Compile-Time Decided Instruction Cache Locking Using Worst-Case Execution Paths*

Heiko Falk
Computer Science 12
University of Dortmund
D-44221 Dortmund
Heiko.Falk@udo.edu

Sascha Plazar
Computer Science 12
University of Dortmund
D-44221 Dortmund
Sascha.Plazar@udo.edu

Henrik Theiling
AbsInt Angewandte Informatik
Science Park 1
D-66123 Saarbrücken
theiling@absint.com

ABSTRACT

Caches are notorious for their unpredictability. It is difficult or even impossible to predict if a memory access results in a definite cache hit or miss. This unpredictability is highly undesired for real-time systems. The Worst-Case Execution Time *(WCET)* of a software running on an embedded processor is one of the most important metrics during real-time system design. The WCET depends to a large extent on the total amount of time spent for memory accesses. In the presence of caches, WCET analysis must always assume a memory access to be a cache miss if it can not be guaranteed that it is a hit. Hence, WCETs for cached systems are imprecise due to the overestimation caused by the caches.

Modern caches can be controlled by software. The software can load parts of its code or of its data into the cache and lock the cache afterwards. Cache locking prevents the cache's contents from being flushed by deactivating the replacement. A locked cache is highly predictable and leads to very precise WCET estimates, because the uncertainty caused by the replacement strategy is eliminated completely.

This paper presents techniques exploring the lockdown of instruction caches at compile-time to minimize WCETs. In contrast to the current state of the art in the area of cache locking, our techniques explicitly take the worst-case execution path into account during each step of the optimization procedure. This way, we can make sure that always those parts of the code are locked in the I-cache that lead to the highest WCET reduction. The results demonstrate that WCET reductions from 54% up to 73% can be achieved with an acceptable amount of CPU seconds required for the optimization and WCET analyses themselves.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Cache memories; B.3.3 [Memory Structures]: Worst-case analysis; D.3.4 [Programming Languages]: Compilers; Optimization

General Terms: Algorithms, Performance

*Partially funded by the European IST FP6 NoE ARTIST2.

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

In contrast to the speed of memories, processor speed has increased dramatically in the past years. To bridge the increasingly large gap between processor and memory speed, memory hierarchies based on caches are today's state of the art. Caches have the advantage of being transparent to the software running on a system – no code modification has to be done since caches are hardware controlled. Caches are effective in reducing the average-case execution time *(ACET)* of a system.

For real-time systems with hard timing constraints, caches are problematic due to their unpredictability. Since they are hardware controlled, it is virtually impossible to determine the latency of a memory access. To verify that timing constraints are met, the designer needs to know the WCET which may be heavily overestimated in the presence of caches. Such overestimates are disadvantageous since they lead to higher production costs of the entire system. For these reasons, Real-time designers have often used processors without caches. Such systems suffer a low average-case performance since each memory access is served by the slow main memory. Processors with *scratchpad* memories have both a good average-case and worst-case performance. However, caches are more common than scratchpads and can be found in almost any modern processor.

Modern caches allow to lock their contents, i. e. to protect it from being flushed by disabling the replacement. This way, it is possible to predict access times of data or instructions that have been locked in the cache, and to make precise statements about the cache's worst-case timing.

In this paper, we present techniques for compile-time decided I-cache lockdown to minimize WCETs. Our algorithms determine a set of functions of a program that is locked into the cache at system startup time. During the whole execution time of the program exploiting cache locking, the locked cache's contents remains invariant *("static locking").* Cache contents selection is done such that the set of selected functions leads to the highest WCET reductions.

WCET minimization is a difficult task due to the inherent worst-case nature of WCETs. In this paper, WCET refers to an upper bound of the maximum execution time a software can ever take. In terms of an application's control flow graph *(CFG)*, the WCET is the execution time along the longest path through the CFG from the program's start to its end node. Realistic applications typically have more than one path through their CFGs. In general, there are lots of feasible parallel paths through a CFG. Almost all these paths are irrelevant for the WCET since the WCET

only depends on the length of the worst-case execution path *(WC-path)*. However, if an optimization modifies an application and reduces its WC-path *WP*, it is possible that after this modification, a completely different path *WP'* is the new WC-path.

In the course of an optimization, the WC-path can change within the CFG. An optimization reducing WCET must take these WC-path changes into account if it wants to be effective. If not, code modifications may be performed at places in the CFG being irrelevant for the WCET since they do not lie on the current WC-path. Hence, the optimization does not perform an effective code transformation. Unfortunately, previously published cache locking optimizations for WCET reduction do not take changing WC-paths into account. The techniques presented in this paper are superior to the current state of the art in that they recompute the current WC-path after each taken decision. Hence, we can be sure to achieve maximum WCET reductions since only relevant code is locked in the I-cache. In addition, our techniques for WC-path recomputation are realized efficiently preventing excessively long runtimes of our optimizations.

Section 2 gives a survey of related work. Section 3 presents our proposed workflow of I-cache locking, followed by the WC-path aware I-cache locking algorithm in Section 4. Section 5 describes the benchmarking results, and Section 6 summarizes this paper and gives an outlook on future work.

2. RELATED WORK

The papers [4, 7] present different algorithms for static I-cache lockdown. These publications are very close to the work presented in this paper. In [7], the authors present two heuristics for cache contents selection. The first one tries to minimize CPU utilization by locking an I-cache. The second algorithm minimizes interference between tasks. In [4], an additional genetic algorithm for cache locking is proposed. All three algorithms have in common that they do not consider changing WC-paths at all. Instead, the WC-path is determined only once before the optimization process takes place. After that, optimization is done along this single WC-path. The authors admit that their approach is non-optimal. Due to the consideration of only one WC-path, we call such techniques "single-path analyses" in the following.

In [6], the techniques of [4, 7] for I-cache locking are extended to deal with changing WC-paths. However, the way how WC-paths are recomputed is not detailed. The authors use a parameter N trading off accuracy of WC-path recomputation with runtime consumption. Since runtimes for WC-path recomputation are still very high, the authors are unable to provide results for some of their benchmarks. In contrast, the techniques presented here scale much better so that we can present results for very large benchmarks.

The work of [8] is complementary to this paper since it presents a D-cache locking algorithm for WCET minimization. Using an extended version of reuse vectors, a system of cache miss equations is set up describing where data reuse translates to locality. These equations can be solved statically and define a set of data to be locked in the D-cache. Data dependencies in the CFG which can not be analyzed statically are handled by a heuristic that locks data which is likely to be accessed. In this paper on D-cache locking, the WC-path is not considered – neither implicitly nor explicitly.

In terms of predictability, locked caches behave similar to software-controlled scratchpad memories. In the past, several papers were published exploiting scratchpads for energy dissipation minimization. In [9, 10], the influence of scratchpads on WCET prediction is studied. Even though WCET is targeted in that work, the selection algorithm deciding which objects to be placed in the scratchpad is not WCET-aware. Instead, a selection algorithm for energy reduction is employed, and the effect of this energy reduction strategy on WCET is evaluated afterwards. Hence, that work is not a true WCET-aware optimization and does not consider WC-paths at all.

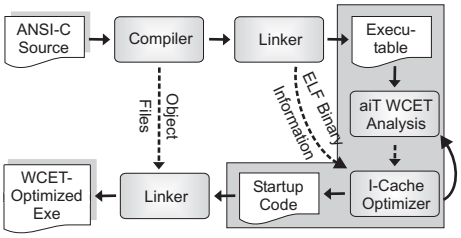


Figure 1: Workflow of I-Cache Locking

3. WORKFLOW

As target architecture, we chose the ARM920T processor. This family of ARM processors is commonly used and is thus a representative embedded processor. Its I-cache is 16 kB large. The cache is 64-way set-associative, each of the 64 sets holds 8 lines. The line size is 32 bytes. Lockdown can be done per set, always over all 8 lines of a set. Hence, locking can be realized at a granularity of 256 bytes. Locking is realized by two coprocessor registers, the victim base pointer and the victim pointer register [2].

The techniques for I-cache locking presented in this paper are realized as post-pass optimization, i. e. they take place after compilation and linking. The optimization workflow (cf. Figure 1) takes a compiled and linked binary executable for the ARM920T processor as input. Using tools from the GNU binutils suite, some useful information is extracted from this ELF binary, e. g. the contained functions, their start addresses and sizes in bytes. In a second step, static WCET analyses of the binary are performed. For this purpose, the commercial software aiT for ARM [1] is used. Depending on the structure of the binary program under analysis, aiT needs to be invoked several times. After all calls of aiT are done, WCET-relevant data of the binary's functions (e. g. execution frequencies and WCETs of functions, caller / callee relationships) are extracted from aiT's outputs. Using this data, the main selection algorithm takes place. It determines those functions to be locked in the I-cache. During optimization, the selection algorithm continuously updates its WC-path without any further WCET analysis. After the selection is done, additional startup code is emitted that performs the lockdown of the selected functions before program execution. The binary program needs to be re-linked with this new startup code again, resulting in an optimized executable.

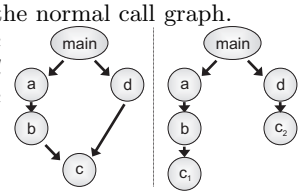
4. WC-PATH AWARE ALGORITHM FOR I-CACHE LOCKING

This section presents the proposed algorithms for WC-path aware I-cache locking. First, an execution flow graph needs to be built (cf. Section 4.1). Using this graph, the WC-path can be computed as described in Section 4.2. The

algorithm for I-cache contents selection is presented in Section 4.3.

4.1 Execution Flow Graph Generation

Using information of the WCET analyzer aiT (cf. Section 3), the context-specific function call graph *(CCG)* of the application under analysis is built. In contrast to a normal call graph, where each function is represented by exactly one graph node, a single function can be represented by several nodes in the *CCG*. This is due to the fact that aiT distinguishes so-called execution contexts for functions. Such a context contains information about all different ways a function `f` can be invoked in the normal call graph.



Example 1: *Assume the conventional call graph depicted as the left graph. As can be seen, function* `c` *is called from* `b` *and* `d`*. Hence, our WCET analyzer attaches two different contexts to* `c`*. The first context contains the information that* `c` *can be invoked via the path* `main` → `a` → `b` → `c`*, and the second context contains the path* `main` → `d` → `c`*. In the CCG, function* `c` *is instantiated twice, each node representing an invocation of* `c` *via one of the two contexts.*

In the *CCG*, each node represents a function in a certain context. An edge (x, y) indicates that function x directly calls y in a context. Weights w_x attached to nodes of the *CCG* represent a function's WCET for a particular context, and edge weights $w_{(x,y)}$ computed by aiT represent how many times y is called by x in a context.

However, the WC-path required for cache contents selection can not be computed using the *CCG*, because information about the control flow within functions is hidden in the *CCG*. For example, the code `if (z) a(); else d();` results in structurally the same *CCG* as `a(); d();` even though the resulting WC-paths are differing. To overcome this problem, the *CCG* is translated into the execution flow graph *(EFG)* which correctly models all possible execution paths between functions.

Definition 1. *The EFG is a weighted graph EFG = (V, E, w_v, w_e). V represents the set of context-specific functions. The node weight w_v represents the context-specific WCET of function v for a single execution of v. The edge weight $w_e = w_{(x,y)}$ denotes an upper bound of how many times execution flow passes from x to y in a context. The edges in the EFG are created such that all paths from the source node to the sinks in the EFG correspond to all possible ways of passing the control flow between the functions.*

Example 2: *Assume the CCG shown in Example 1, and that* `main` *looks as follows:* `if (z) a(); else d();` *Due to the if-else-statement in* `main`*, there are two mutually exclusive paths how the functions can be executed:* `main` → `a` → `b` → c_1 *or* `main` → `d` → c_2*. However, if* `main` *looks as follows:* `a(); d();`*, there is only one possible sequence of function invocations:* `main` → `a` → `b` → c_1 → `d` → c_2*. This path contains the edge* (c_1,`d`) *since after the first invocation of* `c`*, the flow of execution passes back to* `main` *where* `d` *is called unconditionally afterwards. These different constellations of execution flow need to be captured by the EFG, since it models all possible ways of invoking functions.*

To generate the *EFG*, a depth-first search of the *CCG* is combined with static WCET analysis (cf. Figure 2). The

```
1   list<node> DFS( CCG C, EFG G, node s ) {
2     list<node> pathEnds = ( s );
3     C→setVisited( s );
4     if ( G = ∅ ) {
5       C = doWCETAnalysis( s );
6       G→InsertNode( s, C→w_s ); }
7     for ( v ∈ C→getChilds( s ) )
8       if ( !C→visited( v ) )
9         if ( C→w_v == 0 ) {
10          CCG D = doWCETAnalysis( v );
11          G→InsertNode( v, D→w_v );
12          G→InsertEdge( s, v, C→w_(s,v) );
13          pathEnds += DFS( D, G, v ); }
14        else {
15          G→InsertNode( v, C→w_v );
16          for ( node n ∈ pathEnds )
17            G→InsertEdge( n, v, C→w_(s,v) );
18          pathEnds = DFS( C, G, v ); }
19    return pathEnds; }
```

Figure 2: DFS-like Algorithm for EFG Generation

CCG computed by aiT can contain nodes v with a weight of 0, i. e. functions with WCET 0. This situation indicates that v does not lie on the WC-path, because aiT reports WCETs only for those functions being on the WC-path. Hence, if a node s has non-zero weight, but a child v of s has weight zero, this implies that s calls v conditionally, as discussed in Example 2, because the WC-path contains s but not v. Using such WCETs of zero, the internal structure of functions can be deduced which is required for *EFG* construction (lines 9 - 13). If a conditional function call is detected this way, a new parallel path in the *EFG* branching from s and containing v is created.

Since our cache contents selection algorithm requires to know the WCET of all functions, including those not lying on the current WC-path, aiT is invoked several times. In the above situation (s, v) with $w_s \neq 0$ and $w_v = 0$, aiT is called again (line 10), now not with `main` as starting point, but with v. The algorithm by itself has linear complexity since each node of *CCG* is visited only once, and the resulting *EFG* has the same size as the *CCG*. The computational overhead for WCET analyses performed during *EFG* generation is estimated in Section 4.3.

4.2 WC-Path Construction

By definition, the WC-path is the path with maximal execution time the flow of control through a program can ever take. Since the *EFG* created in Section 4.1 reflects all possible execution paths within a program, the WC-path can be determined by finding the longest path from a program's start to its end nodes. Here, path lengths refer to the sum of the edge weights (execution frequencies) multiplied by node weights (execution times). To determine the longest path in *EFG*, we employed a modified variant of Dijkstra's algorithm [3] finding shortest paths in graphs (cf. Figure 3).

Our algorithm maintains three different data structures. The set of nodes S contains all nodes already processed. For a given node v, the array `d` contains the longest distance between the start node s and v. Here, distance denotes the product of a node's WCET with its execution frequency (lines 4, 13). For a node v, the array `p` contains the predecessor of v on the longest path from s to v. In contrast to Dijkstra's algorithm, our code always processes that node v_0

```
1  set<node> WC_Path( EFG G, node s, set<node> e ) {
2    set<node> P = ∅, S = { s };
3    for ( node v ∈ G→V )
4      d[v] = { 0                          if v = s
               { G→w_(s,v) * G→w_v         if (s,v) ∈ G→E
               { −∞                        otherwise
5      p[v] = { s                  if (s,v) ∈ G→E, s ≠ v
               { undefined         otherwise
6    while ( S != G→V ) {
7      v_0 = v ∈ (G→V \ S) | d[v] ⇝ max;
8      if ( d[v_0] == −∞ )
9        break;
10     S = S ∪ { v_0 };
11     for ( node v ∈ G→getChilds( v_0 ), v ∉ S )
12       if ( d[v_0] + G→w_(v_0,v) * G→w_v > d[v] ) {
13         d[v] = d[v_0] + G→w_(v_0,v) * G→w_v;
14         p[v] = v_0; } }
15   for ( node v ∈ e | d[v] ⇝ max; v ≠ s; v = p[v] )
16     P = P ∪ { v };
17   P = P ∪ { s };
18   return P; }
```

Figure 3: Algorithm for WC-Path Construction

leading to longest distances (line 7). During each iteration, the arrays `d` and `p` are invariantly updated such that longest distances are considered (lines 13, 14), instead of shortest ones in the case of Dijkstra's algorithm.

Using the array `p` computed by Algorithm 3, the WC-path from a program's start to an end node can be determined by simply indexing `p[`v`]`, starting with v equal to the program's end node with longest distance, until v is equal to the start node (lines 15 - 17).

As can be seen from Figure 3, no further WCET analyses are required for WC-path construction, since all WCET-relevant data is already included in the *EFG*. Hence, the complexity of our WC-path construction algorithm solely depends on the *EFG*. The entire algorithm depicted in Figure 3 has a complexity of $O((|V| + |E|) \log |V|)$.

4.3 I-Cache Contents Selection

The overall algorithm for WC-path aware I-Cache contents selection is depicted in Figure 4. Basically, the algorithm constructs a set L of functions to be locked in the I-cache. In the beginning, the algorithm performs the setup of the *EFG* (line 5) and of the initial WC-path (line 8). Hereafter, the algorithm iterates as long as there is an unlocked node v on the WC-path whose size fits into the remaining cache capacity (line 9).

For each other unlocked node x, its gain `g[`x`]` is computed (line 11). `g[`x`]` represents the gain when moving x from main memory to the locked I-cache, per byte of the size of x. Let w_x denote the WCET of x if x is placed in main memory, w_x^l is the WCET of x being in the I-cache, s_x the size of x in bytes, and $w_{(*,x)}$ be the overall execution frequency of x over all contexts on the WC-path. Then, `g[`x`]` is defined as

$$\texttt{g[}x\texttt{]} = \frac{w_x - w_x^l}{s_x} * w_{(*,x)}$$

s_x is extracted from the binary program currently optimized (cf. Section 3). The values w_x and $w_{(*,x)}$ are the node and edge weights of x in the *EFG*. To determine the WCET of x if x is placed in the locked I-cache, a WCET analysis is done exclusively for x without consideration of any other function of the program B. During this WCET analysis, it

```
1  set<function> CacheLocking( binary_program B ) {
2    set<function> L = ∅;
3    int S = Cache Size;
4    EFG G = ∅;
5    DFS( ∅, G, "main" );
6    for ( node v ∈ G→V | s_v ≤ S )
7      w_v^l = lockedWCETAnalysis( v );
8    set<node> P =
       WC_Path( G, "main", G→getSinks() );
9    while ( ∃ node v ∈ P |
              (v→getFunction() ∉ L) ∧ (s_v ≤ S) ) {
10     for ( node x ∈ G→V | x→getFunction() ∉ L )
11       g[x] = Compute_Gain( x, G, w_x^l, P );
12     node x = x' ∈ P | (x'→getFunction() ∉ L) ∧
                         (s_x' ≤ S) ∧ (g[x'] ⇝ max);
13     L = L ∪ { x→getFunction() };
14     S = S - s_x;
15     for ( node x' ∈ G→V |
              x'→getFunction() == x→getFunction() )
16       G→w_x' = w_x'^l;
17     P = WC_Path( G, "main", G→getSinks() ); }
18   return L; }
```

Figure 4: Algorithm for I-Cache Contents Selection

is assumed that x is entirely placed in the locked I-cache. The computation of w_x^l takes place only once during the initialization phase (line 7).

After all gains are computed, the node x with the highest gain is selected for lockdown (line 12). Lockdown is performed by adding the function represented by x to L (line 13). The amount of free I-cache space is adjusted (line 14) and the weight of all *EFG* nodes x' representing the currently locked function is set to the WCET of x' after lockdown (lines 15, 16). Using these new node weights, the new WC-path after lockdown of x is computed (line 17). Finally, the overall set of locked functions is returned which then serves for startup code generation as explained in Section 3.

Algorithm 4 has linear complexity, again. All computations of w_x^l (line 7) are as costly as one entire WCET analysis of program B. During *EFG* generation (line 5), another complete WCET analysis of B is done (cf. line 5 of Figure 2). Since only sub-graphs of B need to be re-analyzed during *EFG* generation, and since each node is visited only once, all WCET analyses done in line 10 of Figure 2 are in total as costly as one full WCET analysis of the entire program B. In total, the computational overhead for WCET analyses required by our algorithms is bounded by a maximum of 3 times of the overhead of a single WCET analysis, even though the WCET analyzer may be invoked more often.

5. EVALUATION

This section evaluates the impact of I-cache locking on WCET. First, the benchmarking workflow is presented in Section 5.1. Benchmarking results are given in Section 5.2.

5.1 Benchmarking Methodology

The techniques presented in Section 4 are fully implemented. Our tool for WC-path aware I-cache locking was applied to five real-life benchmarks. For each benchmark, Table 1 lists the size of its binary executable, the total number of functions (source code plus library functions), the number of lines of the benchmark's source codes and finally the benchmark's origin.

	Size	#Fct	#LoC	Origin
ADPCM	109 kB	19	950	mrtc.mdh.se
G723	107 kB	15	1,620	sun.com
Statemate	145 kB	21	1,201	mrtc.mdh.se
Compress	105 kB	11	528	mrtc.mdh.se
MPEG2	595 kB	210	7,916	mpeg.org/MSSG

Table 1: Benchmark Characteristics

To demonstrate the effectiveness of our techniques not only for such a large I-cache of 16 kB as the one of the ARM920T, we provide results for cache sizes varying between 64 bytes and up to 16 kB in the following. For each considered cache size, the source codes of the benchmarks were compiled and linked to an executable program. For this executable, a WCET analysis was done leading to the WCET of the unoptimized benchmark for a cache of the given size in conventional operating mode, i. e. cached instructions are replaced dynamically. The application of the workflow described in Section 3 led to an optimized binary executable of a benchmark exploiting I-cache lockdown. A second WCET analysis of this optimized executable resulted in the benchmark's WCET when using cache locking.

The used WCET analyzer aiT is able to compute WCETs only for systems with normally operating caches. Locked caches are not directly supported. To obtain WCET estimates for locked caches, we provided aiT with additional annotations. These annotations make aiT assume that the functions selected for I-cache lockdown were placed in a memory region having exactly the same access latency as a locked I-cache of the ARM920T, i. e. 1 cycle. In all WCET analyses done for benchmarking, a main memory access was assumed to require 4 additional wait states.

5.2 Benchmarking Results

WCETs

Figure 5 depicts the effect of our WC-path aware I-cache lockdown strategy on the resulting WCETs. It shows the WCETs of the benchmarks after I-cache locking as a percentage of the WCETs without cache locking. The 100% base line thus reflects the WCETs of the benchmarks for a system with I-cache in normal operation mode.

As can be seen from Figure 5, the proposed techniques achieve significant WCET reductions for all cache sizes. Already for a very small I-cache of 64 bytes, lockdown leads to improvements between 2% (ADPCM) and 38.3% (G723). The large WCET reduction for G723 is caused by locking two very small but WCET-critical functions for absolute value computation and quantization.

With increasing I-cache sizes, even higher WCET reductions were achieved since more functions were locked in the I-cache, thus leading to a proportionally higher predictability of instruction fetches. Per benchmark, a monotonic decrease of WCETs was observed with increasing I-cache sizes. For some benchmarks (G723, MPEG2), smoothly decreasing WCET curves were obtained. This smooth decrease shows that many small functions lie on the WC-path of these benchmarks. By increasing the I-cache size by only a few bytes, these benchmarks benefit since the additional cache space is used for lockdown of functions on the WC-path.

In contrast, ADPCM exhibits a stepwise WCET decrease with increasing cache sizes. Here, the steep WCET reduction down to 34% when increasing the cache size from 128 bytes to 256 bytes stems from the lockdown of a system library function for integer division. This system function is heavily used within ADPCM during *sine* and *cosine* computations. When increasing the I-cache size further to 512 bytes, another reduction of WCET down to 27% was observed. Hereafter, a saturation is reached so that caches larger than 512 bytes do not lead to more reductions of the WCET of this benchmark.

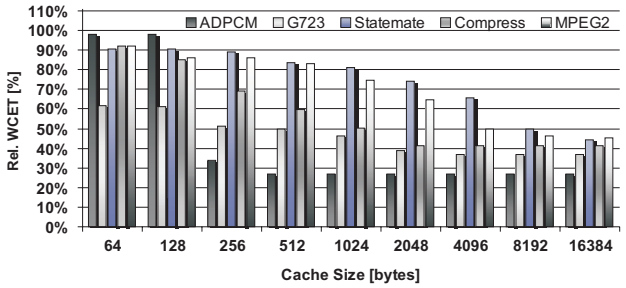


Figure 5: Relative WCETs after I-Cache Locking

The overall largest WCET reductions were achieved for an I-cache size of 16 kB which is the size of the ARM920T's native configuration. For this scenario, WCET reductions between 54.6% (MPEG2) and 73.1% (ADPCM) were measured. On average over all five benchmarks, a WCET reduction of 60.1% was achieved for a 16 kB I-cache.

Overhead of Startup Code

Obviously, I-cache locking involves some overhead, since additional startup code realizing lockdown is required (cf. Section 3) taking additional cycles. These cycles are not included in Figure 5, because WCET analysis using aiT starts with the function `main`, and the startup code is executed before `main`. However, our measurements have shown that the overhead is negligible. With increasing code sizes to be locked, the overhead also increases, whereas the WCETs of the benchmarks decrease. Hence, when expressing the lockdown overhead as percentage of a benchmark's WCET cycles for a given cache size, the overhead must be maximal for a cache size of 16 kB. For 16 kB, the overhead ranges from 0.01% (MPEG2) up to 10.8% (Compress). The overhead for Compress at 16 kB is misleading since it is the smallest benchmark in our setup (cf. Table 1). Compress reaches a saturation point at 2 kB – larger locked caches do not translate into further WCET reductions. For a 2 kB cache, the overhead for I-cache lockdown of Compress is only 1.35%. As can be seen, the overhead is by far over-compensated by the savings achieved using I-cache lockdown.

With constantly 300 bytes, the size overhead of the startup code is negligible, too.

CPU Runtimes for Optimization

As discussed in Section 4, the WCET analyzer is invoked several times during the entire optimization process. This behavior is due to the fact that several WCET analyses are required to obtain all the execution frequencies and WCETs of all functions of a benchmark, once when being locked in the I-cache and once when residing in main memory. However, a noticeable contribution of this work is that all required WCET analyses are performed off-line during an initialization phase. No WCET analysis is done during the entire optimization process at all.

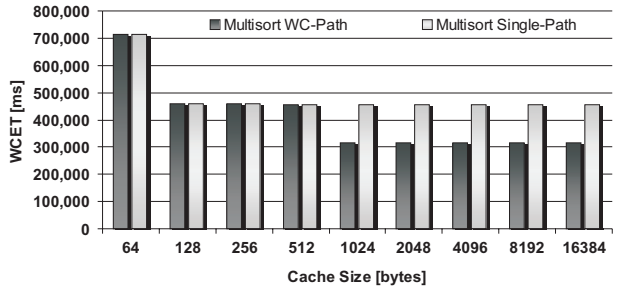


Figure 6: WCET Comparison with Single-Path Analysis

For ADPCM, the entire optimization for a given cache size only takes 6 CPU seconds on an AMD Athlon system running at 2.4 GHz. These 6 CPU seconds include all required WCET analyses as well as the time spent by the algorithms proposed in Section 4. For MPEG2 being by far the largest benchmark (595 kB), a total of 2,179 CPU seconds is required. These 36 CPU minutes are definitely not lightweight. But under consideration that 52 WCET analyses are needed for MPEG2, and since these 36 CPU minutes are only required once during initialization, our approach is still perfectly suited to generate highly efficient production code or to perform design space exploration.

In contrast to existing literature [6], our approach does not lead to excessively increasing CPU runtimes. The algorithms presented in this paper have linear or logarithmical complexity. Since WCET analyses are done during a preprocessing step before the core optimization algorithms are executed, our overall runtimes including WCET analysis time are highly moderate, whereas those of e. g. [6] do not seem to scale well.

Comparison with Single-Path Analysis

Besides this aspect of scalability, another important contribution of our work is the explicit consideration of changing WC-paths during optimization, in contrast to "single-path analyses" like e. g. [4, 7]. Figure 6 shows the resulting absolute WCETs for varying I-cache sizes for the multisort benchmark [10]. Per cache size, the WCETs of our techniques ("Multisort WC-path") are compared with the results of an I-cache locking mechanism similar to [4, 7] and computing the WC-path only once during initialization ("Multisort Single-Path").

As can be seen, both cache allocation techniques lead to the same numbers for cache sizes of 64 to 512 bytes. When increasing the cache size from 64 bytes to 128 bytes, both algorithms move a bubblesort routine being on the initial WC-path onto the I-cache. However, this lockdown of bubblesort on the I-cache leads to a change of the WC-path, which, in contrast to our approach, is not considered by the single-path technique. As a consequence, the single-path technique is unable to reduce WCETs further since it operates on incorrect WCET data from this moment on. Our WC-path aware cache locking technique keeps track of the changing WC-path and locks a selectionsort routine onto the I-cache for cache sizes greater equal than 1 kB. As can be seen from this example, our cache allocation technique reduces the WCET of the benchmark by 55%, whereas the single-path approach only leads to 36% of improvement.

6. CONCLUSIONS

This paper presents an approach for compile-time decided I-cache locking to minimize WCETs of real-time systems. The contributions of this work are twofold. First, we consider the phenomenon that WC-paths may change during the course of an optimization by keeping track of such changing WC-paths within our optimization. Second, our integrated WC-path recomputation techniques are realized in an efficient way preventing the runtimes of our optimizations to increase excessively. Combined, these contributions represent an improvement over the current state of the art which either consists of not considering changing WC-paths at all, or leads to unacceptable optimization runtimes.

For the presented complex benchmarks, we report WCET reductions between 54% and up to 73% for an ARM920T processor. These improvements were achieved within an acceptable amount of time required for optimization. For most of the benchmarks, the entire optimization takes place within a few CPU seconds. Only for a complete MPEG2 encoder, several CPU minutes were required. The benefit of considering changing WC-paths during optimization is demonstrated by a comparison of our approach with a technique representing the current state of the art. This comparison shows that we are able to outperform the so-called single-path techniques by more than 30%.

In the future, we will focus on lockdown of D-caches and on predictable locking schemes modifying the cache contents at runtime. We will also integrate the proposed algorithms into a WCET-aware C compiler [5] to exploit lockdown of code fragments smaller than functions, like e. g. basic blocks.

7. REFERENCES

[1] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzer aiT for ARM. 2006.

[2] *ARM920T Technical Reference Manual*. Advanced RISC Machines Ltd., Literature Number ARM DDI 0151C, 2002.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[4] A. M. Campoy, I. Puaut, A. P. Ivars, et al. Cache contents selection for statically-locked instruction caches: An Algorithm Comparison. In *Proc. of ECRTS*, July 2005.

[5] H. Falk and P. Lokuciejewski. Design of a WCET-Aware C Compiler. In *Proc. of ESTIMedia*, Oct. 2006.

[6] I. Puaut. WCET-centric Software-controlled Instruction Caches for Hard Real-Time Systems. In *Proc. of ECRTS*, July 2006.

[7] I. Puaut and D. Decotigny. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proc. of RTSS*, Dec. 2002.

[8] X. Vera, B. Lisper, and J. Xue. Data Cache Locking for Higher Program Predictability. In *Proc. of SIGMETRICS*, June 2003.

[9] L. Wehmeyer and P. Marwedel. Influence of Onchip Scratchpad Memories on WCET Prediction. In *Proc. of WCET*, June 2004.

[10] L. Wehmeyer and P. Marwedel. Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software. In *Proc. of DATE*, Mar. 2005.