MAKING DYNAMIC MEMORY ALLOCATION STATIC TO SUPPORT WCET ANALYSES[1]

Jörg Herter[2] and Jan Reineke[2]

Abstract
Current worst-case execution time (WCET) analyses do not support programs using dynamic memory allocation. This is mainly due to the unpredictable cache performance when standard memory allocators are used. We present algorithms to compute a static allocation for programs using dynamic memory allocation. Our algorithms strive to produce static allocations that lead to minimal WCET times in a subsequent WCET analyses. Preliminary experiments suggest that static allocations for hard real-time applications can be computed at reasonable computational costs.

1. Introduction

High cache predictability is a prerequisite for precise worst-case execution time (WCET) analyses. Current static WCET analyses fail to cope with programs that dynamically allocate memory mainly because of their unpredictable cache behavior. Hence, programmers revert to static allocation for hard real-time systems. However, sometimes dynamic memory allocation has advantages over static memory allocation. From a programmer's point of view, dynamic memory allocation can be more natural to use as it gives a clearer program structure. In-situ transformation of one data structure into another one may reduce the necessary memory space from the sum of the space needed for both structures to the space needed to store the larger structure.

But why do WCET analyses fail to cope with dynamic memory allocation? In order to give safe and reasonably precise bounds on a program's worst-case execution time, analyses have to derive tight bounds on the cache performance, i.e., they have to statically classify the memory accesses as hits or misses. Programs that dynamically allocate memory rely on standard libraries to allocate memory on the heap. Standard malloc implementations, however, inhibit a static classification of memory accesses into cache hits and cache misses. Such implementations are optimized to cause little fragmentation and neither provide guarantees about their own execution time, nor do they provide guarantees about the memory addresses they return. The cache set that memory allocated by malloc maps to is statically unpredictable. Consequently, WCET analyses cannot predict cache hits for accesses to dynamically allocated memory. Additionally, dynamic memory allocators pollute the cache themselves. They manage free memory blocks in internal data structures which they maintain and traverse during (de)allocation requests. An unpredictable traversal of these data structures results in an equally unpredictable influence on the cache.

While it seems that dynamic memory allocation prohibits a precise WCET analysis, the special re-

[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), the German-Israeli Foundation (GIF) in the "Encasa" project, and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008 (Predator).

[2]Universität des Saarlandes, Saarbrücken, Germany, {jherter, reineke}@cs.uni-saarland.de

quirements for hard real-time systems can be utilized to circumvent predictability issues introduced by dynamic allocation. Hard-real time software contains no unbounded loops nor unbounded recursion and hence no unbounded allocation. In this work, we show how information necessary for WCET analysis on hard real-time systems (like the aforementioned loop and recursion bounds) can be used to transform dynamic memory allocation into static memory allocation. Our proposed algorithm statically determines 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.

We evaluate our approach on several academic example programs.

The following subsection gives an overview on related work, while Section 2 illustrates the importance of cache predictability for WCET analyses and the problems introduced by dynamic memory allocation. In Section 3, we propose algorithms to compute static allocations from program descriptions to substitute dynamic allocation by. Experimental results are given in Section 4, further improvements of our approach are sketched in Section 5. Section 6 concludes the paper.

1.1. Related Work

There are two other approaches to make programs that dynamically allocate memory more analyzable with respect to their WCET. In [4] we proposed to utilize a predictable memory allocator to overcome the problems introduced by standard memory allocators. Martin 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, cannot be very big, due to technological constraints.

2. Caches, Cache Predictability, and Dynamic Memory Allocation

Caches are used to bridge the increasing gap between processor speeds and memory access times. A cache is a small, fast memory that stores a subset of the contents of the large but slow main memory. It is located at or near the processor. Due to the *principle of locality*, most memory accesses can be serviced by the cache, although it is much smaller than the main memory. This enables caches to drastically improve the average latency of memory accesses. To reduce cache management and data transfer overhead, the main memory is logically partitioned into a set of *memory blocks* of size b. Memory blocks are cached as a whole in *cache lines* of equal size. When accessing a memory block, the system has to determine whether the memory block is currently present in the cache or if it needs to be fetched from main memory. To enable an efficient look-up, each memory block can be stored in a small number of cache lines only. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set, i.e., the number of cache lines it consists of, is called the *associativity* k of the cache. At present, k ranges from 1 to 32. Since the number of memory blocks that map to a set is usually far greater than the associativity of the cache, a so-called *replacement policy* must decide which memory block to replace upon a cache miss. *Cache analyses* [2] strive to derive tight bounds on the cache performance. To obtain such bounds, analyses have to classify memory accesses as cache hits or cache misses. A memory access constitutes a *cache hit* if it can be served by the cache. And analogously, *cache miss* denotes a memory access that cannot be served by the cache; instead the requested data has to be fetched from main memory. The better the classification of accesses as cache hits or cache misses is, the tighter are the obtained bounds on the cache performance. To classify memory accesses, a cache analysis needs to know the mapping of program data to cache sets. Otherwise, it does not know which memory blocks compete for the cache lines of each cache set. Using standard dynamic memory allocators, like for instance [7], no knowledge about the mapping of allocated data structures to cache sets is statically available. Assume, for example, a program would allocate 6 memory blocks to hold objects of a singly-linked list. Two possible mappings from cache sets to those objects, assuming a 4-way cache-associative cache with 4 cache sets, are given in Figure 1 (a) and (b), respectively.

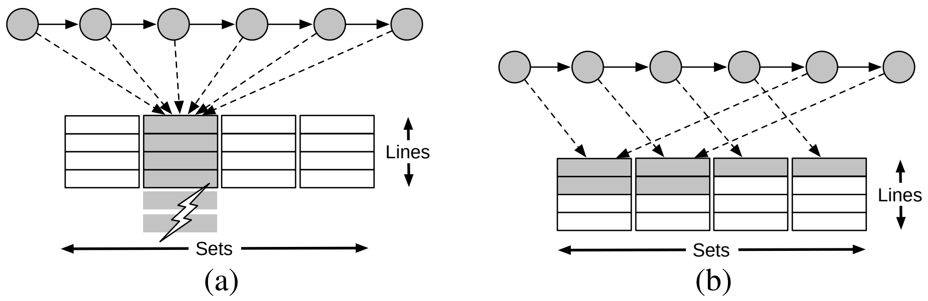


Figure 1. Two possible cache mappings of 6 dynamically allocated objects organized in a singly-linked list.

For the mapping in Figure 1 (a) a further traversal of the list would result in no cache hits at all. Given the mapping depicted in Figure 1 (b), a subsequent traversal of the list would result in 6 cache hits. A conservative cache analysis with no knowledge about the addresses of allocated memory has no other option than to classify all accesses to allocated memory as cache misses, while during actual program runs potentially all accesses may be cache hits. In modern processors, turning off the cache can easily cause a thirty-fold increase in execution time [6], hence, conservatively treating all accesses as cache misses yields very imprecise and thus useless analysis results.

There is a second problem due to dynamic memory allocators besides the resulting inability to guarantee cache hits for the dynamically allocated data. Due to the unpredictable mapping of such data to cache sets, knowledge derived about the caching of statically allocated data is lost if dynamically allocated data is accessed. As it is not known to which cache set newly allocated memory maps to, the analysis has to conservatively treat such memory as potentially mapping to every cache set and evicting cached content from these sets.

Also the manner in which dynamic allocators determine free memory addresses significantly decreases cache predictability. Unused memory blocks are managed in some internal data structure or data structures by the allocators. Upon allocation requests, these structures are traversed in order to find a suitable free block. This traversal is in general statically unpredictable and leads to an equally unpredictable cache pollution as all memory blocks visited during traversal are loaded into the cache. As it is unpredictable which and also how many free blocks are considered upon an allocation request, the response time of the request is also not predictable. Only upper bounds with potentially high overestimations can be given for the execution time of a call of the memory allocation procedure. For a

WCET analysis that relies on tight execution time bounds this is highly undesirable.

In summary, WCET analyses face three problems when dynamic memory allocation is employed: (1) the mapping to cache sets of allocated data is unpredictable, (2) malloc itself causes unpredictable cache pollution, and (3) no bounds on the response times of malloc are guaranteed.

3. Precomputing Static Memory Addresses

To avoid the described problems with dynamic memory allocators, we propose to utilize available knowledge about the program used in a WCET analysis to substitute dynamic memory allocation by static allocation. For hard real-time applications, a good memory mapping has to enable the computation of small WCET bounds.

What do we mean by a memory mapping? We partition the main memory into *memory blocks* the size of a cache line, s.t. each block maps into exactly one cache set, i.e., the memory blocks are aligned with the cache sets. As we will later address the memory at the granularity of memory blocks and never below, *memory address* $(i-1)$ refers to the i-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 assigns to each heap allocated object a memory block.

A heap allocated object may contain several or just a fraction of a single program object(s). 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. To this end, we introduce *heap allocated structures*: ordered sets of heap allocated objects. The heap allocated objects of a heap allocated structure shall be mapped to consecutive memory addresses.

We can relate heap allocated structures either to dynamically allocated program objects, like for example single nodes of a binary search tree, or to entire data structures built from dynamically allocated objects, like for example a linked list consisting of several node objects. What relation we choose depends on the program we want to analyze. For example, it may be advantageous to consider a linked list structure to be a single heap allocated structure, while for larger objects organized in a tree structure we might want to consider each single object a heap allocated structure.

Hence, formally, a memory mapping m is a function that maps each heap allocated memory 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 of structure i.

What we want to compute for a program P is a memory mapping that allows for a WCET bound on P of

$$\min_{\substack{\text{memory}\\\text{mapping } m}} \text{WCET}(P, m)$$

by using as little memory as possible.

WCET Computation by IPET How can we compute WCET(P, m)? As proposed by Li and Malik, the longest execution path of a program can be computed by implicit path enumeration techniques (IPET) [8]. Their approach formulates the problem of finding the longest execution path as an integer linear program (ILP). Given the control-flow graph $G(P) = (V, E, s, e)$ of a program P and loop bounds b_k for the loops contained in P, this ILP 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 (1). Each node is executed as often as the control flow enters and leaves the node ((2) and (3)). And finally, equation (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, we add a constraint ensuring that each outgoing edge of the first block b within the loop is taken at most as often as the the sum over all ingoing edges to b times the loop bound b_l.

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

$$\sum_{j \in \mathcal{J}} y_j = x_k \quad \text{where} \quad \mathcal{J} = \{j \mid E_j = (\cdot, B_k)\} \tag{2}$$

$$\sum_{j \in \mathcal{J}} y_j = x_k \quad \text{where} \quad \mathcal{J} = \{j \mid E_j = (B_k, \cdot)\} \tag{3}$$

$$y_l \leq b_l \cdot \left(\sum_{j \in \mathcal{J}} y_j \right) \quad \text{where} \quad \mathcal{J} = \{j \mid E_j \text{ is loop entry edge to } l\} \tag{4}$$

The WCET bound is then obtained using the objective function:

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

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 A WCET-optimal memory mapping yields a WCET bound equal to

$$\min_{\substack{\text{memory} \\ \text{mapping } m}} \max \sum_{i \in \{i \mid B_i \in V\}} x_i c_i^m \tag{5}$$

However, this problem is no ILP anymore.
We propose the following heuristic approach to compute approximate solutions to (5). Initially, we start with a memory mapping m that uses minimal memory. We then compute the WCET of the program for the memory mapping m using an IPET as described above. An improved mapping can then be obtained by selecting the basic block B_i whose penalty due to conflict misses has the greatest contribution to the WCET bound and modifying m such that c_i^m is minimized. The last step can be repeated until no further improvements can be achieved.
Algorithm 1 implements this strategy as a hill climbing algorithm that internally 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. With side steps, we mean that the algorithm is allowed to select 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 $2 \cdot |\mathcal{B}|$, where $|\mathcal{B}|$ is the number of program blocks.

An ILP formulation to implement $\text{WCET}_{\text{IPET}}()$ has already been introduced. We can also implement $mem\_opt()$ and $block\_opt()$ as ILPs as follows.

Memory Optimal Mapping Let $a_{i,j}$ be an integer variable of an ILP storing the memory address of the j-th heap allocated object of heap allocated structure i. Again, with memory address, we do not mean the physical address, but rather the $a_{i,j}$-th memory block by partitioning the memory into blocks of the size of a cache line. For all i, j, i', j' s.t. the j-th object of structure i may be allocated at the same time as the j'-th object of structure i', we add two constraints:

$$
\begin{aligned}
a_{i',j'} + 1 - a_{i,j} - b_{i,j,i',j'} \cdot \mathcal{C} &\leq 0 \qquad (6)\\
a_{i',j'} - (a_{i,j} + 1) + (1 - b_{i,j,i',j'}) \cdot \mathcal{C} &\geq 0 \qquad (7)
\end{aligned}
$$

where the $b_{a,b,c,d}$ are auxiliary binary variables and $\mathcal{C}$ a constant larger than the value any expression within the ILP can take. Such a $\mathcal{C}$ may be computed statically. These constraints ensure that parts of structures allocated at the same time do not reside at the same memory address. To enforce a consecutive placement of parts of the same structure, we add for all i, j

$$a_{i,j+1} = a_{i,j} + 1 \qquad (8)$$

Computation of the largest used memory address $\bar{a}$ is done by the following constraints for all i, j:

$$a_{i,j} < \bar{a} \qquad (9)$$

We want to minimize our memory consumption, hence we use the following objective function in order to minimize the largest address in use:

$$\min \bar{a} \qquad (10)$$

```
Algorithm 1: Hill climbing algorithm to compute a suitable memory mapping
Data: functions WCET_IPET(), $mem\_opt()$, and $block\_opt()$; $sideSteps$ number of allowed side
      steps
Result: near WCET-optimal memory mapping for dynamically allocated objects
$mapping_{best} \leftarrow mem\_opt()$;
$mapping_{curr} \leftarrow mapping_{best}$;
while $sideSteps > 0$ do
    calculate WCET_IPET($mapping_{curr}$);
    select basic block $b$ with largest WCET contribution due to conflict misses;
    $mapping_{tmp} \leftarrow block\_opt(mapping_{curr}, b)$;
    if $WCET_{IPET}(mapping_{tmp}) < WCET_{IPET}(mapping_{curr})$ then
        $mapping_{curr} \leftarrow mapping_{tmp}$;
        if $WCET_{IPET}(mapping_{tmp}) < WCET_{IPET}(mapping_{best})$ then
            $mapping_{best} \leftarrow mapping_{tmp}$;
        end
    else
        $sideSteps \leftarrow sideSteps - 1$;
        $mapping_{curr} \leftarrow mapping_{tmp}$;
    end
end
```

Block Optimal Mapping To get a block optimal memory mapping with respect to potential conflict cache misses, we modify the ILP used to compute a memory usage optimal mapping as follows. We first compute the cache sets to which memory blocks are mapped that are accessed in the basic block we want to optimize the mapping for.
Let $\#cs$ denote the number of cache sets and $cs_{a,i,j}$ denote binary variables set to 1 iff the j-th block of structure i is mapped to cache set a. The following set of constraints (for each a, i, j, s.t. a is a cache set and block j of structure i is accessed in the considered basic block) sets these $cs_{a,i,j}$ accordingly. However, several auxiliary variables have to be set previously. Let $o_{i,j}$ denote the j-th block of structure i. Then $a_{i,j}$ stores the cache set to which $o_{i,j}$ is mapped, and $n_{i,j}$ stores the result of an integer division of the address of $o_{i,j}$ by $\#cs$. Furthermore, $y_{a,i,j}$ is set to $|a - cs_{i,j}|$ and $ltz_{a,i,j}$ to 0 iff $a - cs_{i,j} < 0$.

$$
\begin{aligned}
a_{i,j} - n_{i,j} \cdot \#cs - cs_{i,j} &= 0 && (11)\\
n_{i,j} &\geq 0 && (12)\\
cs_{i,j} &\geq 0 && (13)\\
cs_{i,j} - \#cs + 1 &\leq 0 && (14)\\
a - cs_{i,j} - ltz_{a,i,j} \cdot \mathcal{C} &\leq 0 && (15)\\
a - cs_{i,j} + (1 - ltz_{a,i,j}) \cdot \mathcal{C} &\geq 0 && (16)\\
a - cs_{i,j} &\leq y_{a,i,j} && (17)\\
-(a - cs_{i,j}) &\leq y_{a,i,j} && (18)\\
a - cs_{i,j} + (1 - ltz_{a,i,j}) \cdot \mathcal{C} &\geq y_{a,i,j} && (19)\\
-(a - cs_{i,j}) + ltz_{a,i,j} \cdot \mathcal{C} &\geq y_{a,i,j} && (20)
\end{aligned}
$$

Finally, we assign $cs_{a,i,j}$ using the following constraints:

$$y_{a,i,j} - \mathcal{C} \cdot (1 - cs_{a,i,j}) \leq 0 \tag{21}$$

$$y_{a,i,j} - (1 - cs_{a,i,j}) \geq 0 \tag{22}$$

We compute the number of potential cache misses p_a resulting from accessing cache set a using the following constraints for all cache sets a; b_a being auxiliary binary variables, and k denoting the associativity of the cache:

$$\left(\sum_{i,j} cs_{a,i,j}\right) - k - b_a \cdot \mathcal{C} \leq 0 \tag{23}$$

$$\left(\sum_{i,j} cs_{a,i,j}\right) - k + (1 - b_a) \cdot \mathcal{C} \geq 0 \tag{24}$$

$$0 \leq p_a \tag{25}$$

$$\left(\sum_{i,j} cs_{a,i,j}\right) - (1 - b_a) \cdot \mathcal{C} \leq p_a \tag{26}$$

The objective function itself is replaced by

$$\min \mathcal{C} \cdot \sum_{0 \leq a < k} p_a + \bar{a} \tag{27}$$

with the intention of minimizing the number of conflict misses that may occur during execution of the considered basic block. However, if the number of cache sets increases, the ILP to compute the block optimal mapping becomes intractable independently of the number of heap objects. Experiments suggest that for target hardware with 256 or more cache sets solutions to the ILPs cannot be computed in reasonable time anymore. Computing block optimal mappings using ILP formulations also introduces severe complexity issues for programs with a large number of heap allocated memory blocks. In order to enable the analysis of larger programs and/or programs for hardware with more cache sets, we propose to replace the block optimal ILP by a heuristic algorithm. Consider a simulated annealing [5] algorithm as sketched in Algorithm 2. The neighborhood $N(m)$ of a memory mapping m can be defined as the set of all memory mappings, s.t. the address of exactly 1 structure s differs in its address in m by 1. Formally speaking, for a memory mapping $m = \bigcup_{o_{i,j} \in \mathcal{O}} \{o_{i,j} \mapsto a_{i,j}\}$, the neighborhood is defined as $N(m) = \big\{m \oplus \{o_{i,j} \mapsto a'_{i,j}\} \big| |a_{i,j} - a'_{i,j}| = 1\big\}$. The evaluation function to determine the costs of a memory mapping m is defined as

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

The first component ensures that memory mappings without memory conflicts are preferred. The number of potential conflict misses is minimized by the second component. For mappings with the same number of conflict misses, component three favors the ones with minimal memory consumption. 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 $1.2^{-|\mathcal{I}| \cdot restarts}$.

Although, a memory optimal mapping can be computed by our ILP formulation for most programs, this procedure can be easily replaced by a similar simulated annealing heuristic.

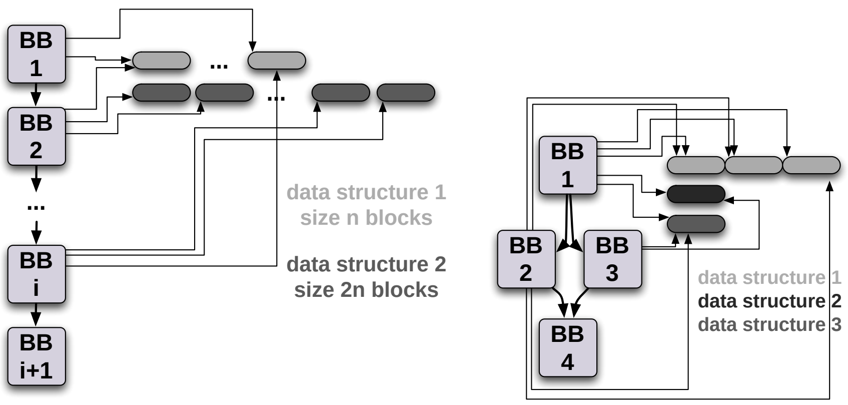
```
Algorithm 2: Simulated annealing algorithm to compute a suitable block optimal mapping
Result: memory mapping with minimal potential conflict misses for program block $b$
$restarts \leftarrow 0$;
$mapping_{best} \leftarrow mem\_opt()$;
while $restarts < RESTARTS$ do
    $mapping_{curr} \leftarrow mem\_opt()$;
    $Temperature \leftarrow Temperature_{MAX}$;
    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 $eval(mapping_{tmp}) < eval(mapping_{best})$ then
                $mapping_{best} \leftarrow mapping_{tmp}$;
            end
            set $mapping_{curr}$ to $mapping_{tmp}$ with probability $e^{\frac{eval(mapping_{curr})-eval(mapping_{tmp})}{Temperature}}$;
        end
        $Temperature \leftarrow Temperature \cdot Cooling\_Ratio$;
    end
    $restarts \leftarrow restarts + 1$;
end
```

4. Experiments

Our preliminary experiments consider as target platforms the PowerPC MPC603e and a simpler toy architecture (DTA): a 2-way set-associative cache with 2 cache sets. The PowerPC CPU utilizes separated 4-way set-associative instruction and data caches of 16 KB, organized in 128 cache sets with a cache line size of 32 bytes.

Unfortunately, we lack a collection of real-life programs on which we could evaluate our algorithm. We therefore consider a typical textbook example used to motivate the use of dynamic memory allocation. When transforming one data structure into another, dynamic memory allocation can 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. Suppose you have a data structure A consisting of n objects and each object occupies l memory blocks. How much memory is needed to transform this structure into a structure B with n objects each occupying k memory blocks? Assuming that $k \geq l$, a moments thought reveals that we need $n \cdot k + l$ memory blocks for this transformation. We tried our algorithm on several program descriptions, where data structures consisting of n heap objects are transformed into other structures consisting of an equal number of objects of larger size (Structure-copy(n)). In all instances, the computed solution used $n \cdot k + l$ memory blocks. In Structure-copy(3)ex we added an additional program block to Structure-copy(3) in which the first and the last block of the first structure as well as the last block of the second structure are accessed. On the simplified hardware the memory optimal mapping for this program is not WCET-optimal anymore. The new WCET-optimal memory uses one additional memory address to resolve the potential conflict misses in the additional program block. Our algorithm successfully discovered that.

As a more complex example, consider a program as sketched in Figure 2 (b). Here, allocating all 5 blocks sequentially is memory and WCET-optimal for the PPC hardware. However, the best memory mapping for our toy hardware would be to allocate the memory block for data structure 3 before that of structure 2. Again, our algorithm discovers that.



(a) Structure-copy(n) (b) If-branch

Figure 2. (Basic-)Block Graph with data accesses for some example programs.

Program	(Target) Hardware	Memory-/Block-Optimal Algorithm	Analysis Time (min/max/**avg**) ms
Structure-copy(3)	DTA	ILP/ILP	23/64/**38.5**
Structure-copy(3)	DTA	ILP/Simulated Annealing	22/82/**49.6**
Structure-copy(3)	PPC603e	ILP/ILP	23/79/**43.2**
Structure-copy(3)	PPC603e	ILP/Simulated Annealing	23/89/**43.6**
Structure-copy(3)ex	DTA	ILP/ILP	73/164/**100.1**
Structure-copy(3)ex	DTA	ILP/Simulated Annealing	29/52/**35.7**
Structure-copy(3)ex	PPC603e	ILP/ILP	23/32/**24.9**
Structure-copy(3)ex	PPC603e	ILP/Simulated Annealing	23/30/**24.8**
If-branch	DTA	ILP/ILP	43,332/44,005/**43,755**
If-branch	DTA	ILP/Simulated Annealing	26/35/**28.5**
If-branch	PPC603e	ILP/ILP	17/41/**20.7**
If-branch	PPC603e	ILP/Simulated Annealing	17/25/**18.4**
Structure-copy(100)	DTA	Simulated Annealing/Simulated Annealing	9,118/9,710/**9,362.3**
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**

Figure 3. Running times of example analyses each executed 10 times on a 2.66 GHz Core 2 Duo with 2 GB RAM.

5. Improvements and Future Work

Our algorithms rely on solving ILPs whose complexity might significantly increase with the number of data structures or more precisely with the number of $a_{i,j}$ variables. However, many programs allocate objects on the heap, although these objects do not survive the function frame they are allocated in. Hence, such heap objects might be allocated on the stack, in which case they would not contribute to the complexity of our algorithm. Escape analysis is a static analysis that determines which heap allocated objects' lifetimes are not bounded by the stack frame they are allocated in [1, 3]. By allocating objects on the stack that do not escape the procedure they are created in, 13% to 95% of a programs heap data can be moved onto the stack [1]. Incorporating an escape analysis and a program transformation moving heap objects onto the stack might therefore significantly reduce the complexity of our algorithm for real-life programs. Therefore, we are currently developing a novel, precise escape analysis based on shape analysis via 3-valued logic [9].

6. Conclusions

Transforming dynamic allocation into static allocation circumvents the main problems of WCET analyses introduced by the unpredictability of memory allocators. Preliminary benchmarks and experiments suggest that such a transformation can be computed with reasonable efforts. However, whether it is feasible to fully automate the process of transforming dynamic allocation to static allocation, i.e., also automatically generate program descriptions, and at what computational costs still remains to be seen.

References

[1] BLANCHET, B. Escape Analysis for Object-Oriented Languages: Application to Java, *SIGPLAN Not. Vol. 10*, 1999.

[2] FERDINAND, C. and WILHELM, R. Efficient and Precise Cache Behavior Prediction for Real-Time Systems, *Real-Time Systems*, 17(2-3):131–181, 1999.

[3] GAY, D. and STEENSGAARD, B. Fast Escape Analysis and Stack Allocation for Object-Based Programs, *CC'00*, 82–93, 2000.

[4] HERTER, J., REINEKE, J., and WILHELM, R. CAMA: Cache-Aware Memory Allocation for WCET Analysis, *WiP Session of ECRTS'08*, 24–27, July 2008.

[5] KIRKPATRICK, S., GELATT, C.D., and VECCHI, M.P. Optimization by Simulated Annealing, *Science, Number 4598*, 671–680, 1983.

[6] LANGENBACH, M., THESING, S., and HECKMANN, R. Pipeline Modeling for Timing Analysis, *Proceedings of the Static Analyses Symposium (SAS)*, volume 2477, 2002.

[7] LEA, D. A Memory Allocator, *Unix/Mail, 6/96*, 1996.

[8] LI, Y.S. and MALIK, S. Performance Analysis of Embedded Software Using Implicit Path Enumeration, *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 456–461, 1995.

[9] SAGIV, M., REPS, T., and WILHELM, R. Parametric Shape Analysis via 3-valued Logic, *ACM Transactions on Programming Languages and Systems, Vol. 24, No. 3*, Pages 217–298, May 2002.

[10] SCHOEBERL, M. Time-predictable Cache Organization, *STFSSD'09*, March 2009.