PRECOMPUTING MEMORY LOCATIONS FOR PARAMETRIC ALLOCATIONS[1]

Jörg Herter[2] and Sebastian Altmeyer[2]

Abstract

Current worst-case execution time (WCET) analyses do not support programs using dynamic memory allocation. This is mainly due to the unpredictability of cache performance introduced by standard memory allocators. To overcome this problem, algorithms have been proposed that precompute static allocations for dynamically allocating programs with known numeric bounds on the number and sizes of allocated memory blocks. In this paper, we present a novel algorithm for computing such static allocations that can cope with parametric bounds on the number and sizes of allocated blocks. To demonstrate the usefulness of our approach, we precompute static allocations for a set of existing real-time applications and academic examples.

1. Introduction

In modern embedded hardware, caches are used to bridge the increasing gap between processor speed and memory access times. For a timing analysis striving to derive tight bounds on a program's worst-case execution time these caches impose additional challenges. Such an analysis may not conservatively assume each memory access to be a cache miss without risking to be overly imprecise as turning off the cache completely may lead to a thirty fold increase of the execution time [5]. Also, lower bounds on the number of cache hits are not necessarily leading to tight bounds due to *timing anomalies* [7]. Timing anomalies denote situations where local worst-case behavior, i.e. a cache miss, does not always lead to global worst-case behavior. Hence, a statically more predictable cache behavior enables the derivation of tighter bounds on the programs worst-case execution times.

In the presence of dynamic memory allocation, WCET analyses fail to determine precise time bounds due to the unpredictability dynamic memory allocation inflicts on the cache behavior. General purpose dynamic memory allocators strive to cause little fragmentation and neither provide guarantees about their own worst-case execution time, nor do they provide information about the cache set mapping of the memory addresses they return. The cache set that a dynamically allocated memory block is mapped to is therefore statically not predictable. Consequently, WCET analyses can, in general, not classify accesses to dynamically allocated memory as cache hits or cache misses. Additionally, the processes of dynamic memory allocation as well as deallocation pollute the cache themselves. Memory allocators manage free memory blocks in internal data structures which they maintain and traverse during allocation and deallocation operations. These unpredictable traversals of internal data structures result in unpredictable influences on the cache.

[1]This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008 (Predator).

[2]Saarland University, Saarbrücken, Germany, {jherter, altmeyer}@cs.uni-saarland.de

To circumvent these problems, programmers revert to static memory allocation within hard real-time programs. However, often dynamic memory allocation has advantages over static memory allocation. To enable dynamic memory allocation for hard real-time applications, Herter and Reineke proposed an algorithm to statically precompute memory addresses and replace calls to the memory allocator by sequences of fixed addresses [3]. This way, programmers can use dynamic memory allocation to alleviate the task of efficiently reusing memory, while the program itself can—after an automatic transformation—be analyzed by current WCET analyses.
However, the algorithm proposed in [3] is limited to programs that contain only loops bounded by a numerical value. Additionally, the requested block sizes—or at least tight bounds on these—need to be statically known. In this paper, we propose a novel algorithm that can cope with parametric bounds on the number of possible loop iterations and requested block sizes.

Figure 1 visualizes the application area for the proposed algorithm. We start with a program using dynamic memory allocation that is intended to be deployed in a hard real-time setting. However, due to the unpredictability that dynamic memory allocation inflicts on the cache behavior of the program, no tight bounds for the program's worst-case execution can be determined using current WCET analyses.

In a first phase, we apply a static program analysis to compute liveness information for the dynamically allocated objects. This information together with user supplied loop and recursion bounds is used as input for the proposed algorithm for precomputing static memory addresses. In the second and final phase, we replace calls for memory allocation by functions that return a sequence of precomputed addresses. Using fixed memory addresses, calls to `free` become obsolete and can simply be removed. This phase yields a modified program in which the memory addresses of all objects are statically known. For such a program, current WCET analyses can compute tight bounds on its worst-case execution time.

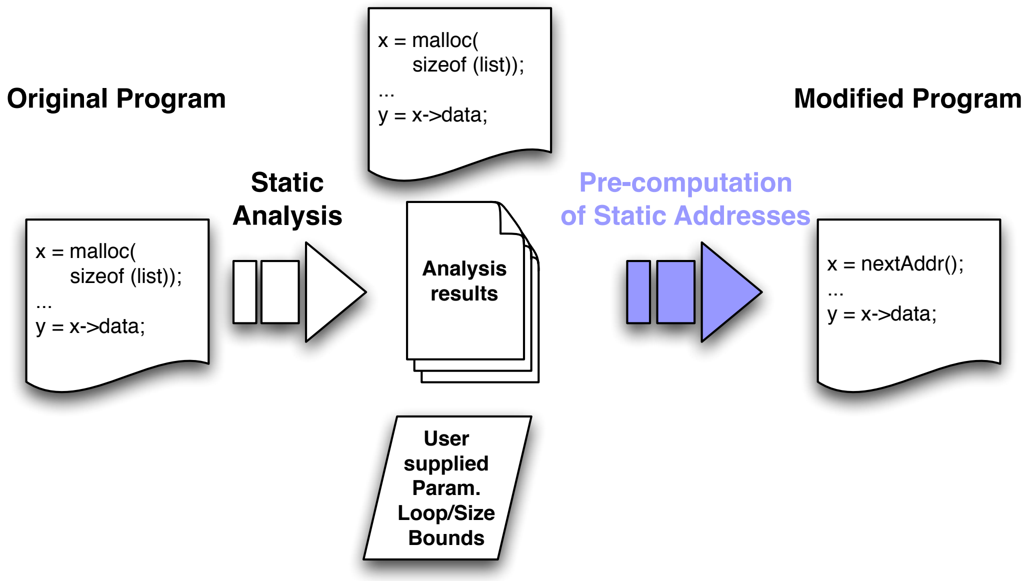

Original Program
x = malloc(sizeof (list));
...
y = x->data;
Static Analysis
x = malloc(sizeof (list));
...
y = x->data;
Analysis results
User supplied Param. Loop/Size Bounds
Pre-computation of Static Addresses
Modified Program
x = nextAddr();
...
y = x->data;

Figure 1. Field of application for the proposed algorithm

The remainder of this paper is organized as follows. Section 2 briefly reviews related work. In Section 3, we formally describe a program's memory allocation behavior. The algorithm presented in Section 4 uses such a formal description to statically precompute memory addresses for the allocated memory blocks. In Section 5, we present results obtained from precomputing memory allocations for a set of academic and real-life programs.

2. Related Work

There are two other approaches to make programs that dynamically allocate memory more analyzable with respect to their WCET. In [4], Herter et al. propose to utilize a predictable memory allocator to overcome the problems introduced by standard memory allocators. Schoeberl proposes different (hardware) caches for different data areas [10]. Hence, accesses to heap-allocated objects would not influence cached stack or constant data. The cache designated for heap-allocated data would be implemented as a fully-associative cache with an LRU replacement policy. For such an architecture it would be possible to perform a cache analysis without knowledge of the memory addresses of the heap-allocated data. However, a fully-associative cache, in particular with LRU replacement, is limited in size due to technological constraints. Those two approaches have the advantage not to rely on tight bounds on the number of allocation requests and the sizes of requested memory blocks.

An algorithm to statically precompute memory addresses and replace calls to the memory allocator by fixed addresses for programs in which all occurring loops and requested sizes can be (tightly) bounded by a numerical value was proposed in [3]. This approach, although applicable to a smaller class of applications, has the advantage to yield entirely static allocations, removing memory allocation procedures completely.

3. A Formal Model for Memory Allocations

To describe a program's memory allocation behavior, we start by collecting all allocation sites, i.e. occurrences of `malloc` within the program, in a set M. Per assumption, we know how often each allocation site can be reached during program allocation as loop and recursion bounds are known, at least parametrically. Hence, we 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 may be invoked. P denotes the set of parametric loop and recursion bounds. For each allocation site m, we construct a function f_m such that $f_m(i)$ evaluates to the size of the memory block requested the i-th time allocation site m is reached. These sizes may be over-approximated by intervals. And consequently,

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

where $\mathcal{I}_m$ is a set of intervals, is the set containing functions describing the allocation requests for all allocation sites. Now, 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.

For precomputing feasible memory allocation we also need to know which allocated blocks have overlapping lifetimes. Liveness information for dynamically allocated memory blocks can safely be over-approximated using shape analysis [9]. We assume this information to be available and encode it in a conflict function

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

that evaluates to 1 iff its argument requests at least two memory blocks with overlapping lifetimes.

When precomputing static memory addresses for originally dynamically allocated memory blocks, we may not want to ignore cache set mappings completely. Hence, we are presented two options. 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 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. The second option would be to strive for good cache performance such that the risk is reduced 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 [8]. 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 [1]. To exploit this heuristics, we construct a bias function

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

such that $\mathcal{B}(r_1, r_2)$ evaluates to 1 iff the block requested in r_1 is likely to be accessed prior to the one requested in r_2. 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 can also approximate object access behavior using shape analysis [9]. 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 a data structure analysis [6] together with the heuristics that objects organized in the same data structure are likely to be accessed contemporaneously.

An allocation problem is then a six-tuple

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

where L is a set of constraints on the parameters in P.

An allocation is a feasible solution to an allocation problem of the form

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

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

An optimal allocation 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.

Finding such optimal solutions is still at least NP-hard. Let (V, E, k) be a given instance of the k-colorability problem for a graph $G = (V, E)$. Generate the allocation problem

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

where B maps all arguments to 0 and C is defined s.t.

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

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.

We choose a heuristic approach to finding *good* solutions as striving for optimal solutions would render our technique only applicable to very small applications.

4. Algorithm

Before starting to precompute suitable memory addresses for allocated memory blocks, the algorithm transforms its input $I = (M, U, L, A, \mathcal{C}, \mathcal{B})$ as follows. Using the bias function $\mathcal{B}$, maximal ranges of allocated memory blocks that should be placed adjacent in memory are identified. These ranges are then split again into blocks, such that the sizes of the resulting *normalized blocks* are multiples of the size of a cache line. The last normalized block may be of smaller size and splitting must not occur within the bounds of an allocated memory block. This transformation yields a new allocation problem $I' = (M', U', L', A', \mathcal{C}')$ for a parametric set of normalized blocks. M' denotes now the set of maximal ranges of allocated memory blocks, or *abstract allocation sites*, while the functions collected in A' map to the sizes of the single normalized blocks of these ranges. $U' \subset \mathbb{N} \cup P'$ and L' are the accordingly updated constraints on loop/recursion bounds and parameters. We gain two advantages from this transformation step. The number of blocks considered by the algorithm is in general reduced, leading to a smaller problem instance. Also, the bias function was consumed and the algorithm does not need to respect further constraints introduced by this function.

Given the input I', the 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 *memory block chunks*. A chunk is a relative placement of memory blocks from different abstract allocation sites in M', such that there is no conflict within a chunk itself. Chunks are then placed sequentially in memory such that 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-th request from abstract allocation site m is located within the chunk at relative position o. An allocation, i.e. solution to an allocation problem, contains several types of chunks and the number of occurrences of a specific chunk is parametric. By this, we reduce the computation of a solution to an allocation problem to finding an appropriate selection of chunks.

Let us consider a standard example for advantageous use of dynamic memory allocation, namely in-situ list copy. Assume, we want to copy a singly-linked list M_s to a doubly-linked list M_d with minimal memory consumption. The size of a list is 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-th element of the singly-linked list has a conflict with the j-th element of the doubly-linked iff $j < i$. An allocation scheme with minimal memory consumption is given in Figure 2a. Note that a memory optimal allocation is not necessarily unique, neither are optimal solutions in general. Figure 2b gives a second possible mapping for our list example with minimum memory consumption.

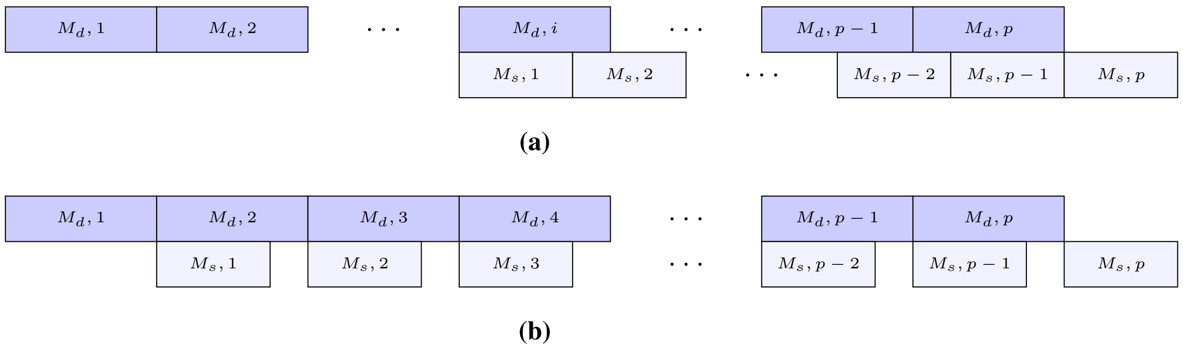


Figure 2. Allocation schemes for list-copy with minimal memory consumption

What set of chunks would we like our algorithm to compute? With the additional constraint that the size of a chunk is determined by the size of the largest normalized memory block it contains, we would anticipate a solution set

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

Figure 3 shows these chunks and their corresponding number of occurrences. Putting these chunks consecutively in memory yields the optimal allocation scheme shown in Figure 2b.

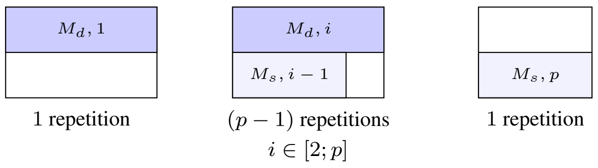


Figure 3. Allocation chunks for the list-copy example

We distinguish 2 kinds of chunks, singleton and repetitive chunks. A singleton chunk is a chunk that is generated exactly once, while multiple instances of repetitive chunks are generated. In our list example, the algorithm generates 2 singleton and 1 repetitive chunks.

Our algorithm to compute such sets of chunks for a given allocation problem works as follows. The algorithm maintains a workset of unprocessed, i.e. not yet located within a chunk, requests for normalized blocks. This set is initialized with the set

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

While the workset is not empty, the algorithm creates singleton chunks followed by sequences of repetitive chunks. Algorithm 1 gives the pseudo code for this main routine.

```
Algorithm 1: Algorithm to compute a suitable set of chunks
Data: Problem specification $I' = (M', U', L', A', \mathcal{C}')$
Result: Allocation scheme as a set of chunks
workset = Set of requests $R$ obtained from $M'$, $U'$ and $A'$;
while $workset \neq \emptyset$ do
    createChunk(workset, true); // first chunk – unrolling
    removeProcessedRequests();
    if $workset = \emptyset$ then break;
    createChunk(workset, false); // create repetitive chunk
    computeRepetitions(); // repeat last chunk
    removeProcessedRequests();
end
```

The function createChunk creates new chunks and adds normalized blocks until no further blocks are requested for a given abstract allocation site or no further blocks can be added without either causing a conflict or exceeding 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 2 gives the pseudo code for this function.

```
Algorithm 2: createChunk
Data: Set of requests $R$, boolean isSingleton
Result: Chunk
if ¬ IsSingleton then sortByRequestSize($R$)
for $(m, i) \in R$ do
    boolean added = true;
    while $i < u_m \wedge added$ do
        added = addRequestToChunk();
        if added then i++;
    end
end
```

Requests or normalized blocks are added to a given chunk in the following way. If the chunk is empty, the request is always added at the first position of the chunk and the size of the chunk is set to the size of the first request. Subsequent blocks are temporarily placed at position $p = 0$. In case this does not cause conflicts, the request is added and the algorithm returns true. While conflicts do occur, the subsequent request is shifted to the next position $p + 1$ until either all conflicts are solved and the requested block is added or no space in the chunk is left and the block is not added. Algorithm 3 gives the pseudo code for this operation.

```
Algorithm 3: addMallocToChunk
Data: Request $r$
Result: boolean added
if chunk.isEmpty() then
    addAtZero($r$);
    return true;
end
int pos = 0;
while $pos \leq ChunkSize - sizeOf(r)$ do
    If $\neg$ conflictInChunk() return true;
    pos++;
end
return false;
```

The problem specification may contain several parameters and 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 problem specification may be used to decide these conditions. If the set L is not sufficient to allow for deciding conditionals, we split the specification depending on the various outcomes.

For instance, a conditional

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

leads to the following two problem specifications:

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

and

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

Hence, each time the set of restrictions on parameters, L_S, does not contain enough information to decide whether a conditional c 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_{S_t} to $L_S \cup \{c\}$, and accordingly in S_f to $L_S \cup \{\neg c\}$.

5. Experiments

We did a preliminary evaluation of our algorithm using two programs that perform an in-situ copy from one data structure to another as well as a small set of existing (hard) real-time programs taken from the MiBench benchmark suite [2].

In-Situ Copy The memory allocation behavior of a program copying a singly-linked list to a doubly-linked list as used as an example in Section 4 can be formalized as

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

With the intended meaning that there are two allocation sites, one for the elements of the singly-linked list, one for those of the doubly-linked one. 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. $\mathcal{C}_{lc}$ is constructed, such that there are conflicts between list elements that are in-use at the same time. $\mathcal{B}_{lc}$ evaluates to 0 for all inputs, i.e. no bias is given, to prevent bias disabling the algorithm to compute memory optimal solutions. Our algorithm was able to compute for such a program description a memory optimal set of chunks as depicted in Figure 2b. With $\mathcal{B}_{lc}$ constructed such that elements adjacent within a list are to be put adjacent in memory, our algorithm computes a similar set of chunks as depicted in Figure 4. This normalization yields blocks of size 96 KB and 32 KB for the doubly- and singly-linked list, composed of 8 and 4 objects of the original lists, respectively. However, the repetitive chunk contains now two blocks from the singly-linked list together with one block from the doubly-linked one.

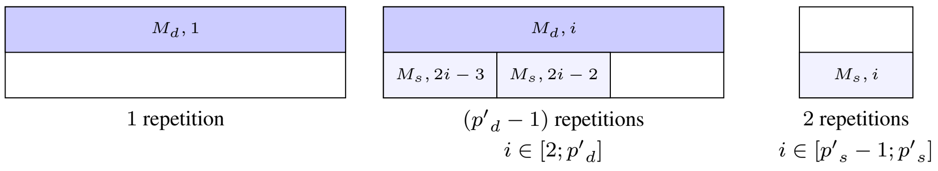


Figure 4. Allocation chunks for normalized in-situ list copy

Consider next the reverse list-copy from doubly-linked to singly-linked elements

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

with parametric sizes. At allocation site m_s blocks of size q KB are requested, at site m_d 12 KB blocks, with $4 < q \leq 12$. On this example, our algorithm computes the following solutions.

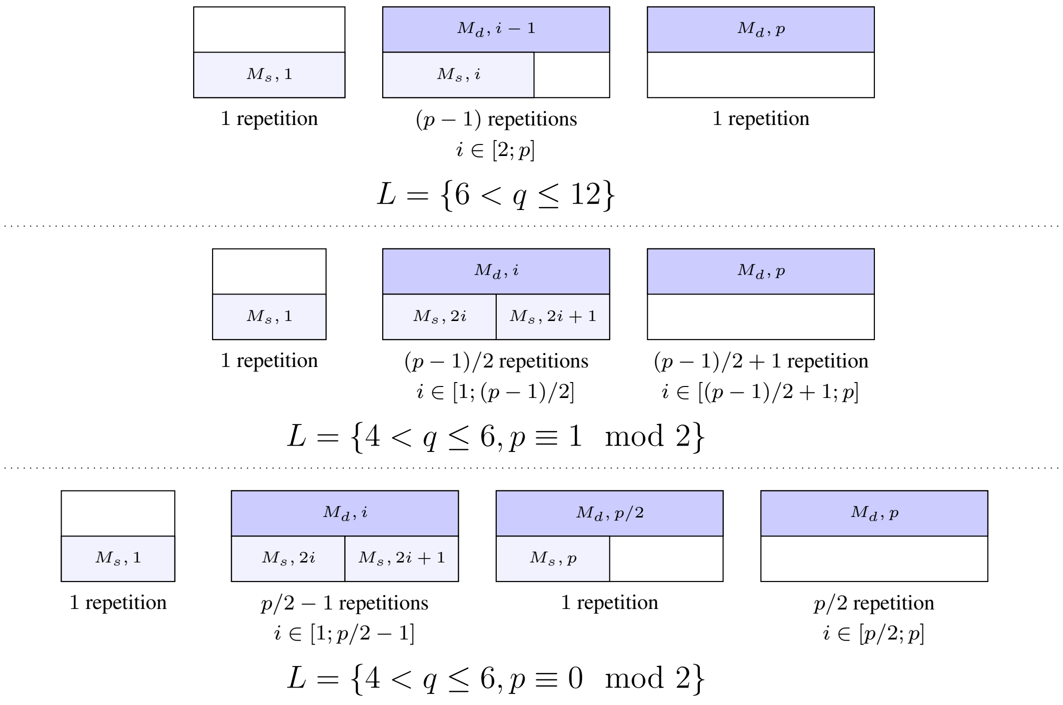
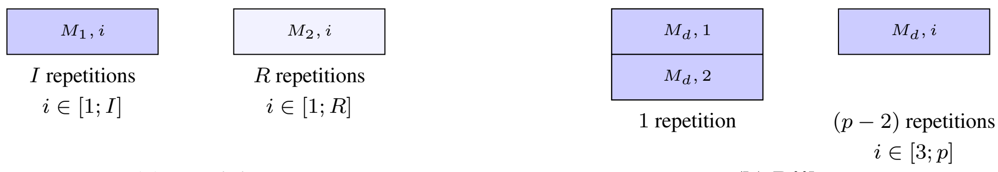


Figure 5. Allocation chunks for the reversed in-situ list-copy example

Patricia (MiBench) 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. This application uses patricia tries to construct a routing table. The benchmark program is formalized to

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

where $M_p = \{1, 2\}, U_p = \{I, R\}$, $A_p = \{f_1 : \mathbb{N}^{\leq I} \mapsto [8 \cdot ml, 8 \cdot ml], f_2 : \mathbb{N}^{\leq I} \mapsto [8 \cdot ml, 8 \cdot ml]\}$, and $\mathcal{C}_p(C) = 1$. The parameter ml denotes a variable value, possibly determinable by a value analysis. As we cannot safely determine that two allocated blocks are not contemporaneously in-use, our algorithm is not able to compute a better memory allocation than the one given in Figure 6a. This application shows one limitation of precomputing memory addresses, namely that a previous analysis must be able to gather precise liveness information regarding which allocated blocks are alive at the same time.



(a) Patricia **(b) Dijkstra**

Figure 6. Allocation chunks for the Patricia and Dijkstra test cases

Dijkstra (MiBench) `Dijkstra` constructs a large graph (as an adjacency matrix) and then computes the shortest paths between pairs of nodes using repeated applications of Dijkstra's algorithm. The program can be described by

$$(M_d, U_d, \{\}, A_d, \mathcal{C}_d, \mathcal{B}_d)$$

where $M_d = \{1\}$, $\{u_1\} = \{n^2\}$, $A_d = \left\{\left\{x \mapsto [16, 16] \mid x \in \mathbb{N}^{\leq n^2}\right\}\right\}$, and

$$\mathcal{C}_d(C) = \begin{cases} 1 & \text{if } \exists i, j \in C. j \in (i, (i-1) \cdot n] \\ 0 & \text{otherwise} \end{cases}$$

Here, n is the number of nodes of the constructed graph. For this application, our algorithm computed a chunk set as depicted in Figure 6b.

Susan (MiBench) `Susan` is an image processing application used to determine the position of edges and/or corners within the input image for guidance of unmanned vehicles. Its allocation behavior can be formalized to

$$(M_s, U_s, \{\}, A_s, \mathcal{C}_s, (R \times R) \mapsto \{0\})$$

where $M_s = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $\bigcup_{m \in M_s} u_s = \{1\}$, $A_s = \{f_1 : \{1 \mapsto x \cdot y\}, f_2 : \{1 \mapsto 516\}, f_3 : \{1 \mapsto (14 + x)(14 + y)\}, f_4 : \{1 \mapsto 16\}, f_5 : \{1 \mapsto 4 \cdot x \cdot y\}, f_6 : \{1 \mapsto 4 \cdot x \cdot y\}, f_7 : \{1 \mapsto 4 \cdot x \cdot y\}, f_8 : \{1 \mapsto x \cdot y\}, f_9 : \{1 \mapsto 4 \cdot x \cdot y\}\}$, and

$$\mathcal{C}_s(C) = \begin{cases} 1 & \text{if } \{(1, 1)\} \subsetneq C \vee \{(7, 1), (8, 1)\} \subseteq C \vee \{(3, 1), (4, 1)\} \subseteq C \\ 0 & \text{otherwise} \end{cases}$$

Here, x and y are fixed parameters determining the size of the processed images. Again, our algorithm computed a memory optimal chunk set as depicted in Figure 7.

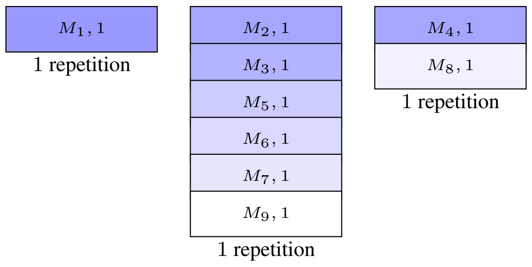


Figure 7. Allocation chunks for the Susan test case

6. Conclusions

Statically precomputing memory addresses for otherwise dynamically allocated blocks yields significant advantages. Compared to striving for predictable memory allocation, predictability of the program is increased as addresses of (heap) objects become statically known. Furthermore, with allocation and deallocation removed completely from the program, so are unpredictability resulting from cache pollution caused by allocators and the uncertain response times of (de)allocation routines removed. However, static precomputation is not applicable to all hard real-time applications. While the algorithm presented in this paper increases the class of programs that such an approach can cope with, there are still limitations. The Patricia benchmark showed that if such an approach is to preserve the main advantage of dynamic allocation, efficient memory reuse, precise information about which blocks may be allocated at the same time must be available to the algorithm. Hence, such an approach relies on precise static preanalyses to yield results that enable similar program performance compared to a program using dynamic memory allocation. Without such analyses, predictability comes at the price of overly high memory consumption. Furthermore, too little information about the relations between parameters can lead to an overly large solution set, as each time decisions cannot be made due to incomplete information, the algorithm considers two cases, splitting to two solution paths. However, given decent information about the program to transform, our algorithm showed very promising results. An automatic analysis to gather these informations as well as an exhaustive evaluation of the presented algorithm are to be tackled next.

References

[1] CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. Making pointer-based data structures cache conscious. *Computer 33*, 12 (2000), 67–74.

[2] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 3–14.

[3] HERTER, J., AND REINEKE, J. Making dynamic memory allocation static to support WCET analyses. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis* (June 2009).

[4] HERTER, J., REINEKE, J., AND WILHELM, R. CAMA: Cache-aware memory allocation for WCET analysis. In *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems* (July 2008), M. Caccamo, Ed., pp. 24–27.

[5] LANGENBACH, M., THESING, S., AND HECKMANN, R. Pipeline modeling for timing analysis. *Proceedings of Static Analysis Symposium (SAS) 2477* (2002).

[6] LATTNER, C., AND ADVE, V. Data structure analysis: A fast and scalable context-sensitive heap analysis. Tech. rep., 2003.

[7] LUNDQVIST, T., AND STENSTRÖM, P. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)* (Washington, DC, USA, 1999), IEEE Computer Society, p. 12.

[8] PETRANK, E., AND RAWITZ, D. The hardness of cache conscious data placement. *Nordic J. of Computing 12*, 3 (2005), 275–307.

[9] SAGIV, M., REPS, T., AND WILHELM, R. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. 24*, 3 (2002), 217–298.

[10] SCHOEBERL, M. Time-predictable cache organization. In *STFSSD '09: Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 11–16.