WCET-AWARE SOFTWARE BASED CACHE PARTITIONING FOR MULTI-TASK REAL-TIME SYSTEMS [1]

Sascha Plazar[2], Paul Lokuciejewski[2], Peter Marwedel[2]

Abstract

Caches are a source of unpredictability since it is very difficult to predict if a memory access results in a cache hit or miss. In systems running multiple tasks steered by a preempting scheduler, it is even impossible to determine the cache behavior since interrupt-driven schedulers lead to unknown points of time for context switches. Partitioned caches are already used in multi-task environments to increase the cache hit ratio by avoiding mutual eviction of tasks from the cache.

For real-time systems, the upper bound of the execution time is one of the most important metrics, called the Worst-Case Execution Time (WCET). In this paper, we use partitioning of instruction caches as a technique to achieve tighter WCET estimations since tasks can not be evicted from their partition by other tasks. We propose a novel WCET-aware cache partitioning algorithm, which determines the optimal partition size for each task with focus on decreasing the system's WCET for a given set of possible partition sizes. Employing this algorithm, we are able to decrease the WCET depending on the number of tasks in a set by up to 34%. On average, reductions between 12% and 19% can be achieved.

1 Introduction

Embedded systems often operate as hard real-time systems which have to meet hard timing constraints. For these systems, it is mandatory to know the upper bound of the execution time for each task and possible input data. This bound is called *Worst-Case Execution Time*.

Caches have become popular to bridge the gap between high processor and low memory performance. The latency for an access to a certain memory address highly depends on the content of the cache. If an instruction to be fetched already resides in the cache, then a so called *cache hit* occurs and the fetch can be usually performed within one cycle. Otherwise, it results in a *cache miss*. The desired address has to be fetched from the slow main memory (e.g. Flash) leading to penalty cycles depending on the processor and memory architecture.

It is hard to determine statically if an arbitrary memory access results in a cache hit or a cache miss. However, caches are used in real-time systems because they can drastically speed up the execution of programs. Hence, a lot of effort has been successfully put into research to make sound prediction about the worst-case cache performace during a program's execution. AbsInt's *aiT* [2] is a tool that performs static analyses on binary programs to predict their cache behavior and WCET.

[1]The research leading to these results has received funding from the European Community's ArtistDesign Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n^{o} 216008.

[2]Computer Science 12 — TU Dortmund University — D-44221 Dortmund, Germany — FirstName.LastName@tu-dortmund.de

In environments with preemptive schedulers running more than one task, it is impossible to make any assumption about the memory access patterns. This is mainly caused by interrupt-driven scheduling algorithms causing context switches at unknown points of time. Thus, the program's state in not known at which a context switch occurs. It is also unknown at which address the execution of a program continues, hence it is unknown which line of the cache is evicted next. An unknown cache behavior forces to assume a cache miss for every memory access implying a highly overestimated systems overall WCET. As a consequence, the underlying system has to be oversized to meet real-time constraints resulting in higher costs for hardware.

We adapt an existing technique called software based cache partitioning [16] to make the instruction cache (*I-cache*) behavior more predictable. This can be guaranteed since every task has its own cache partition from which it can not be evicted by another task. Our novel WCET-aware cache partitioning aims at selecting the optimal partition size for each task of a set to achieve the optimal WCET minimization. The main contributions of this paper are as follows:

1. Compared to existing approaches which focus on minimization of average-case execution times, our WCET-aware cache partitioning explicitly evaluates WCET data as metric for optimization.
2. In contrast to previous works which presented theories to partition a cache in software, our approach comprises a fully functional implementation of a cache partitioning method.
3. We show that our ILP-based WCET-aware cache partitioning is effective in minimizing a system's WCET and outperforms existing algorithms.

The paper is organized as follows: In the next section, we present related work. Existing techniques to partition a cache as well as our new algorithm are explained in Section 3. Section 4 introduces the compiler WCC used to integrate our novel cache partitioning algorithm. An evaluation of the performance which is achieved by our WCET-aware cache partitioning, is presented in Section 5. Finally, we conclude our work and give a brief overview of future work.

2 Related Work

The papers [16, 5, 15] present different techniques to exploit cache partitioning realized either in hardware or in software. In contrast to our work, these implementations either do not take the impact on the WCET into account or do not employ the WCET as the key metric for optimization which leads to suboptimal or even degraded results. In [16], the author presents ideas for compiler support for software based cache partitioning which serves as basis for the partitioning techniques presented in this paper. Compared to the work in this paper, a functional implementation or impacts on the WCET are not shown. In [5], a hardware extension for caches is proposed to realize a dynamic partitioning through a fine grained control of the replacement policy via software. Access to the cache can be restricted to a subset of the target cache set which is called columnization. For homogeneous on-chip multi-processor systems sharing a unified set-associative cache, [15] presents partitioning schemes based on associativity and sets.

A combination of locking and partitioning of shared caches on multi-core architectures is researched in [18] to guarantee a predictable system behavior. Even tough the authors evaluate the impact of their caching schemes on the worst-case application performance, their algorithms are not WCET-aware. Kim et al. [11] developed an energy efficient partitioned cache architecture to reduce the energy per access. A partitioned L1-cache is used to access only one sub-cache for every instruction fetch leading to dynamic energy reduction since other sub-caches are not accessed.

The authors of [4] show the implications of code expanding optimizations on instruction cache design. Different types of optimizations and their influence on different cache sizes are evaluated. [12] gives an overview of cache optimization techniques and cache-aware numerical algorithms. It focuses on the bottleneck memory interface which often limits the performance of numerical algorithms.

Puaut et al. counteract the problem of unpredictability with locked instruction caches in multi-task real-time systems. They propose two low complexity algorithms for cache content selection in [17]. A drawback of statically locking the cache content is that the dynamic behavior of the cache gets lost. Code is no more automatically loaded into the cache, thus code which is not locked into the cache can not profit from it anymore.

Vera et al. [22] combine cache partitioning, dynamic cache locking and static cache analysis of data caches to achieve predictability in preemptive systems. This eliminates overestimation and allows to approximate the worst-case memory performance.

Lokuciejewski et. al rearrange the orders of procedures in main memory to exploit locality in the control flow leading to a higher cache performance [13]. Worst-case calling frequencies serve as metrics for WCET minimization but multi-task sets are not supported.

3 WCET-aware Cache Partitioning

Caches have become popular to bridge the growing gap between processor and memory performance since they are transparent from the programmer's point of view. Unfortunately, caches are a source of unpredictability because it is very difficult to determine if a memory access results in a cache hit or a cache miss. Static analysis is a technique to predict the behavior of the cache [19] and make sound prediction about the WCET of a program which allows the profitable usage of caches in real-time systems running a single task.

In general, real-time systems consist of more than one task which makes it often impossible to determine the worst-case cache behavior. Due to interrupt driven schedulers, points of time for context switches can not be statically determined. Thus, it is not predictable which memory address is fetched from the next task being executed and one can not make proven assumptions which cache line is replaced by such an unknown memory access. Due to this fact, every memory access has to be treated as a cache miss leading to a highly overestimated WCET caused by an underestimated cache performance.

In a normally operating cache, each task can be mapped into any cache line depending on its memory usage. To overcome this situation, partitioned caches are recommended in literature [16, 5, 15]. Tasks in a system with partitioned caches can only evict cache lines residing in the partition they are assigned to. Reducing the prediction problem of replaced cache lines to one task with its own cache partition, allows the application of well known single task approaches for WCET- and cache performance estimation. The overall execution time of a task set is then composed of the execution time of the single tasks with a certain partition size and the overhead required for scheduling including additional time for context switches.

Infineon's TriCore architecture does not support partitioned caches in hardware so that partitioning has to be done in software. The following section describes the basics of software based cache partitioning schemes applied in our WCET-aware cache partitioning algorithms. In Section 3.2, a heuristic approach is applied to determine the partitioning based on the tasks' sizes. Section 3.3 presents our novel algorithm for selecting an optimal partition size w.r.t. the overall WCET of a system.

3.1 Software based Cache Partitioning

The author in [16] presents a theory to integrate software based cache partitioning into a compiler toolchain without an existing implementation. Code should be scattered over the address space so that tasks are mapped to certain cache lines. Therefore, all tasks have to be linked together in one monolithic binary and a possible free space between several parts has to be filled with *NOPs*. Partitioning for data caches involves code transformation of data references.

The theory to exactly position code in the address space to map it into certain cache lines is picked up here, but a completely different technique is applied to achieve such a distribution. We restrict ourselves to partitioning of I-caches, thus only software based partitioning of code using the new technique is discussed. However, a partitioning of data caches w.r.t. WCET decrease is straightforward using a modified version of our algorithm.

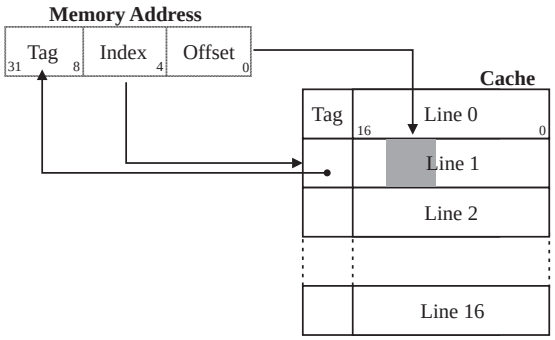

Figure 1: Addressing of cache content

For the sake of simplicity, in the following we assume a direct-mapped cache. Way associative caches can be partitioned as well: The desired partition size has to be divided by the degree d of associativity since any particular address in main memory can be mapped in one of d locations in the cache. In this case, predictable replacement policies (e.g. *last recently used* for TriCore) are allowed to enable static WCET-analysis. However the replacement policy has no influence on the partitioning procedure.

Assuming a very small cache with $S = 256$ bytes capacity divided into $l = 16$ cache lines, results in a cache line size of $s = 16$ bytes. When an access to a cached memory address is performed, the address is split into a tag, an index, and an offset part. Our example in Figure 1 shows the 4 offset bits addressing the content inside a cache line, whereas 4 index bits select a cache line. The remaining address bits form the tag which is stored in conjunction with the cache line. The tag bits have to be compared for every cache access since arbitrary memory addresses with the same index bits can be loaded into the same line.

To partition a cache, it has to be ensured that a task assigned to a certain partition only allocates memory addresses with index bits belonging to this partition. For an instruction cache divided into two partitions of 128 bytes, one partition ranges from cache line 0 to line 7 and the second one from line 8 up to 15. If a task T_1 is assigned to the first partition, each occupied memory address must have index bits ranging from `000b` up to `111b` accessing the cache lines 0 to 7 and arbitrary offset bits. Together, index and offset bits correspond to memory addresses modulo cache size ranging from `0x00` to `0x7f`. A task T_2 assigned to the second partition has to be restricted to cover only memory addresses modulo cache size from `0x80` up to `0xff`.

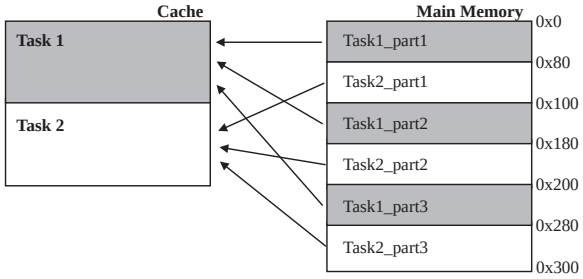

Figure 2: Mapping of tasks to cache lines

Tasks exceeding the size of the partition they are mapped to, have to be split and scattered over the address space. Figure 2 illustrates the partitioning for tasks T_1 and T_2 into such 128 bytes portions and the distribution of these portions over the main memory. Task T_1 is allocated to portions which are mapped to the first half of the cache since all occupied memory addresses modulo cache size

range from 0-127. The same has to meet for task T_2 occupying memory addresses modulo cache size ranging from 128-255.

Obviously, partitioning does not depend on the cache line size since a contiguous part of the memory is always mapped into the same amount of cache memory. Only the atomic size for composing partitions is equal to the cache line size, thus the partition size must be a multiple thereof.

WCC's workflow employs the linker to achieve such a distribution of code over the address space. Individual linker scripts are used (compare Listing 1 for the aforementioned example) with relocation information for every task and its portions it is divided into. For linker basics refer to [3].

The output section `.text`, to be created in the output binary (line 1), is aligned to a memory address which is a multiple of the cache size to ensure that the mapping starts at cache line 0. Line 3 allocates the assembly input section `.task1_part1` at the beginning of the `.text` output section, thus the beginning of the cache. The content of this section must not exceed 128 bytes since line 4 sets the relocation counter to the address 128 bytes beyond the start address, which is mapped into the first line of the second cache half. Line 5 accomplishes the relocation of section `.task2_part1` to the new address. The other sections are mapped in the same manner.

```
.text: {                                  1
  _text_begin = .;                        2
  *(.task_part1)                          3
  . = _text_begin + 0x80;                 4
  *(.task2_part1)                         5
  . = _text_begin + 0x100;                6
  *(.task1_part2)                         7
  . = _text_begin + 0x180;                8
  *(.task2_part2)                         9
  . = _text_begin + 0x280;                10
  *(.task2_part3)                         11
} > PFLASH-C                              12
```

Listing 1: Linker script example

On the assembly level, each code portion which should be mapped to a partition, has to be attached to its own linker section to cause a relocation by the linker; e.g. `.task_part1` for the first 128 bytes memory partition of task T_1. To restore the original control flow, every memory partition has to be terminated by an additional unconditional branch to the next memory partition of the task unless the last instruction of this block already performs an unconditional transfer of control.

For further jump corrections required by growing displacements of jump targets and jump sources refer to [16].

3.2 Size-driven Partition Size Selection

The author in [16] propose to select a task's partition size depending on its size relative to the size of the complete task set. For our example, a task set with $m = 4$ tasks $T_1 - T_4$ having a size of $s(T_1) = 128$ bytes, $s(T_2) = 256$ bytes, $s(T_3) = 512$ bytes and $s(T_4) = 128$ bytes should be assumed. Hence, the complete task set has an overall code size of 1 kB, whereas we use the assumed cache from the previous section with a capacity of $S = 256$ bytes.

According to its size, task T_i's partition size computes as follows:

$$p(T_i) = \frac{s(T_i)}{\sum_{j=1}^{n} s(T_j)} * S_{cache} \tag{1}$$

e.g. T_1 is assigned to a partition with $128\ bytes/1024\ bytes = 1/8$ of the cache size. Accordingly the assigned partition sizes are: $p(T_1) = 32$ bytes, $p(T_2) = 64$ bytes, $p(T_3) = 128$ bytes and $p(T_4) = 32$ bytes.

3.3 WCET-driven Partition Size Selection

The size of a cache may have a drastic influence on the performance of a task or an embedded system. Caches with sufficient size can decrease the runtime of a program whereas undersized caches can lead

to a degraded performance due to a high cache miss ratio. Hence, it is essential to choose the optimal partition size for every task in order to achieve the highest possible decrease of the system's overall WCET.

Current approaches select the partition size depending on the code size or a tasks priority [16, 18]. They aim at improving a system's predictability and examine the impact of partitioning on the WCET but do not explicit aim at minimizing its WCET.

In this section, we present our novel approach to find the optimal partition sizes for a set of tasks w.r.t. the lowest overall WCET of a system. We use integer linear programming (*ILP*) to select the partition size for each task from a given set of possible partition sizes.

We assume that there is a set of m tasks which are scheduled periodically. There is a schedule interval within each task $T_i \in T$ is executed exactly c_i times, which is repeated continuously. The length of this interval is the least common multiple of the m tasks' periods. Furthermore, we assume a set P of given partition sizes with $|P| = n$ partitions, e.g. $P = \{0, 128, 256, 512, 1024\}$ measured in bytes. Let x_{ij} be a binary decision variable determining if task T_i is assigned to a partition with size $p_j \in P$:

$$x_{ij} = \begin{cases} 1, \text{ if } T_i \text{ assigned to } p_j \\ 0, \text{ else} \end{cases}$$

To ensure that a task is assigned to exactly one partition, the following m constraints have to be met:

$$\forall i = 1..m : \sum_{j=1}^{n} x_{ij} = 1 \tag{2}$$

$WCET_{ij}$ is T_i's WCET for a single execution if assigned to partition p_j, then the WCET for a single task T_i is calculated as follows:

$$WCET(T_i) = \sum_{j=1}^{n} x_{ij} * WCET_{ij}$$

Since we focus on WCET minimization, we define the cost function to be minimized for the whole task set:

$$WCET = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} * c_i * WCET_{ij} \tag{3}$$

To keep track of the limited cache size S we introduce another constraint:

$$\sum_{j=1}^{m} \sum_{i=1}^{n} x_{ij} * p_j \ \leq S \tag{4}$$

Using equations 2 to 4, we are able to set up the cost function and $m + 1$ constraints as input for

Input: Set of tasks T, set of partition sizes P, execution counts C, cache size S
Output: Set of partitioned tasks T

```
1  begin
2    for t_i ∈ T do
3      for p_j ∈ P do
4        partitionTask( t_i, p_j);
5        WCET_ij = determineWCET( t_i );
6        WCET = WCET ∪ WCET_ij;
7      end
8    end
9    ilp = setupEquat( T, P, WCET, C, S);
10   X = solveILP( ilp);
11   forall x_ij ∈ X : x_ij = 1 do
12     partitionTask( t_i, p_j);
13   end
14   return T;
15 end
```

Algorithm 1: **Pseudo code of cache partitioning algorithm**

an ILP solver like *lp_solve* [1] or *CPLEX* [10]. After solving the set of linear equations, the minimized WCET and all variables $x_{ij} = 1$, representing the optimal partition sizes for all tasks, are known.

The number of necessary WCET analyses depends on the number of tasks and the number of possible partition sizes which have to be taken into account: $\#Analyses_{WCET} = |T| * |P| = m * n$

To determine the WCETs, to set up all equations and to apply partitioning, Algorithm 1 is employed. A given task set, the instruction cache size and a set of possible partition sizes for the tasks are required as input data. The algorithm iterates over all tasks (line 2) and temporary partitions each task (line 3 to 4) for all given partition sizes. Subsequently, the WCET for the partitioned task is determined invoking AbsInt's static analyzer aiT (line 5). Exploiting the information about tasks, partition sizes, cache size and gathered WCETs, an ILP model is generated regarding equations 3 to 4 (line 8) and solved in line 9.

Afterwards, the set X includes exactly one decision variable x_{ij} per task T_i with the value 1 whereas p_j is T_i's optimal partition size w.r.t. minimization of the system's WCET. Finally, in lines 11 to 12 a software-based partitioning of each task with its optimal partition size, as described in Section 3.1, is performed.

4 Workflow

Software based cache partitioning needs the support of an underlying compiler to collect WCET-data, perform the required code modifications and scatter the code over the address space. We employ our WCET-aware C compiler framework, called *WCC* [8], intended to develop various high- and low-level optimizations. WCC is a compiler for the Infineon TriCore TC1796 processor coupling AbsInt's static WCET analyzer *aiT* [2] which provides WCET data that is imported into the compiler backend and made accessible for optimizations.

Figure 3 depicts WCC's internal structure reading the tasks of a set 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 transformed into our high-level intermediate representation (*IR*) *ICD-C* [6]. Each task in a set is represented by its own IR.

In the next step, the *LLIR Code Selector* translates the high-level IRs into low-level IRs called *ICD-LLIR* [7]. On these TriCore TC1796 specific assembly level IRs, the software based cache partitioning can be performed. To enable such a WCET-aware optimization, AbsInt's *aiT* is employed to perform static WCET analyses on the low-level IRs. Therefore, mandatory information about loop bounds and recursion depth is supplied by flow fact annotations.

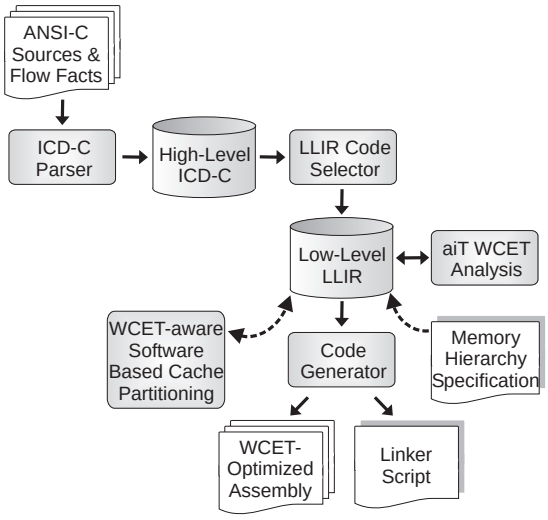

Figure 3: Workflow of the WCET-aware C compiler WCC

Optimizations exploiting memory hierarchies such as our novel software based cache partitioning require detailed information about available memories, their sizes and access times. For this purpose, WCC integrates a detailed memory hierarchy specification available on *ICD-LLIR* level.

Finally, WCC emits WCET-optimized assembly files and generates suitable binaries using a linker script reflecting the utilized internal memory layout.

5 Evaluation

This section compares the capability of our WCET-driven cache partitioning to existing partition size selection heuristic based on tasks sizes. We use different task sets from media and real-time benchmark suites to evaluate our optimization on computing algorithms typically found in the embedded systems domain. Namely, tasks from the suites *DSPstone* [21], *MRTC* [14] and *UTDSP* [20] are evaluated. WCC supports the Infineon TriCore architecture whose implementation in form of the TC1796 processor is employed for the evaluation. The processor integrates a 16 kB 2-way set associative I-cache with 32 bytes cache line size.

Overall, the used benchmark suites include 101 benchmarks so that we have to limit to a subset of tasks for cache partitioning. For lack of specialized benchmarks suites, sets of tasks stemming from the mentioned benchmark suite, as proposed in [9], are generated and compiled with the optimization level `-O3`. Using these sets, we benchmark the capability of decreasing the WCET achieved by standard partitioning algorithms compared to our WCET-aware approach.

Different numbers of tasks (5, 10, 15) in a set are checked to determine their effect on the WCET. To gather presentable results, we compute the average of 100 sets of randomly selected tasks for each considered cache sizes and the differing task set sizes. Seven cache sizes with the power of two are taken into account, ranging from 256 bytes up to 16 kB. Thus, the overall number of ILPs for every benchmark suite, which has to be generated and solved, is:

$$|ILPs| = 3 * 100 * 7 = 2100$$

Due to the fact that we do not take scheduling into account for benchmarking, the tasks execution frequencies c_i (cf. equation (3.3)) are set to one, thus, the system's WCET is composed of the task's WCETs for a single execution.

Figure 4 shows the relative WCETs for the benchmark suite *DSPstone Floating Point*, achieved by our novel optimization presented in Section 3.3 as percentage of the WCET achieved by the standard heuristic presented in Section 3.2. The nominal sizes of the task sets range on average from 1.5 kB for 5 tasks up to 5 kB for 15 tasks. Substantial WCET reductions can only be obtained for smaller caches of up to 1 kB since almost all tasks fit into the cache from 4 kB on. There, WCET reductions between 4% and 33% can be observed. In general, larger task sets result in higher optimization potential for all cache sizes.

Figure 5 depicts the average WCET for the *MRTC* benchmark suite. The average code size of the generated task sets is comparatively large with 6 kB for 5 tasks, 12 kB for 10 tasks and 19 kB for 15 tasks. Hence, there is more potential to find a better distribution of partition sizes. This can be seen in a nearly

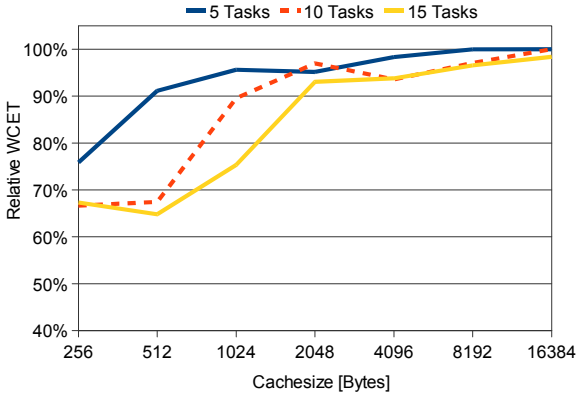

Figure 4: Optimized WCET for DSPstone Floating Point relative to standard approach

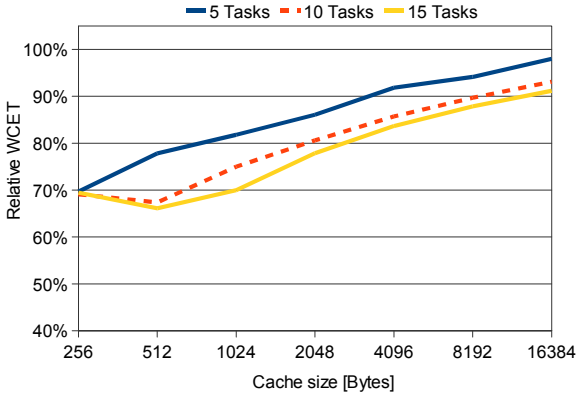

Figure 5: Optimized WCET for MRTC relative to standard approach

linear correlation of the optimization potential and the quotient of task set size and cache size. For 5 tasks in a set, WCET reductions up to 30% can be gained. 10 tasks per set have a higher optimization potential, so that 7% to 31% decrease of WCET can be observed. Optimizing the sets of 15 tasks, 9% up to 31% lower WCETs can be achieved.

A similar situation can be observed in Figure 6 for the *UTDSP* benchmark suite. The average code sizes for the task sets range from 9 kB to 27 kB. This leads to an optimization potential of 4% for a 5 task set completely fitting into the cache and 17% up to 36% for a 15 task set especially for small cache sizes. For this benchmark suite, the same behavior can be observed: for smaller cache sizes and larger code sizes our algorithm achieves better results compared to the standard approach.

Using caches larger than 16 kB, our algorithm is not able to achieve better or only marginal better results than if the standard method from section 3.2 is applied. This comes from the fact that mostly there is no optimization potential if all tasks fit into the cache. For realistic applications, the cache would be much smaller than the amount of code. There is also no case where the standard algorithm performs better than our approach since we use ILP models to always obtain the optimal partition size distribution.

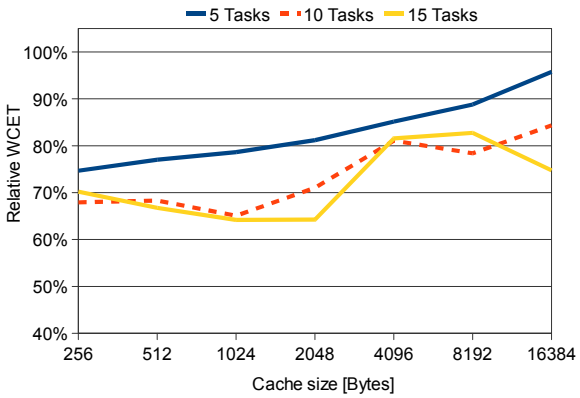

Figure 6: Optimized WCET for UTDSP relative to standard approach

Compilation Time

To consider compilation and optimization time on the host system, we utilize an Intel Xeon X3220 (2.40 GHz). A complete toolchain iteration is decomposed into the three phases compilation, WCET analysis, and optimization. The stage WCET analysis comprises all aiT invocations necessary to compute the tasks' WCETs for possible partition sizes.

The time required for a combined compilation and optimization phase ranges from less than one second (*fir* from MRTC) to 30 seconds for *adpcm* from UTDSP. Compared to this, the duration for performing static WCET analyses used for construction of an ILP is significantly higher with up to 10 hours .

6 Conclusions and Future Work

In this paper, we show how to exploit software based cache partitioning to improve the predictability of worst-case cache behavior in focus of multi-task real-time systems. Employing partitioned caches, every task has its own cache area from which it can not be evicted by other tasks. We introduce a novel algorithm for WCET-aware software based cache partitioning in multi-task systems to achieve predictability of cache behavior. The linker is exploited to achieve a restriction of tasks to be mapped into certain cache lines. An ILP model, based on the tasks' WCETs for different partition sizes, is set up and solved to select the optimal partition size for each task w.r.t. minimizing the systems WCET.

The new technique was compared to simple partition size selection algorithms in order to demonstrate its potential. The results show that our algorithm always finds better combinations of tasks' partition sizes than the size-based approach. Inspecting small task sets, we are able to decrease the WCET up to 30% compared to the standard approach. Better results can be achieved for larger task sets with up to 33% WCET reduction.

On average, we were able to outperform the size-based algorithm by 12% for 5 tasks in a set, 16% for task sets with 10 tasks, and 19% considering tasks sets with 15 tasks.

In general, the larger the task sets (and by association the code sizes) are, the better the results. This means: the algorithm performs best for realistic examples and less well for small (more academic) examples.

In the future, we intend to extend our algorithm to support partitioning of data caches. This enables predictable assumptions for the worst-case behavior of data caches accessed by multiple tasks in embedded systems with preempting schedulers. Another goal is the tightly coupling of offline scheduling algorithm analyses to automatically prefer those tasks during optimization which miss their deadlines.

References

[1] lp_solve reference guide. `http://lpsolve.sourceforge.net/5.5/`. v. 5.5.0.14.

[2] ABSINT ANGEWANDTE INFORMATIK GMBH. Worst-Case Execution Time Analyzer aiT for TriCore. `http://www.absint.com/ait`.

[3] CHAMBERLAIN, S., AND TAYLOR, I. L. *Using ld*, 2000. Version 2.11.90, `http://www.skyfree.org/linux/references/ld.pdf`.

[4] CHEN, W. Y., CHANG, P. P., CONTE, T. M., AND HWU, W. W. The Effect of Code Expanding Optimizations on Instruction Cache Design. *IEEE Trans. Comput. 42*, 9 (1993).

[5] CHIOU, D., RUDOLPH, L., DEVADAS, S., AND ANG, B. S. Dynamic Cache Partitioning via Columnization. In *Proceedings of DAC* (2000).

[6] ECKART, J., AND PYKA, R. ICD-C Compiler Framework. `http://www.icd.de/es/icd-c`, 2009. Informatik Centrum Dortmund, Embedded Systems Profit Center.

[7] ECKART, J., AND PYKA, R. ICD-LLIR Low-Level Intermediate Representation. `http://www.icd.de/es/icd-llir`, 2009. Informatik Centrum Dortmund, Embedded Systems Profit Center.

[8] FALK, H., LOKUCIEJEWSKI, P., AND THEILING, H. Design of a wcet-aware c compiler. In *Proceedings of WCET* (`http://ls12-www.cs.tu-dortmund.de//research/activities/wcc`, 2006).

[9] HARDY, D., AND PUAUT, I. WCET Analysis of Multi-Level Non-Inclusive Set-Associative Instruction Caches. In *Proceedings of RTSS* (2008).

[10] ILOG. CPLEX. `http://www.ilog.com/products/cplex`.

[11] KIM, C., CHUNG, S., AND JHON, C. An Energy-Efficient Partitioned Instruction Cache Architecture for Embedded Processors. *IEICE - Trans. Inf. Syst.*, 4 (2006).

[12] KOWARSCHIK, M., AND WEI, C. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies* (2003), Springer.

[13] LOKUCIEJEWSKI, P., FALK, H., AND MARWEDEL, P. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proceedings of ECRTS* (Prague/Czech R., 2008).

[14] MÄLARDALEN WCET RESEARCH GROUP. Mälardalen WCET benchmark suite. `http://www.mrtc.mdh.se/projects/wcet`, 2008.

[15] MOLNOS, A., HEIJLIGERS, M., COTOFANA, S. D., AND EIJNDHOVEN, J. Cache Partitioning Options for Compositional Multimedia Applications. In *Proceedings of ProRISC* (2004).

[16] MUELLER, F. Compiler Support for Software-Based Cache Partitioning. *SIGPLAN Not. 30*, 11 (1995).

[17] PUAUT, I., AND DECOTIGNY, D. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proceedings of RTSS* (Washington, DC, USA, 2002), IEEE Computer Society.

[18] SUHENDRA, V., AND MITRA, T. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In *Proceedings of DAC* (New York, USA, 2008).

[19] THEILING, H., FERDINAND, C., AND WILHELM, R. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Syst. 18*, 2-3 (2000).

[20] UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, 2008.

[21] V. ZIVOJNOVIC, J. MARTINEZ, C. S., AND MEYR, H. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of ICSPAT* (Dallas, TX, USA, 1994).

[22] VERA, X., LISPER, B., AND XUE, J. Data Caches in Multitasking Hard Real-Time Systems. In *Proceedings of RTSS* (Cancun, Mexico, 2003).