For each forwarded packet, a procedure `find_netif` (see Listing 2) traverses this list and checks, for each interface, whether the packet's destination address belongs to the network this interface is connected to.

Performing a cache analysis of `find_netif` poses significant challenges. The list may contain statically allocated elements residing at known addresses, but also entries dynamically allocated by some initialization routine (for example depending on some configuration data or constructed by probing hardware). We also need to look at the program as a whole: local cache behavior depends heavily on previous computations, possibly done in remote places of the program.

We solve this problem by performing a *shape analysis*—a separate analysis that infers compact descriptions of the data structures manipulated by the program.

III. How Can Shape Analysis Help?

Consider the inner loop of `find_netif`. When executed, several memory accesses occur, including an access to the memory block pointed to by `netif`. For simplicity, we assume that allocated structures are cache block-aligned. Also note that a single structure may span several blocks but the offset inside the structure is always statically known.

An analysis has to have some static description of the list `netifs` in order to derive precise information on the cache behavior. Such a description can be given in form of a *shape graph* as depicted in Figure 2. This graph represents a situation where the list has always at least three elements. Identifiers `A32`, `A78`, ... represent allocation sites. By allocation site we mean either a program statement that calls the allocator (in case of dynamically allocated memory) or an abstract "declaration" that identifies a symbol in the binary executable (in case of static allocation).

The graph shows that the first element is always allocated at allocation site `A32`, while the second element may be allocated either at `A78` or `A103`. The other objects are allocated at `A51`.

Given such shape graphs and some external description of allocation sites (what are possible addresses or cache sets, is the address always block-aligned, etc.), the cache behavior of `find_netif` can be predicted with reasonable precision.

The parametric framework for *shape analysis* proposed in [2] can be used to instantiate static analyses that, given a program to analyze, compute sets of (possible) shape graphs for each program point. Shape graphs are represented by logical structures using three-valued first-order logic with

transitive closure. These logical structures can express all essential properties needed for our cache analysis.

Figure 1 sketches our proposed analysis and the propagation of derived information therein. Shape analysis can be seen as a preprocessing stage before the main cache analysis.

In practice, cache analysis is performed on the machine code level. As it is very hard to operate on machine code directly, an additional layer of abstraction in form of an intermediate representation is introduced. A dedicated component of a WCET analyzer called "the frontend" handles the translation to an appropriate representation (usually some form of a control flow graph) [4]. On such a control flow graph, we perform a shape analysis as outlined before. The results of this analysis associate program locations that access memory objects with locations that allocate these objects.

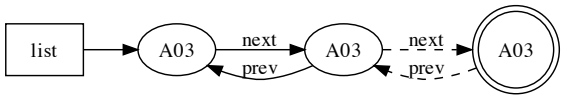

Fig. 3. Shape graph representing a valid doubly-linked list with at least three nodes.

However, this approach may lead to imprecision in even simple cases. Consider a program that manipulates a doubly-linked list by accessing the list node `list`, and subsequently accessing `list->next->prev`. Assume that `list` points to a valid doubly-linked list with a sufficient number of nodes. Such a structure is described by the shape graph in Figure 3. The most precise fact we can state in this situation is that both operations will access memory allocated at some program point `A03`. However, we would like to detect that both memory accesses go to the exactly same address.

To achieve this, we extend the expressiveness of our shape graphs by adding logical predicates that track memory access points. I.e. for each program statement accessing memory objects, we add a predicate that is true for the object that was accessed the last time this statement was executed. After performing shape analysis and some additional post-processing, we are able to derive information about the program like: "Memory access operation at `L352` accesses the same address that was accessed by `L054` last time that statement was executed." or "Memory access operation at `L132` accesses either the same address that was accessed by `L131` or an unknown address allocated at `L004`." We generate such information for each program point in a symbolic form and pass them on to the main cache analysis. The phase generating this information is labeled "Memory access aliasing information" in Figure 1.

Extracting this information from shape graphs involves a minimization step which we found to be equivalent to the *minimal hitting set problem*. This problem is in general NP-hard but the instances we encounter are very small so that solutions are computed reasonably fast using integer linear programming (ILP).

IV. Allocation sites description

By using shape analysis we are able to associate with each memory access an allocation site at which the accessed object was allocated. For later phases of the analysis we need to know the set of all possible objects (or addresses) an allocation site may generate.

In case of statically allocated memory, the address is known and this information will be exact. For dynamic memory allocation, we need to use an allocator that provides control over the cache set mapping of the memory blocks allocated with it. CAMA [5] is such an allocator: it can be guided via an additional argument to what cache set a returned address shall be mapped. For invocations of the dynamic memory allocator we then need to examine CAMA's *cache set argument*.

As a simple solution, we could require this argument to be a compile-time constant which could be read from the analyzed binary. In many cases though, allocating all objects of the same kind in the same cache set defeats the purpose of caching. System designer could as well choose a processor architecture without data cache.

Most WCET analyzers perform a step called *value analysis* [4, Section 3.1]. Its aim is to bound values of registers and local variables at each program point. Such information can be used to derive possible values of CAMA's cache set argument and, in consequence, can be used as allocation site information suitable for our analysis.

The accuracy of the entire analysis crucially depends on the quality of allocation site information. Therefore, a robust implementation should allow users to supply hand-made annotations if the automatically computed bounds are insufficient.

V. Adapting Cache Analysis to Use Memory Access Aliasing Information

We base our main cache analysis on an existing approach [6] using abstract interpretation [7]. We augment this analysis with additional information computed in preceding phases of our analysis.

Abstract interpretation makes a distinction between *concrete semantics* and *abstract semantics*. Concrete semantics, in case of a cache analysis, describes all possible cache states that may occur at a particular program point. If it were always computable, it would be the most precise cache behavior analysis. Abstract semantics is a safe approximation of the concrete semantics. It is less precise but can be constructed to be always computable. Concrete and abstract semantics are related by *abstraction* and *concretization* functions.

In [6], a single cache state is represented by a function:

$$c : \text{Line} \mapsto \text{Store} \cup \{\_\}$$

that maps a cache line to the address of the memory block it currently holds (or "_" if the line is empty). Abstract cache states generalize this to sets of addresses:

$$\hat{c} : \text{Line} \mapsto \mathcal{P}(\text{Store} \cup \{\_\})$$

In preceding phases of our proposed analysis, we generated descriptions for each memory-accessing statement s collecting

0	$\{(?, A)\}$
1	$\{(\texttt{L07}, B)\}$
2	$\{(\texttt{L03}, C), (\texttt{L63}, C)\}$
3	$\{\_\}$

Fig. 4. Example of an abstract state.

possible addresses s may access. We also determined which other statements may have accessed this address before s.

We need to incorporate this knowledge into our abstract cache states. Therefore, along with the stored address, i.e., memory block, we also track which statements caused this block to be loaded into the cache:

$$\hat{c} : \text{Line} \mapsto \mathcal{P}((\text{Loc} \times \mathcal{P}(\text{Store})) \cup \{\_\})$$

Loc denotes the set of all control flow graph nodes, i.e., locations in the program. We assume that each node corresponds to no more than one memory access. We also need a special "empty" location "?" for statements that load a memory block for the first time.

Note that a naïve implementation of sets of addresses using explicit enumeration would be extremely costly. We either need to restrict ourselves to some smaller family of sets (e.g. consider only sets of addresses with same value modulo some constant: $A_i = \{x \mid x \mod C = i\}$) or add an additional layer of abstract interpretation. The latter approach is preferable as it allows to use existing *numerical abstractions* [8, p. 7] which were explicitly developed to approximate sets of integers.

Figure 4 shows a possible abstract state for a small cache with just 4 cache lines. A, B, and C denote sets of addresses (possibly overlapping). In the example, cache line 0 may contain any address from set A. Line 1 may contain any address from set B. However, we know that this address is exactly the same address that was accessed by statement `L07` last time that statement was executed. Knowing that some statement s will always access the same memory block as `L07` from memory aliasing information, enables us to predict a cache hit when s is executed. Line 2 may contain any address from C, but there are two instructions that may have caused this block to be loaded into cache. Line 3 will always be empty.

After we specified our abstract domain and concrete semantics, we can derive equations suitable for computing the analysis using fixed point iteration. This has to be done for each cache architecture, corresponding to different concrete semantics.

VI. Progress and Implementation

Two parts of the proposed analysis have already been implemented: shape analysis and extraction of memory aliasing information. The shape analysis phase is implemented with TVLA [9] and currently uses hand-crafted control-flow graphs as input programs. An appropriate front end that generates these control-flow graphs from binary or source programs, however, is not yet implemented. Generating memory aliasing information from shape graphs is done by a simple program that generates an ILP to extract the desired information from the shape graphs. The program then invokes GLPK [10] to solve the generated ILP.

Future work currently concentrates on deriving appropriate analyses for different cache architectures. That is, architecture specific implementations of the main phase (labeled "cache behavior description" in Figure 1). The domain of this analysis phase depends on the number of available cache sets and their associativity (how many memory blocks can be cached per cache set) but is straightforward to implement given the theoretical construction from Section V. However, the different *cache replacement policies* employed by the different architectures enforce adapted transfer functions that transform one cache state to the next within the analysis. Cache replacement policies are used to decide what cache line is evicted from the cache when a new memory block is to be cached but no cache line is free. Examples of such policies are LRU (least recently used, i.e., replace the block that was not accessed for the longest time) and FIFO (first in first out, i.e., replace the block that has been in the cache for the longest time). Hence, the effect of a memory access on the cache depends highly on the employed replacement policy.

We also plan to research various ways of inferring descriptions of allocation sites with and without user-supplied data.

References

[1] C. Ferdinand and R. Wilhelm, "Efficient and Precise Cache Behavior Prediction for Real-Time Systems," *Real-Time Systems, 17(2-3)*, pp. 131–181, 1999.

[2] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric Shape Analysis via 3-Valued Logic," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 3, 2002.

[3] A. Dunkels, "Full TCP/IP for 8-Bit Architectures," *MobiSys International Conference on Mobile Systems*, 2003.

[4] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008.

[5] J. Herter, P. Backes, F. Haupenthal, and J. Reineke, "CAMA: A Predictable Cache-Aware Memory Allocator," in *To Appear in: Proceedings of the 23st Euromicro Conference on Real-Time Systems (ECRTS '11)*. IEEE Computer Society, July 2011.

[6] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache Behavior Prediction by Abstract Interpretation," pp. 52–66, 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=760063

[7] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[8] P. Cousot, "Abstract Interpretation Based Formal Methods and Future Challenges," in *Informatics - 10 Years Back. 10 Years Ahead.* London, UK: Springer-Verlag, 2001, pp. 138–156. [Online]. Available: http://portal.acm.org/citation.cfm?id=724445

[9] T. Lev-Ami and S. Sagiv, "TVLA: A System for Implementing Static Analyses," in *Proceedings of the 7th International Symposium on Static Analysis*, ser. SAS '00. London, UK: Springer-Verlag, 2000, pp. 280–301. [Online]. Available: http://portal.acm.org/citation.cfm?id=647169.718161

[10] GNU Linear Programming Kit. [Online]. Available: http://www.gnu.org/software/glpk/