WCET-aware Static Locking of Instruction Caches[*]

Sascha Plazar
Computer Science 12
TU Dortmund University
D - 44221 Dortmund, Germany
sascha.plazar@tu-dortmund.de

Jan C. Kleinsorge
Computer Science 12
TU Dortmund University
D - 44221 Dortmund, Germany
jan.kleinsorge@tu-dortmund.de

Peter Marwedel
Computer Science 12
TU Dortmund University
D - 44221 Dortmund, Germany
peter.marwedel@tu-dortmund.de

Heiko Falk
Institute of Embedded Systems /Real-Time Systems
Ulm University
D - 89081 Ulm, Germany
heiko.falk@uni-ulm.de

ABSTRACT

In the past decades, embedded system designers moved from simple, predictable system designs towards complex systems equipped with caches. This step was necessary in order to bridge the increasingly growing gap between processor and memory system performance. Static analysis techniques had to be developed to allow the estimation of the cache behavior and an upper bound of the execution time of a program. This bound is called worst-case execution time (*WCET*). Its knowledge is crucial to verify whether hard real-time systems satisfy their timing constraints, and the WCET is a key parameter for the design of embedded systems.

In this paper, we propose a WCET-aware optimization technique for static I-cache locking which improves a program's performance and predictability. To select the memory blocks to lock into the cache and avoid time consuming repetitive WCET analyses, we developed a new algorithm employing integer-linear programming (*ILP*). The ILP models the worst-case execution path (*WCEP*) of a program and takes the influence of locked cache contents into account. By modeling the effect of locked memory blocks on the runtime of basic blocks, the overall WCET of a program can be minimized. We show that our optimization is able to reduce the estimated WCET (abbr. $WCET_{est}$) of real-life benchmarks by up to 40.8%. At the same time, our proposed approach is able to outperform a regular cache by up to 23.8% in terms of WCET_{est}.

[*]Part of the work on this paper has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A3. The research leading to these results was also partially funded by the European IST FP7 NoE ArtistDesign. And work on this publication has partially been supported by Deutsche Forschungsgemeinschaft (DFG) under grant FA 1017/1-1.

1. INTRODUCTION

Caches have become popular in the domain of embedded systems to bridge the growing gap between high processor and low memory performance. They are developed to work transparently from the programmer's point of view by integrating a fully autonomous hardware controller. Recently used memory blocks are kept as copies in fast cache memories since they are likely to be accessed in the near feature. If a block can be fetched from the cache, a so-called cache hit occurs. Otherwise, a cache miss occurs and the requested content has to be fetched from the slow main memory resulting in penalty cycles due to pipeline stalls.

Although caches effectively improve the average-case performance, they are a source of predictability problems due to their dynamic behavior. Static analysis techniques have been developed to compute conservative bounds on the cache behavior and its influence on the runtime of a program.

The worst-case execution time (*WCET*) of a program is the upper bound of the execution time for all possible input data and all possible initial system states. The WCET is a key parameter for real-time scheduling and the development of hardware platforms which have to satisfy critical timing constraints. Since the real WCET of a system cannot be determined, static timing analyzers are employed to determine WCET estimations ($WCET_{est}$).

The WCET of a program corresponds to the length of the worst-case execution path (*WCEP*) which is that path of the control flow graph (*CFG*) with the highest execution time. Optimization of elements like functions on the WCEP can shorten this longest path in such a way that another path becomes the new WCEP. Optimization of elements not lying on the WCEP will not result in a reduction of the WCET.

Hence, possible switches of the WCEP have to be taken into account during optimizations.

In this paper, we present a novel WCET-aware optimization for static locking of instruction caches. The optimization aims at reducing the WCET of a program by statically locking parts of a program into the cache. Statically means that the cache content is loaded and locked in advance and does not change during the program's execution. An integer-linear programming (*ILP*) approach is employed to select the memory blocks to lock. The ILP explicitly models the CFG of a program in order to cope with switching WCEPs. The impact of locked cache contents on the execution time of the contained basic blocks is modeled as well and thereby avoids repetitive WCET analyses. To enable such a WCET-centric optimization, the sophisticated static WCET analyzer *aiT* developed by AbsInt [1] is employed. The main contributions of this paper are as follows:

- Cache content is statically locked based on its impact on the WCET_{est} of a program.
- The presented ILP-based approach determines a set of memory blocks to lock into the cache. Its objective is the reduction of the WCET_{est} by explicitly modeling the impact of locked cache contents on the WCEP of a program.
- We show that WCET_{est} reductions of up to 40.8% can be achieved compared to a system without cache for a set of real-life programs.
- The practical relevance of our new optimization is attested by outperforming a regular cache by up to 23.8% w. r. t. WCET_{est} reductions.

This paper is organized as follows: In the next section, an overview of related work considering memory and cache-based optimizations is provided. Section 3 presents our new WCET-aware algorithm for static I-cache locking. Section 4 introduces the WCET-aware C compiler *WCC* employed to develop our novel algorithm. An evaluation in Section 5 compares the performance of a system with a regular cache with our novel WCET-aware static I-cache locking technique. Finally, Section 6 concludes our work and presents ideas for future work.

2. RELATED WORK

Falk et al. counteract possible predictability problems of caches with a static allocation of program code to so-called scratchpad memories (*SPM*) [5]. They employ integer-linear programming to select the optimal content of the SPM w. r. t. the program's WCET_{est}. The disadvantage of moving parts of the code to SPMs is the necessary correction of the control flow: Far jumps have to be inserted to branch between different memories leading to an unavoidable code and runtime overhead. In contrast, the cache locking based optimization presented in this work exploits the transparent behavior of a cache and performs a lockdown of cache content inside the startup code of a program.

Another work considering scratchpad allocation is presented in [18]. Suhendra et al. developed an ILP-based allocation of frequently accessed data objects to faster memories in order to decrease the overall WCET_{est}. Their model of the program's WCET and possible execution paths serves as basis for the ILP-based algorithm presented in [5] and was also employed for the technique discussed in Section 3. Since only the intra-function control flow is modeled, a time consuming branch-and-bound approach or a sub-optimal heuristic is employed to optimize along the WCEP. Furthermore, moving data objects to SPM is much more easier than locking instruction blocks into the I-cache. Data elements can be almost arbitrarily moved around to fill the SPM without gaps. Thereby, elements are never competing for the same memory/cache lines as occurring during cache locking.

In [8], Gebhard et al. present a technique for rearranging the positions of tasks to improve the cache performance. The interdependency relation of tasks is evaluated in order to determine a memory layout which maximizes the number of persistent cache sets for each task.

Papers [16] and [3] present techniques for statically locked instruction caches which are very close to the work presented in this paper. In [16], Puaut et al. present two algorithms which try to minimize the CPU utilization and the interferences between different tasks, respectively. Although they consider the WCET as metric, they are not able to react on switching WCEPs since they always optimize along an initially determined WCEP. Compared to [16], [3] presents an additional genetic algorithm which has the disadvantage that for each created individual, a time consuming WCET analysis has to be performed.

Puaut et al. also present techniques for I-cache locking [15] which consider changing WCEPs. However, the way how WCEPs are recomputed is not detailed. The authors use a parameter N trading off accuracy of WCEP recomputation with runtime consumption. Since runtimes for WCEP recomputation are still very high, the authors are unable to provide results for some of their benchmarks. In contrast, the techniques presented in this work scale much better so that results for very large benchmarks can be gathered.

Static I-cache locking is also proposed by Falk et al. in [7]. An *Execution Flow Graph* (EFG) is employed to model possible execution paths on function level. The WCEP is determined by applying a modified Dijkstra algorithm before the most promising function on the WCEP is locked into the cache. To consider paths which are not the initial WCEP, two analyses are required for each alternative path in order to compute the gain of the functions on such a path. As opposed to this, the approach presented in this paper only requires two WCET analyses in total.

An extension of [7] was developed by Liu et al. in [12]. There, an *Execution Flow Tree* is presented which is traversed to generate a simple ILP which selects the functions to lock into the cache. In contrast to [7] and [12], the approach proposed in this paper is able to model the intra-function WCEP including loops at basic block level. The influence of the memory layout on lockable memory blocks – and thereby the runtime of basic blocks – is also taken into account; [7] and [12] ignore the fact that selected func-

```
void foo1(int x) {
  if(x<100)
    foo2();
  else
    foo3();
}
```

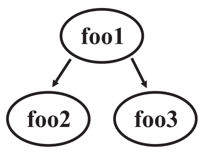


Figure 1: Exemplary program and resulting call graph

tions may conflict in the cache and thus cannot be locked simultaneously.

3. WCET-AWARE STATIC CACHE LOCKING

Embedded systems are equipped with caches in kilobyte ranges, typically from 1 kB up to 16 or 32 kB. Compared to growing main memories in megabyte ranges, caches are rather small. The I-cache controller tries to keep copies of frequently executed memory lines containing sequences of instructions as cache content for a faster access.

Since the cache can only keep a fraction of the information residing in slower memories, cache misses can occur if a memory address to be accessed is not already stored in the cache (called *real cache misses*). If, however, a memory address is already stored in the cache, a cache hit occurs and the content can usually be fetched within one cycle.

The amount of cache misses highly depends on the ratio of cache to memory size, the cache replacement policy and the structure of the executed program. A high amount of cache misses implies costly reloading of content from the slow main memory and leads to a high number of penalty cycles due to pipeline stalls.

Besides unavoidable real cache misses, the computed WCET of a program is affected by the overestimation of a static WCET analyzer as well: If the memory address of an instruction fetch cannot be determined, it also cannot be determined if a memory access results in a cache hit or a cache miss. In such a case, the worst case – usually a cache miss – has to be assumed (called *assumed cache misses*). Figure 1 shows a code snippet and the resulting call graph for which such a situation can occur. If the value analysis of a static timing analyzer cannot determine whether `foo`'s parameter `x` is less than 100, a cache analysis has to consider both the `if`- and the `else`-path. For a memory layout as depicted in Figure 2, `foo1` and `foo2` are mapped to the same cache area as `foo3`. Now, the worst case has to be assumed where for each execution of `foo1`, `x` toggles between a value less than 100 and equal or above. Thus, for each execution of `foo1`, it has to be assumed that either `foo1` and `foo2` evict `foo3` from the cache or vice versa. This leads to an unnecessary high number of assumed cache misses if, for instance, `x` is usually below 100.

3.1 Cache Locking

To overcome the problems of real and assumed cache misses, several processor architectures – especially in the embedded domain – are equipped with techniques to lock the content

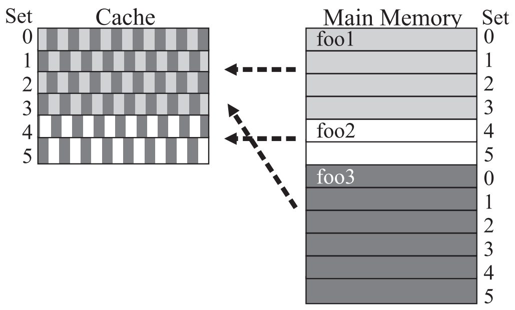


Figure 2: Worst-case cache behavior

of caches. The embedded processor ARM926EJ-S [2], which is considered in this paper, uses a cache-way-based locking scheme where dedicated locking bits steer if the normal cache allocation is allowed to access the corresponding cache way. By locking cache ways, it can be ensured that an access to locked content always results in a cache hit and thus the number of real cache misses can be reduced. Overestimation of the WCET (caused by assumed cache misses) can be reduced as well since a static timing analyzer can doubtlessly determine which memory accesses result in cache hits and which ones result in cache misses.

For an n-way set-associative cache with a size of S_{cache} bytes and a line size of S_{line} bytes, each way consists of l cache lines:

$$l = \frac{S_{cache}}{n * S_{line}}$$

Each way comprises $S_{way} = S_{cache}/n$ bytes. Due to the modulo addressing function of cache controllers, memory addresses $m \bmod S_{way} = 0$ are mapped to the beginning of a cache way. Thus, the main memory can be divided into memory blocks with a size of S_{way} bytes for which each block can be entirely locked into a single cache way.

For a way-based lockdown of cache content as supported by the ARM926EJ-S, only memory blocks with a granularity based on such a partitioning can be locked into the cache. Loading content from the main memory and locking it into a single cache line causes some costs C_{line}. Thus, the costs for locking a complete way are as follows:

$$c_{lock} = l * C_{line} \tag{1}$$

In order to support an automatic lockdown of instruction caches to reduce the WCET, possible WCEP switches have to be recognized and handled which makes optimization challenging. The following sections present our ILP-based optimization technique which is capable of modeling a program's control flow and thereby ensures that optimizations are always performed along the WCEP. It determines an optimal set of memory blocks to lock into the cache w. r. t. the WCET of a program. Our algorithm requires only *two* WCET analyses and is able to consider the influence of locked cache lines on the execution time of a basic block. The next Section 3.2 models the costs for the locking of cache lines. Section 3.3 introduces basic block costs representing their execution times, whereas 3.4 describes the modeling of a function's control flow in the ILP. Afterwards, Section 3.5

models the global control flow whereas Section 3.6 describes the ILP's objective function.

3.2 Lockdown Constraints

In the following, ILP variables are represented using lowercase letters whereas constants are represented by uppercase letters. For ILP-based cache locking, the code of a program to be optimized has to be considered as m memory blocks with a size equal to the cache's way size S_{way}. The blocks start at memory addresses which are mapped exactly fitting into cache ways (cf. Section 3.1).

For each of these blocks, a binary decision variable x_i decides whether memory block mb_i is locked into an arbitrary cache way:

$$x_i = \begin{cases} 1, & \text{if memory block } mb_i \text{ is locked} \\ & \text{into the cache} \\ 0, & \text{else} \end{cases} \quad (2)$$

An n-way set-associative cache can keep copies of up to n such memory blocks at the same time since ways can only be locked entirely. Thus, a constraint has to be formulated to ensure that the size of the content to lock does not exceed the cache size:

$$\sum_{i=1}^{m} x_i <= n \quad (3)$$

Since the cache has to be filled with code and locked in advance, overhead in form of execution cycles arise before the program's execution. For a cache with ways composed of l cache lines, Equation (1) is extended to model the overall lockdown overhead:

$$o_{lock} = \sum_{i=1}^{m} x_i * l * C_{line} \quad (4)$$

3.3 Basic Block Costs

The execution time for a single execution of a basic block b_j is represented by C_j^{main} if b_j is entirely executed from main memory whereas C_j^{cache} is the execution time if the block is locked into the cache. s_j is the size of basic block b_j in bytes and s_j^i is the amount of bytes from basic block b_j overlapping with memory block mb_i. Then, d_j^i is the runtime reduction in cycles if parts of basic block b_j located in memory block mb_i are fetched from the cache due to a lockdown of mb_i:

$$d_j^i = \frac{s_j^i}{s_j} * (C_j^{main} - C_j^{cache}) \quad (5)$$

Each basic block b_j of a function F causes some costs c_j. These costs represent the WCET of b_j depending on the memory from which b_j's instructions are fetched. If b_j or parts of it are locked into the cache, the execution time decreases by d_j^i cycles:

$$c_j = C_j^{main} - \sum_{i=1}^{m} x_i * d_j^i \quad (6)$$

3.4 ILP Model of the Control Flow of Functions

For reducible CFGs (refer to [13], p. 196-197, for definition), an innermost loop L of F has exactly one basic block b_{entry}^L being the loops' unique entry point, and possibly several back-edges turning it into a cyclic graph. Not considering these back-edges turns L's CFG into an acyclic graph. $G_L = (V, E)$ denotes this acyclic graph in the following. Without loss of generality, it can be assumed that there is at least one basic block b_{exit}^L in G_L being the loop's exit node. The WCET w_{exit}^L of block b_{exit}^L is equal to its costs:

$$w_{exit}^L = c_{exit}^L \quad (7)$$

The WCET of a path leading from a node $b_j \neq b_{exit}^L$ of G_L to one of the exit nodes b_{exit}^L must be greater than or equal to the WCET of any successor b_{succ} of b_j in G_L, plus the cost c_i of b_j:

$$\begin{aligned} \forall b_j \in V \setminus \{b_{exit}^L\} : \ \forall (b_j, b_{succ}) \in E : \\ w_j \geq w_{succ} + c_j \end{aligned} \quad (8)$$

A path constructing example is illustrated in Appendix A.1.

Since paths are built bottom-up, variable w_{entry}^L models an upper bound of the WCET of all paths of a loop L if it is executed exactly once. In order to model multiple executions of L, all CFG nodes $v \in V$ of G_L are represented by a super-node v_L. The costs of v_L are the product of L's WCET for a single execution and L's maximal loop iteration count (cf. Appendix A.2):

$$c_L = w_{entry}^L * Count_{max}^L \quad (9)$$

Replacing a loop L by a super-node v_L in the CFG may turn another loop L' of F directly surrounding L into an innermost loop with acyclic CFG G'_L. Hence, the constraints of Equations (8) and (9) can be formulated for L'. This way, the innermost loops of F are successively collapsed in the CFG so that ILP constraints modeling F's control flow are created from the innermost to the outermost loops.

During optimization, a WCEP switch of a program can only happen at such points in the CFG where a basic block b_j has more than one successor. Only there, forks in the control flow are possible where the outgoing paths can have different WCETs. But since Equation (8) is formulated for each successor of b_j, variable w_j always reflects the WCET of any path starting at b_j – irrespective of the fact which successors are actually part of the current WCEP. This way, the constraint of Equation (8) realizes the implicit consideration of WCEPs and their changes in the ILP.

The fundamental structure of the ILP constraints of Equations (7) – (9) stems from the work of Suhendra et al. proposed in [18]. In order to implement a fully functional cache locking technique, these basic constraints had to be refined substantially. Our extensions of the original ILP formulation are Equations (2) – (6) and the equations described in the following sections.

3.5 ILP Model of the Global Control Flow

Up to this point, Equations (5) – (9) only model the intraprocedural control flow of a single function within the ILP. In this way, Suhendra's approach is not able to model the global control flow and thus requires repetitive WCET analyses to optimize along a switching WCEP. After each local optimization step, a WCET analysis is performed to update the functions' timing and path data. Such a procedure is time consuming and is not desired in the context of cache locking. Since locking of memory blocks into the cache can have an impact on the run time of several functions at the same time, a local optimum could be the result otherwise.

To overcome these difficulties, the global control flow of a program is modeled within the ILP: without loss of generality, we assume one dedicated entry block b^F_{entry} as first block of function F. For b^F_{entry}, the ILP variable w^F_{entry} denotes the WCET of any path starting at b^F_{entry} for a single execution of F.

However, some basic block b_j of a function G may contain a call to function F. In this situation, F's WCET represented by variable w^F_{entry} has to be added to the WCET of block b_j. Thus, the control flow constraint in Equation (8) is extended by w^F_{entry}, representing F's WCET, if block b_j calls F:

$$\forall b_j \in V \setminus \{b^L_{exit}\} : \; \forall (b_j, b_{succ}) \in E : \quad w_j \geq w_{succ} + c_j + w^F_{entry} \tag{10}$$

The interested reader is referred to Appendix A.3 for an illustrating example.

3.6 Objective Function

The overall goal of the ILP is to minimize a system's WCET by locking memory blocks into cache ways. Due to the nature of Equations (8) and (10), variable w^F_{entry} corresponds to the WCET of function F including the WCETs of all functions called by F. Function `main` is the unique entry point of an entire program; hence, variable $w^{\texttt{main}}_{entry}$ denotes the WCET of the program. Since the lockdown of cache content has to be done in advance, the overhead from Equation (4) have to be added to the overall WCET of a system:

$$w_{system} = w^{\texttt{main}}_{entry} + o_{lock} \tag{11}$$

Finally, the value of this variable has to be minimized by the ILP:

$$w_{system} \rightsquigarrow min. \tag{12}$$

4. WORKFLOW

WCET-driven optimizations such as our novel WCET-driven cache locking require support of an underlying compiler to collect WCET data and to add the required locking operations to a program's startup code. We employ the WCET-aware C compiler framework (*WCC*), developed by Falk et al. [6], which is intended to assist the development of specialized high- and low-level WCET-driven optimizations. It is a compiler targeted at Infineon's TriCore TC1796 processor coupled with AbsInt's static WCET analyzer *aiT* [1] which provides WCET$_{est}$ data that is imported into the compiler backend and made accessible for optimizations. We extended this framework to support code generation and optimization for the ARM platform.

Figure 3 depicts *WCC*'s internal structure. The solid arrows show the flow of information through a normal optimizing compiler, whereas the dotted arrows show the extensions necessary for tailored WCET-directed optimizations. One or more files of a program are read in the form of ANSI-C source files with user annotations for loop bounds and recursion depths, called *flow facts*. These source files are parsed and

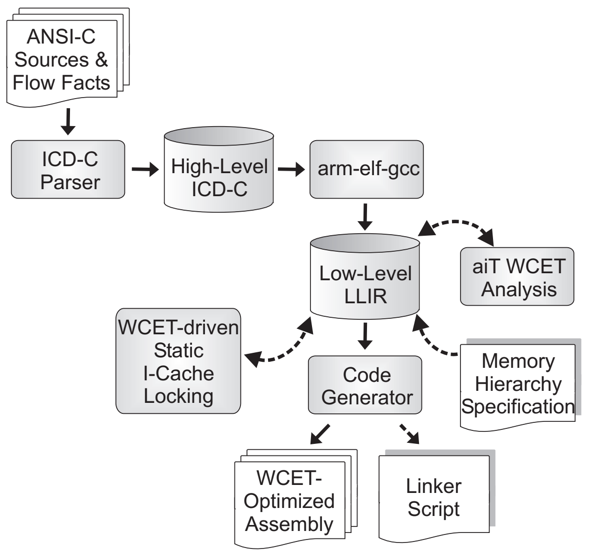


Figure 3: Workflow of the WCET-aware C compiler *WCC*

transformed into the high-level intermediate representation (*IR*) called *ICD-C* [17]. At this level, the compiler frontend provides several standard compiler optimizations focusing on ACET minimization.

In the next step, the `arm-elf-gcc` translates the high-level IR into assembly code which is read in to generate a low-level IR called *ICD-LLIR* [4]. Again, several standard compiler optimizations can be performed – now on this ARM-specific low-level IR. One of these optimizations is the proposed WCET-driven static cache locking technique.

To enable such a WCET-aware optimization, *aiT* is employed to perform static WCET analyses on the low-level IR. Mandatory information about loop bounds (among others required as constant $Count^L_{max}$ in Equation (9)) and recursion depths is supplied by flow fact annotations. These flow facts are automatically translated from the high-level IR to the low-level IR by exploiting DWARF debug information as proposed by Plazar et al. in [14]. They are always kept valid and consistent during each optimization and transformation step of the compiler.

Finally, *WCC* emits WCET-optimized assembly files and its own linker script in order to generate the optimized binary.

5. EVALUATION

In order to demonstrate the effectiveness of our WCET-aware cache locking technique, the approach presented in this paper is applied to a set of real-life benchmarks. In Section 5.1, the experimental environment is described which is employed to perform evaluations. Section 5.2 discusses the WCET$_{est}$ reductions achieved by our cache locking described in Section 3, whereas Section 5.3 discusses the required optimization runtime. Finally, Section 5.4 draws a comparison between our new optimization with existing WCET-aware cache locking techniques.

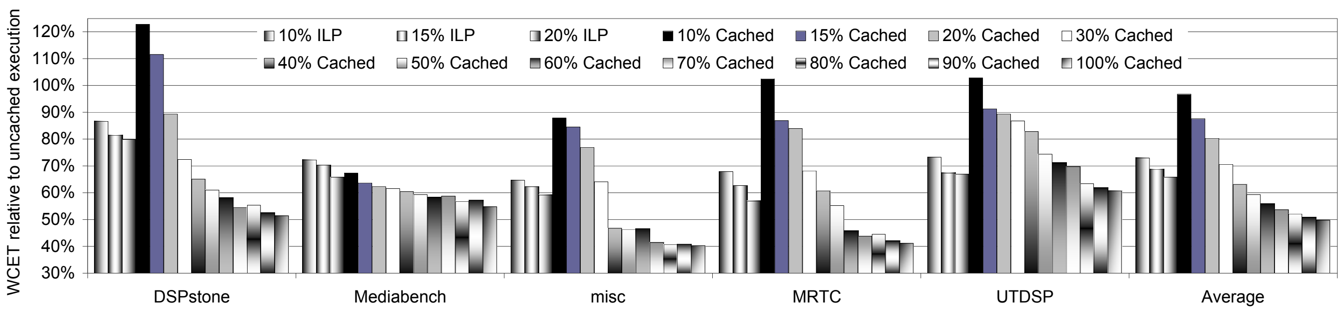


Figure 4: Relative WCET$_{est}$s for a 2-way set-associative cache

5.1 Experimental Environment

For benchmarking, we employed the ARM926EJ-S processor which is equipped with 1 MB ROM as main memory from which content can be fetched within 6 cycles. The processor integrates a 16 kB I-cache with 32 bytes line size, least recently used (LRU) replacement strategy and a configurable associativity of 2 or 4. Content which is available in the cache can be accessed within 1 cycle. The cache supports way-based cache locking for which Equations (1) – (12) are tailored. As stated in [7], loading and locking a single cache line of 32 bytes requires 47 cycles. These 47 cycles are used as constant C_{line} in Equations (1) and (4).

Uniformly, the optimization level *O3* is used for which the *WCC* compiler (cf. Figure 3) applies 42 different optimizations in order to evaluate the performance of our new algorithms on highly optimized code. 23 of these optimizations, comprising various loop optimizations, are performed on the high-level IR ICD-C. The remaining optimizations are performed on the low-level IR ICD-LLIR.

For our evaluations, we used 100 benchmarks stemming from the benchmark suites *DSPStone* [20], *MediaBench* [11], *MRTC* [9] and *UTDSP* [19]. Additionally, we added a set of miscellaneous benchmarks referred to as *misc*. The code size of the benchmarks ranges from 100 bytes (*matrix_1x3*) up to 20 kB for the *gsm* benchmark.

Today's embedded systems are equipped with main memories in megabyte ranges. Compared to this, their caches are rather small with capacities between 2 kB and 32 kB. Due to this ratio of small cache compared to a large main memory, we artificially limited the cache sizes to 10, 15 and 20% of the program's overall code size. This guarantees that we use a similar ratio of cache size to program size for all optimizations and static WCET analyses as found in current embedded systems, in order to generate comparable results. Otherwise, it would make more sense to use a small scratchpad memory – which is fully precitable – instead of a cache to store the program for a fast execution. But for the sake of completeness, cache sizes of up to 100% of the program size are considered for a regular cache in order to explore the best possible reductions of the WCET$_{est}$.

For solving the ILP model generated by the algorithm in Section 3.1, *IBM ILOG CPLEX* [10] is utilized which is a sophisticated solver for integer programming problems.

5.2 WCET Estimations

Figure 4 depicts the results achieved by our static cache locking algorithm if applied to the considered 100 benchmarks. The 100% line is equal to the estimated WCET of the benchmarks compiled with the optimization level *O3* executed in a system without any cache. For each benchmark, the left three bars represent the results achieved by our static cache locking technique if the cache amounts to 10, 15 and 20% of the overall program size. These resulting WCET$_{est}$s already include the overhead for loading and locking parts of a program's code into the cache before its execution. In order to assess the efficacy of static cache locking compared to a regular cache, the right bars represent the results if the benchmarks are executed on a system with a regular cache. All bars depict the average WCET$_{est}$ of the optimized programs of each benchmark suite for a system equipped with a 2-way set-associative cache computed by the static WCET analyzer as percentage of its "uncached" version.

By locking content into the I-cache, our ILP-based optimization is able to reduce the WCET$_{est}$ of the programs by up to 35.4% for 10% cache for the *misc* benchmarks. For the same benchmark set, the WCET$_{est}$ is reduced by up to 37.7 and 40.8% for 15 and 20% cache size, respectively. A system with a regular cache is able to achieve WCET$_{est}$ reductions of up to 32.7% for the *MediaBench* suite and 10% cache size. If the cache amounts to 15 and 20% of the overall program size, the WCET$_{est}$ is reduced by up to 36.2 and 37.8%, respectively.

On average over all considered 100 benchmarks, WCET$_{est}$ reductions of 27.1, 31.2 and 34.3% can be achieved for 10, 15 and 20% cache size if our novel ILP-based cache locking technique is applied. For a regular cache, however, only average WCET$_{est}$ reductions of 3.3, 12.3 and 19.5% can be registered for 10, 15 and 20% cache size. Here, the ILP-based cache locking optimization outperforms the regular cache by 23.8, 18.9 and 14.8% for 10, 15 and 20% cache size.

If the cache size of a system without cache locking is increased up to 100% of the program size, the WCET$_{est}$ is decreased by up to 59.7% for the *misc* benchmarks. Since the benchmarks which are gaplessly arranged in memory entirely fit into the cache, no cache misses due to evictions can occur. Thus, the achieved results represent the highest possible WCET$_{est}$ reductions for a regular cache as well as for a statically locked cache. On average, WCET$_{est}$ reductions of 50.3% for 100% cache size can be documented.

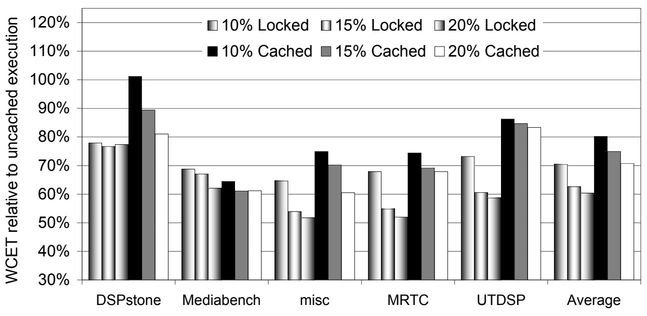


Figure 5: Relative WCET$_{est}$s for a 4-way set-associative cache

Although larger caches can keep copies of more instructions, it can happen that smaller caches achieve lower estimated WCETs due to less cache misses. For instance, a system executing the *MRTC* benchmark suite which is equipped with a cache of 70% size has a lower WCET$_{est}$ than with 80% size. This behavior is caused by the fact that by increasing the cache size, the mapping of memory blocks to cache lines is changed due to the modulo addressing function. Perhaps, other blocks now compete for the same cache lines resulting in a completely different eviction behavior and possibly increased WCET. For the cases where the capacity of a cache with l lines is doubled, consistently lower WCET$_{est}$s can be observed: the set of memory blocks mapped to the same cache line n is bisected into the blocks which are still mapped to the same line as before and the set of blocks which are mapped to cache line $l+n$. This can only lead to a reduction of cache misses which are induced by conflicts.

For caches with larger associativity, a memory block can be mapped to a larger amount of cache ways and thereby, the number of conflicts tends to be decreased. Figure 5 depicts the results if a 4-way set-associative cache is employed. Both the system with a regular cache and a system with cache content locked by our optimization algorithm perform better than for a 2-way set-associative cache.

Our ILP-based cache locking optimization is able to decrease the WCET$_{est}$ compared to a system without cache by up to 35.4% for 10% cache size. For cache sizes of 15 and 20%, WCET$_{est}$ reductions of up to 46.1 and 48.2% can be achieved for the *misc* benchmarks, respectively. Especially for small cache sizes of 10%, the regular cache significantly profits from a larger associativity and achieves WCET$_{est}$ reductions of up to 35.5%. For caches with larger capacities of 15 and 20%, even higher WCET$_{est}$ reductions of up to 38.9 and 38.8% are achieved, respectively.

Even though the regular cache outperforms the statically locked cache for *MediaBench*, the average WCET$_{est}$ reductions are worse: The ILP-based cache locking decreases the WCET$_{est}$ by 29.5, 37.4 and 39.6% for 10, 15 and 20% cache size, whereas the regular cache can only decrease the WCET$_{est}$ by 19.8, 25.0 and 29.2% for the same cache sizes.

5.3 Optimization Time

An Intel Xeon E5506 (2.13 GHz) was utilized to consider the time required for optimization of the benchmarks in Section 5.2. Most of the time necessary for our novel WCET-aware static I-cache locking optimization was consumed by the WCET analyses using *aiT* which is always executed on a single CPU core.

For a single WCET analysis for a system without cache, up to 90 CPU minutes are required for the *latnrm_32_64* benchmark stemming from the *UTDSP* benchmark suite. Thereby, an optimization run spends up to 3 hours for the two required WCET analyses to determine the constants C_j^{main} and C_j^{cache} (cf. Section 3.3). But more than 90% of the considered benchmarks are analyzable within 2 minutes. This is still suitable for most application scenarios since the optimization essentially doubles the compilation time.

The complexity of solving the ILPs generated by the optimization discussed in Section 3.1 is of no practical relevance. For a CFG with n nodes, the ILP has a size of $O(n^2)$ constraints. For a program consisting of m memory blocks of size S_{way}, the ILP contains $O(n^2 + m)$ variables. The employed ILP solver *CPLEX* takes up to 1 CPU minute (*lmsfir_32_64* from *UTDSP*) but mostly terminates within a few seconds for the considered benchmarks. Compared to the WCET analysis required to determine the cost constants C_j^{main} and C_j^{cache} for each basic block, these values are negligible.

5.4 Comparison with existing optimizations

Besides the considerable performance gain compared to a regular cache, the algorithm presented in this paper also outperforms state-of-the-art optimizations for instruction cache locking. Plazar's algorithm [7] as well as the optimization presented by Liu [12] are only able to lock complete functions. Their disadvantage is that if an entire function is locked into the cache, code blocks which are not part of the WCEP are locked into the cache as well. This wastes cache space and thereby optimization potential. Even so, it cannot be determined how many unused blocks are locked into the cache: For example, a block on the WCEP which eminently contributes to the WCET is locked into the cache. If, as a result, a WCEP switch occurs, it is possible that the locked block is no longer part of the new WCEP. Thus, only counting the locked instructions of the optimized program which are not part of the final WCEP cannot answer the question how many unused content is locked. Generally speaking, this observation applies for all cache locking based optimizations – also for the one presented in this paper.

In the following, the algorithm presented in this paper is compared to Falk's *EFG* algorithm which is only marginally outperformed by Liu's optimal approach. It turned out, that function-based locking techniques are not well suited to handle small caches. If, for instance, cache sizes of 10% of the overall program size are considered, there are cases where no promising function fits into the cache. Thus, we extended Falk's approach such that if the most promising function does not fit into the remaining cache, only the beginning of this function is locked into the free cache memory.

Figure 6 shows the results achieved by Falk's *EFG* algorithm (labeled "Reference") compared to our ILP-based optimization if a 2-way set-associative cache is considered. Our new optimization outperforms locking techniques based on func-

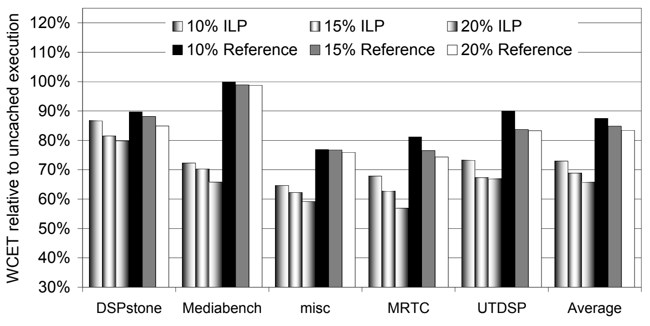


Figure 6: Relative WCET$_{est}$s compared to function-based locking

tions by up to 33% for the *MediaBench* suite. The *EFG* algorithms performs bad in this case, since most benchmarks have only few functions but monolithic computation kernels. The only locked functions do not entirely fit into the cache and their hotspots are often located in the middle or even at the end of the function which did not fit into the cache anymore.

But in most cases, the function-based locking profits from our modification which enables partially locking of promising functions into the remaining cache. Nevertheless, the function-based cache locking technique is outperformed by 14.6, 16 and 17.7% w. r. t. WCET$_{est}$ reductions for 10, 15 and 20% cache size averaged over all considered benchmarks. This underlines the predominance of fine-grained cache locking techniques based on memory blocks instead of on entire functions.

Although we did not perform an evaluation of the optimization runtimes, a qualitative statement can be made: To consider paths which are not the initial WCEP, Falk's EFG requires two analyses for each alternative path in order to compute the gain of the functions on such a path. Since 97 of 100 considered benchmarks have concurrent paths, the amount of required WCET analyses and thereby the optimization runtime is at least doubled for Falk's EFG approach compared to our ILP-based optimization.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown that the WCET$_{est}$ of a program can be effectively reduced by locking parts of its code into the I-cache. The locked content is preserved against eviction leading to an increased number of cache hits and a decreased overestimation of its WCET.

We presented a novel WCET-aware optimization for statically locked instruction caches. The introduced ILP-based approach is tailored to select the content of instruction caches during compile time w. r. t. the WCET$_{est}$ of a program to optimize. The selected content is loaded and locked by the startup code before the program's execution. The overhead for loading and locking code into the cache is explicitly considered in the ILP. As compared to existing approaches, repetitive time-consuming WCET analyses are avoided by modeling the control flow of the program in order to always optimize along a possibly switching WCEP.

We have shown that our algorithm was able to decrease the WCET$_{est}$ of a set of real-life programs by up to 40.8%. On average over all considered benchmarks, WCET$_{est}$ reductions between 27.1 and 39.6% were achieved for cache sizes ranging from 10 – 20%. A regular cache is outperformed by 9.7 – 23.8% and existing function-based cache locking techniques by 14.6 – 17.7% for the same cache sizes, respectively.

In the future, we plan to extend our static cache locking optimization to be able to handle multi-task sets. The cache should be automatically split, and the partitions should be exclusively assigned to individual tasks.

It also seems to be worthwhile to extend the optimization presented in this paper to dynamically locked caches. If the locked content of a cache is exchanged during execution, for instance, at function calls, code which will not be used in the near future could be replaced by other heavily used code blocks.

Furthermore, we intend to combine a code positioning technique on basic block level with the ILP-based optimization presented in this paper. By bundling basic blocks residing on the WCEP, code which does not contribute to the WCET can be moved aside. Thereby, unprofitable code snippets could be excluded from locking into the cache which would otherwise be unavoidable for way-based locking. Possibly, more basic blocks on the WCEP could be locked into the cache leading to a decreased WCET compared to cache locking without code positioning.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the *aiT* framework.

7. REFERENCES

[1] AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzers. 2012. `http://www.absint.com/ait`.

[2] Advanced RISC Machines Ltd (ARM^{TM}). *ARM926EJ-S Technical Reference Manual*, `ARM DDI 0198E` edition, 2001-2008.

[3] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix. Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, Washington, DC, USA, 2005.

[4] J. Eckart and R. Pyka. ICD-LLIR Low-Level Intermediate Representation. `http://www.icd.de/es/icd-llir`, 2012. Informatik Centrum Dortmund.

[5] H. Falk and J. C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of Design Automation Conference (DAC)*, San Francisco, USA, 2009.

[6] H. Falk and P. Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Journal on Real-Time Systems*, 46(2):251–300, 2010.

[7] H. Falk, S. Plazar, and H. Theiling. Compile-Time Decided Instruction Cache Locking using Worst-Case

Execution Paths. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, New York, NY, USA, 2007.

[8] G. Gebhard and S. Altmeyer. Optimal Task Placement to Improve Cache Performance. In *Proceedings of ACM & IEEE International Conference on Embedded Software (EMSOFT)*, New York, USA, 2007.

[9] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proceedings of Workshop on Worst Case Execution Time Analysis (WCET)*, Brussels, Belgium, 2010.

[10] International Business Machines Corporation (IBM). IBM ILOG CPLEX V12.1, 1987, 2009.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, Washington, DC, USA, 1997.

[12] T. Liu, M. Li, and C. J. Xue. Minimizing WCET for Real-Time Embedded Systems via Static Instruction Cache Locking. In *Proceedings of IEEE Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, San Francisco, CA, United States, 2009.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[14] S. Plazar, P. Lokuciejewski, and P. Marwedel. A Retargetable Framework for Multi-objective WCET-aware High-level Compiler Optimizations. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS) WiP*, Barcelona, Spain, 2008.

[15] I. Puaut. WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, Dresden, Germany, July 2006.

[16] I. Puaut and D. Decotigny. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, Austin, TX, USA, 2002.

[17] R. Pyka and J. Eckart. ICD-C Compiler Framework. `http://www.icd.de/es/icd-c`, 2012. Informatik Centrum Dortmund.

[18] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of IEEE International Real-Time Systems Symposium (RTSS)*, 2005.

[19] UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, 2012.

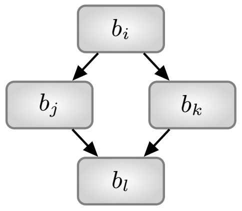
[20] V. Živojnović, J. Martinez, C. Schläger, and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT)*, Dallas, TX, USA, 1994.

APPENDIX

A. ILP CONTROL FLOW MODELING

A.1 Path Modeling

In the following, a small example illustrates the modeling of the WCEP for a sequence of basic blocks:



A path is modeled bottom-up starting at basic block b_l. As per Equation (7), its WCET only depends on the costs c_l which in turn depend on the memory from which b_l is executed:

$$w_l = c_l$$

According to Equation (8), the WCET of block b_j (b_k is modeled analogously) is equal to its own costs plus the WCET of its only successor b_l:

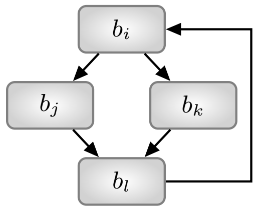
$$w_j \geq c_j + w_l$$
$$w_k \geq c_k + w_l$$

Finally, the WCET of the first basic block b_i which ends with a branch instruction is modeled as follows:

$$w_i \geq c_i + w_j$$
$$w_i \geq c_i + w_k$$

A.2 Loop Representation

If the sequence of basic blocks in example A.1 is turned into a loop named $L1$, the resulting control flow can be as follows:



According to Equation (9), the costs of the supernode v_{L1} representing loop $L1$ amount to the WCET of its entry basic block b_i multiplied by its maximum iteration count:

$$c_L = w_i * Count^L_{max}$$

A.3 Global Control Flow

Assumed that basic block b_j of example A.1 calls a function `foo`, the WCET of `foo`'s entry basic block $b^{\texttt{foo}}_{entry}$ is added to the costs of b_j according to Equation (10). Only the constraint of A.1 modeling the WCET of b_j thus has to be extended:

$$w_j = c_j + w_l + w^{\texttt{foo}}_{entry}$$