WCET-aware Code Generation and Communication Optimization for Parallelizing Compilers

Simon Reder
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
simon.reder@kit.edu

Jürgen Becker
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
becker@kit.edu

***Abstract*—High performance demands of present and future embedded applications increase the need for multi-core processors in hard real-time systems. Challenges in static multi-core WCET-analysis and the more complex design of parallel software, however, oppose the adoption of multi-core processors in that area. Automated parallelization is a promising approach to solve these issues, but specialized solutions are required to preserve static analyzability. With a WCET-aware parallelizing transformation, this work presents a novel solution for an important building block of a real-time capable parallelizing compiler. The approach includes a technique to optimize communication and synchronization in the parallelized program and supports complex memory hierarchies consisting of both shared and core-private memory segments. In an experiment with four different applications, the parallelization improved the WCET by up to factor 3.2 on 4 cores. The studied optimization technique and the support for shared memories significantly contribute to these results.**

***Index Terms*—real-time, multi-core, compiler tools, WCET**

I. Introduction

The computational performance of single-core processors is limited by physical constraints for structure sizes, power dissipation, and clock frequency. Multi-core processors solve these limitations but suffer from a significantly higher complexity in developing appropriate parallel software. Additional challenges arise in hard real-time systems, where a safe upper bound for the *Worst-Case Execution Time* (WCET) must be determined using *static analysis*. In the worst case, a processor core can heavily influence the execution times of other cores, e.g., when multiple concurrent memory accesses collide at a shared hardware resource. This so-called *timing interference* is hard to predict at design time, which makes it difficult to compute tight upper bounds for the WCET of parallel programs.

A possible solution to the problem of timing interference is to enforce *temporal isolation*, which prevents interference effects either in hardware or in software. On the hardware side, temporal isolation can be achieved with arbitration schemes like TDMA. On the software side, the parallel program can be constrained to enforce a strict temporal separation of read/write phases to shared memories. While TDMA creates additional hardware-bottlenecks and thus is not commonly used in commercial off-the-shelf hardware, the software-based methods limit the amount of parallelism that can be exploited. Regarding mixed-criticality systems, both software-based and hardware-based approaches can result in poor average-case performance.

An alternative to temporal isolation is the estimation of interference delays using advanced static analysis techniques. This approach allows for more parallelism but requires a static analysis of the concurrent program. The latter has been proven NP-hard or even undecidable for more general synchronization models [1]. Consequently, the synchronization structure of parallel programs must be sufficiently restricted to limit the amount of possible rendezvous/synchronization scenarios. Developing such programs, however, is tedious and error-prone, which is why automated parallel code-generation tools are highly desirable. While considerable previous works investigated the underlying problem of automatic parallelization, there are only a few that address hard real-time software and generate sufficiently restricted code to enable static interference analysis.

In this work, we aim for a parallelization that supports bare-metal hardware platforms and static WCET analysis without requiring temporal isolation. In absence of (potentially unpredictable) runtime-schedulers, parallelizing compilers typically consist of the following three steps: I) *extraction of parallel tasks* form the sequential input program, II) a *static task-scheduler* and III) a *transformation from a sequential into a parallel program representation*. In this paper, we focus on step III and present a WCET-aware parallel code generator, which optimizes both data handling and synchronization structure of parallel programs towards reduced WCET bounds. In this context, our main contributions are:

1) A *WCET-aware source-to-source compiler flow* to transform an input program in the programming language C into predictable multi-core C code
2) A *code-level optimization method for communication and synchronization* in parallelized programs based on a predictable FIFO communication model
3) The *automated handling of shared variables* for mixed message-based and shared-memory communication

II. Related Work

Considerable previous works mainly focus on the scheduling problem (step II) for parallel real-time software. The approach presented in [2] is one of several examples that aim for interference-free parallel schedules. The works [3], [4] go one

978-3-9819263-4-7/DATE20/©2020 EDAA

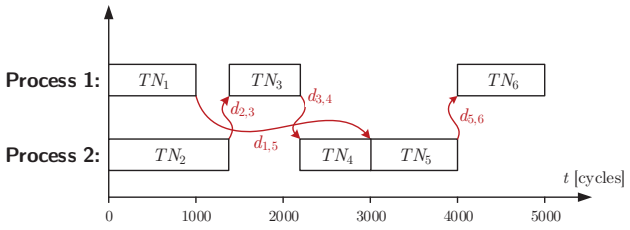
step further and account for interference in the optimization model. This allows them to explore the trade-off between parallelism and interference costs. However, the input programs for these approaches must be represented as a directed acyclic graph (DAG), which typically prevents the parallelization of loop nests with loop-carried dependencies.

Automatic parallelization of embedded algorithms with cyclic dependencies is a more general problem that is addressed in the related works [5]–[10]. From these works, only ARGO [5] and parMERASA [9] address WCET-aware parallelization of hard real-time applications. In [9], however, the code generation step is not fully-automated as the code must be fitted into algorithm skeletons by hand. To extract parallelism from sequential programs and generate a schedule (step I & II), ALMA [6], ARGO [5], and the work of Cordes et al. [7] use the *Hierarchical Task Graph* (HTG) proposed by Girkar et al. [11]. The HTG can resolve the problem of cyclic dependencies and expose parallelism in loop nests using a tree of DAGs. Although the HTG has been successfully applied to real-world programs in [5], [7], generating multiple interrelated DAG schedules remains a complex optimization problem. The complexity increases further when interference costs are included in the scheduling stage. This is, e.g., reflected by the considerable ILP solving times reported by Rouxel et al. [3] for relatively small DAGs. Results like these confirm the intuition that the problem of automated parallelization is hard to efficiently solve within a single optimization model. Tools like ALMA [6], MAPS [8] or ARGO [5] therefore rely on complex compiler flows with multiple optimization stages.

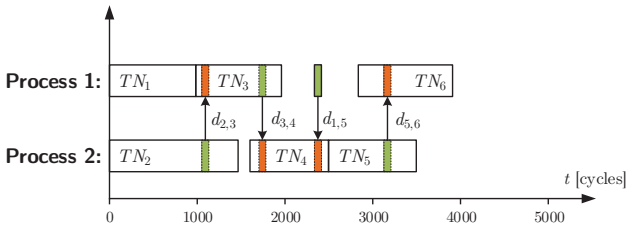
The parallelizing transformation of the input code (step III) is a possible place to include an additional fine-grained optimization stage after the scheduling phase. For this step, [6] proposes optimizations of memory allocation and communication API calls, while [8] applies a set of generic and platform-specific transformations. ARGO [5] is the only previous work to propose a concept for fine-grained re-positioning and clustering of synchronization operations, but no details or appropriate algorithms have been presented yet.

In this work, we introduce a compiler flow with post-schedule optimizations for hard real-time applications based on the concepts presented in [5]. The prerequisites for interference analysis and parallelization of loop nests require a specialized code-generation approach that guarantees static analyzability by construction. We support both shared-memory for efficient exchange of large data fields, and message-based communication to transfer smaller amounts of data between local scratchpad memories while avoiding interference at the main memory.

In contrast to our approach, the mentioned related works, except for [4], [5], do not address the static analyzability of the generated code in the presence of interference effects. Different from [4], we do not require a DAG-based program structure but target a more general execution model. To the best of our knowledge, there is no previous work reporting details on a WCET-aware parallelizing transformation that aims for static analyzability, supports loop-carried dependencies, and optimizes synchronization operations on code-level.



(a) Schedule



(b) Realization

Fig. 1: Example schedule and a possible realization

III. Problem Definition

To ensure predictable synchronization delays and interference effects, most design decisions (e.g. synchronization points & memory allocation) should be determined at compile-time. Our approach is to generate a deterministic parallel program graph [12], which exclusively uses FIFO channels and a synchronous-data-flow pattern for inter-core synchronization. Furthermore, we require data transmitted by a FIFO-based *send* operation to be received by exactly one *receive* operation, which must be known at compile-time and share the same iteration domain. This is required to restrict the program to a deterministic set of rendezvous points. To synchronize shared memory accesses, we introduce the synchronization operations *signal* and *wait*, which internally use FIFO channels as well. Since communication and synchronization both map to FIFO operations, they can in most cases be treated analogously. Due to limited FIFO sizes, blocking operations can occur on both the sender and receiver side, such that pairs of *send* and *receive* operations must be able to run in parallel. Altogether, these restrictions allow a subsequent multi-core WCET analysis to predict synchronization delays and to determine code snippets that may run in parallel and thus interfere with each other.

The input schedule is assumed to use a decomposition of the source code into a Hierarchical Task Graph (HTG), where each task node TN_i corresponds to a defined code snippet. The HTG encapsulates loops in separate hierarchy levels, such that each level can be represented as an acyclic graph $TG = (V, E, w)$. The task nodes $TN_i \in V$ are connected by directed dependency edges $d_{i,j} = (TN_i, TN_j) \in E$ and have a weight $w(TN_i)$, which represents their approximated *a priori* WCET. An example schedule for such a task graph is depicted in Figure 1a. The schedule can be described by a mapping function $\mu : V \mapsto \mathcal{P}$ and a start-time function $\sigma : V \mapsto \mathbb{N}_0$, where $\mathcal{P} = \{P_1, P_2, \dots\}$ is a set of parallel processes.

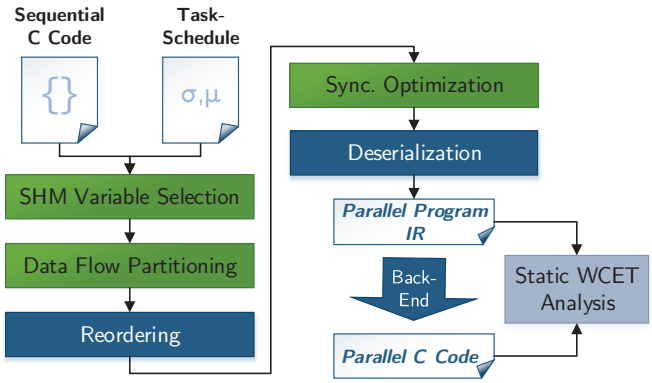


Fig. 2: Steps of the parallelizing compiler flow

To keep the problem sizes manageable, scheduling algorithms typically consider tasks as black boxes and dependencies as precedence constraints for the order of the task nodes. Such schedules can contain open degrees of freedom, which need to be determined in an appropriate realization. This especially includes the insertion of synchronization and communication at suitable positions, while preserving the order of code snippets as defined by the dependencies. Synchronization/communication for the edge $d_{1,5}$ in Figure 1 can e.g. be placed anywhere in the range from 1000 to 3000 cycles. If the internal structure of task nodes is taken into account, it might to a certain degree be even moved to the inside of task nodes. This way, the example schedule could be realized as shown in Figure 1b, which shortens the critical path by 1000 cycles. Another optimization is possible for the case that $d_{1,5}$ and $d_{3,4}$ do not require communication, e.g. because they just synchronize writes to shared memory variables. In that case, $d_{1,5}$ would be redundant and could be merged with $d_{3,4}$, because the latter already ensures that TN_5 starts after TN_1.

Our compiler flow exploits the named degrees of freedom with a code-level optimization method. In contrast to optimizations on schedule-level, this approach is not restricted to task boundaries, linear task sequences or a single acyclic HTG level.

IV. Parallelizing Compiler Flow

The individual steps of our program transformation flow are depicted in Figure 2. The basic idea is that each step creates a consistent intermediate representation, which can be used to generate functionally correct C code. This property improves the testability, which can be an advantage with regards to a potential tool qualification. A static multi-core WCET analyzer can use the parallel intermediate representation and a compiled binary of the C code to derive a safe WCET bound.

In the first step of the flow, we decide for all larger variables in the program, whether they should be handled with shared memory semantics or with multiple core-private copies and message-based communication. This step can take complex memory hierarchies consisting of several shared and core private memory segments with different sizes into account. The required information about the target platform is provided

```
float signal[1024];
cplx_float out1[512], out2[512], result[1024];
int N = get_sample_count();                  // process 0
get_signal(signal);                          // process 0
fft(signal, 0, N/2, out1);                   // process 1
fft(signal, N/2, N/2, out2);                 // process 0
combine_fft(out1, out2, N_p0, result);       // process 0
```

Data Flow Partitionning

```
float signal[1024];
cplx_float out1[512], out2[512], result[1024];
int N_p0 = get_sample_count();               // process 0
int N_p1;
comm_copy(N_p0, N_p1);
get_signal(signal);                          // process 0
sync_point(0, 1);
fft(signal, 0, N_p1/2, out1);                // process 1
sync_point(1, 0);
fft(signal, N_p0/2, N_p0/2, out2);           // process 0
combine_fft(out1, out2, N_p0, result);       // process 0
```

Reorder & optimize Sync.

```
float signal[1024];
cplx_float out1[512], out2[512], result[1024];
int N_p0 = get_sample_count();               // process 0
int N_p1;
get_signal(signal);                          // process 0
comm_copy(N_p0, N_p1);
fft(signal, 0, N_p1/2, out1);                // process 1
fft(signal, N_p0/2, N_p0/2, out2);           // process 0
sync_point(1, 0);
combine_fft(out1, out2, N_p0, result);       // process 0
```

Fig. 3: Input code and intermediate outputs for an example program running through the transformation flow

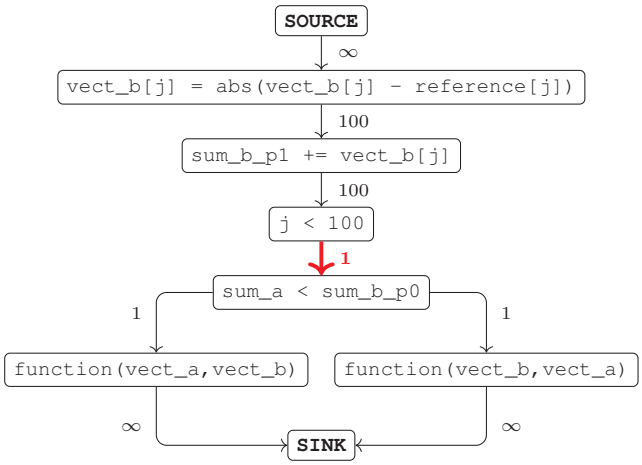
by an architecture description format. To get a feasible and optimized decision, we use an extension of the ILP-based memory allocation approach presented by Avissar et al. in [13]. Based on the core mapping $\mu(TN_i)$ of the memory accesses in the program, we enhance the ILP model to consider the tradeoff between multiple core-private variable copies versus a single copy in a potentially slower shared memory.

After a set of shared variables has been selected, the *Data Flow Partitioning* step creates an intermediate representation, which still has a sequential control flow, but separated data fields per process. The output of this transformation is illustrated in Figure 3 for a simple example program, which computes a Fast Fourier Transformation (FFT). The FFT algorithm can be parallelized using the fact, that each FFT can be represented as a sum of two smaller FFTs, which are weighted with complex twiddle factors. In the simplified example, this sum is computed by the function `combine_fft` and the selected task-mappings are annotated as comments. We furthermore assume that the variables `signal` and `out2` have been selected as shared variables, such that only the variable `N` is subdivided into core-private duplicates. To keep the program representation consistent, the data flow partitioning step inserts `comm_copy` operations as placeholders for later communication as well as `sync_point` calls to represent later shared-memory synchronization.

The purpose of the subsequent *Reordering* step is to sort the blocks in the sequential control flow according to their start

```
double sum_a = 0, sum_b_p0 = 0, sum_b_p1 = 0;
// ------ process 0 ------
for (int i = 0; i < 100; ++i) {
    vect_a[i] = abs(vect_a[i] - reference[i]);
    sum_a += vect_a[i];
}
// ------ process 1 ------
for (int j = 0; j < 100; ++j) {
    vect_b[j] = abs(vect_b[j] - reference[j]);
    sync_point(1, 0);
    sum_b_p1 += vect_b[j];
    comm_copy(sum_b_p1, sum_b_p0);
}
// ------ process 0 ------
if (sum_a < sum_b_p0) {
    function(vect_a,vect_b);
} else {
    function(vect_b,vect_a);
}
```

(a) Example code after data flow partitioning



(b) Minimum edge cut problem for the `sync_point` in Fig. 4a

Fig. 4: Example code and one of the resulting edge-cut problems

times $\sigma(TN_i)$ in the schedule. With the control flow in its final order, the *Synchronization Optimization* step can optimize the placement of the placeholder operations `sync_point` and `comm_copy` with respect to the goals described in Section III. The corresponding algorithm will be discussed in Section V. For the example in Figure 3, it might decide to merge the first two redundant operations and to re-position the last `sync_point`, which enables parallel execution of the two smaller FFTs.

The final *deserialization* step splits the sequential control flow into separate processes $P_k \in \mathcal{P}$ based on the mapping function $\mu(TN_i)$. Placeholder operations are replaced in the respective processes by send/receive for `comm_copy` and signal/wait for `sync_point` operations. Applying the deserialization as the last step of the flow ensures a defined order of FIFO operations with equal iteration domains and guarantees that the parallel program is serializable. Serializability, in turn, can be proven to imply freedom of deadlocks [12].

V. Synchronization Placement

The synchronization placement problem in Figure 3 is relatively simple due to the linear control flow. To allow movement and merging of placeholder operations in the general case, loops and branches must be considered appropriately. In the following, we present a heuristic approach to realize this optimization in an arbitrary control flow graph $CFG = (V, E^{cf})$.

A. Individual Placement

For each synchronization/communication placeholder $s \in \mathcal{S}$, we define a subset $VP(s) \subset E^{cf}$ of control flow edges, to which the operation can possibly be moved. In general, a synchronization operation with source process P_i and target process P_j has the purpose to ensure the completion of a set of blocks $\mathcal{B}^{src}(s)$ mapped to P_i before starting the execution of any block in a second set $\mathcal{B}^{dest}(s)$ mapped to P_j. When moving the operation s within the CFG, it must be guaranteed that all control flow paths leading from any block in $\mathcal{B}^{src}(s)$ to any block in $\mathcal{B}^{dest}(s)$ contain an instance of s. Based on this requirement, the set $VP(s)$ of valid positions for s can be defined as the union of all control edges on any of the paths between $a \in \mathcal{B}^{src}(s)$ and $b \in \mathcal{B}^{dest}(s)$.

To compute the best position(s) for a single synchronization operation s, we construct a minimum edge-cut problem based on the CFG sub-graph with all edges $e \in VP(s)$. Figure 4 illustrates that for a code example with two synchronization operations. The `sync_point` (s_0) is used to ensure that the shared variable `vect_b` is not read before completing the assignment, while the `comm_copy` operation (s_1) transfers the value of the variable `sum_b_p1`. The initial placement is suboptimal since both synchronizations take place in a loop with 100 iterations. The minimum-cut problem for the `sync_point` operation s_0 is depicted in Figure 4b and uses the iteration count of the CFG edges as edge weight. It furthermore contains a source node, which is adjacent to all blocks in $\mathcal{B}^{src}(s_0)$, and a sink node being adjacent to the blocks in $\mathcal{B}^{dest}(s_0)$. Infinite edge weights prevent the respective edges from being part of the solution. To find the position(s) with the lowest iteration count, we can compute the minimum s-t cut between source and sink using the Edmonds-Karp algorithm. With the result, we place an instance of the synchronization operation at every edge that is part of the minimum cut. The weight of the cut represents the total costs for the solution, which would be 1 for the example in Figure 4.

B. Edge Weights

The example in Figure 4 assumes iteration counts as edge weights, which is insufficient to minimize synchronization delays. Our algorithm therefore uses combined weights based on iteration counts and the worst-case arrival times $WCAT(e, P_i)$ for the considered edge $e \in E^{cf}$ and every process $P_i \in \mathcal{P}$. These times are defined by the worst-case time to reach the edge e from the start node of the CFG while ignoring blocks that are not mapped to P_i. The path lengths are computed using an approximated breakdown of sequential WCET times to individual CFG blocks. The overall edge weight $w^{cut}(e, s)$

for a given synchronization operation s between the processes P_i and P_j is then computed as follows:

$$w^{cut}(e,s) := I(e) \cdot \big(a \cdot n^{bytes}(s) + b\big) + \big|\, WCAT(e,P_j) - WCAT(e,P_i) \big|$$

In this equation, $I(e)$ stands for the worst-case iteration count of e, while $n^{bytes}(s)$ represents the number of bytes to be transferred on the FIFO channel. The parameters a and b describe the overhead introduced by the actual implementation of the FIFO operations.

C. Joint Placement

Merging redundant synchronization requires the corresponding operations to be moved to the same position and to have an equivalent effect. To meet the former condition, we extend our minimum-cut method to find cost efficient common positions for a set of multiple synchronization operations $\{s_0 \dots s_n\}$. This is done by defining a joint set of valid positions $VP(\{s_0, \dots, s_k\}) := VP(s_0) \cup \dots \cup VP(s_k)$ as well as two joint sets of blocks $\mathcal{B}^{src}(\{s_0, \dots, s_k\})$ and $\mathcal{B}^{dest}(\{s_0, \dots, s_k\})$, which are constructed in the same way. For the example in Figure 4, such a joint placement could move both placeholder operations to the highlighted edge, where they would be merged into a single `comm_copy` operation.

A joint placement only makes sense, if two potentially redundant synchronization operations s_0 and s_1 have at least one position in common: $VP(s_0) \cap VP(s_1) \neq \emptyset$. If this precondition is met, joint placements could still be more costly, e.g. when some advantageous positions become infeasible due to additional parallel edges with infinite weight. The problem of finding a beneficiary set of joint solutions can be formulated as graph clustering problem with respect to the so-called joint solution graph $J = (V^{joint}, E^{joint})$. The vertices $v \in V^{joint}$ each represent a subset of synchronization operations $v \subset \mathcal{S}$ that are jointly placed at common positions. The undirected edges $\{v_i, v_j\} \in E^{joint}$ then represent possibilities to generate a joint placement for the union $v_i \cup v_j$ of all operations in both vertices. An edge $\{v_i, v_j\}$ consequently exists for potentially redundant sets of synchronizations with $VP(v_i) \cap VP(v_j) \neq \emptyset$.

To explore the joint solution space, we propose the greedy heuristic shown in Algorithm 1. The idea is to start with an initial joint solution graph J that contains individual placements only (i.e. $\forall v_i \in V^{joint} : |v_i| = 1$). If two synchronization operations can be merged, they are connected by an edge, which makes them part of the same connected component. The algorithm iterates over these connected components sorted by their average topological order in the CFG. This order ensures, that changes to the edge arrival times $WCAT$ caused by the insertion of earlier synchronization can be taken into account when computing optimized positions for later synchronization. To achieve that, the algorithm repeatedly updates the edge arrival times before processing the next connected component.

The actual clustering of adjacent vertices in J is done in an iterative greedy algorithm. For each edge $\{v_i, v_j\} \in E^{joint}$, we define the gain $\textsc{Gain}(\{v_i, v_j\})$ as the sum of costs for the individual placement solutions of v_i and v_j minus the costs of the joint placement for $v_i \cup v_j$. Based on that, we search the edge with the highest gain and cluster the connected vertices into a single one. This process is iteratively repeated until there are no more edges with positive gain in the current connected component.

Algorithm 1 Synchronization placement and clustering

Input: $CFG = (V, E^{cf})$ and $\mathcal{S}$
Output: Synchronization positions $POS : \mathcal{S} \mapsto \mathbb{P}(E^{cf})$

```
POS(s) ← ∅,   ∀ s ∈ S
V^joint ← { v ⊂ S : |v| = 1 }
E^joint ← FindCombinableVertexPairs(V^joint)
J ← (V^joint, E^joint)
CP ← ListConnectedVertexSets(J)
SortInCfgOrder(CP)
for i = 0 to |CP| − 1 do
    UpdateEdgeArrivalTimes(CFG, POS)
    C ← CP[i]
    ComputeMinCuts(C)
    e^max ← FindEdgeWithMaxGain(C, E^joint)
    while Gain(e^max) > 0 do
        (J, C) ← ClusterAdjacentVertices(e^max, J, C)
        e^max ← FindEdgeWithMaxGain(C, E^joint)
    end while
    for all v ∈ C do
        for all s ∈ v do
            POS(s) ← GetMinCutEdges(v)
        end for
    end for
end for
```

VI. Experimental Results

We evaluate the presented parallelizing transformation using the four benchmark applications in Table I. The Canny+Hough as well as the k-means benchmark have been prepared for parallelization by splitting larger arrays into smaller parts that can be processed independently. Schedules for the HTG representation were generated using the WCET-aware HEFT-LA heuristic from [10]. As hardware target, we use a variant of the predictable tile-based research platform presented in [5]. It consists of a 2x2 Network-on-Chip (NoC) in mesh topology with 4 tiles that each contain a Leon3 processor core and 512 Kbyte of private on-chip memory. One of the tiles hosts a larger off-chip shared memory with a size of 2GB. The system is implemented as FPGA prototype and runs with 100MHz. Our main evaluation criterion is the speedup when comparing the WCET of the generated parallel program to the WCET of the sequential input code on a single Leon3 core. The latter is determined with the commercial WCET analyzer aiT, while the former is a result of the ARGO multi-core WCET estimator [5], which internally uses aiT as well.

Figure 5 shows the WCET speedups achieved using 4-core parallelizations of the benchmarks with different combinations of shared memory and synchronization placement optimizations. For a fair comparison, the reference case with disabled synchronization refinement did not use inefficient placements like in Figure 4 but inserted the operations in the outermost possible loop nest that surrounds the source HTG-task.

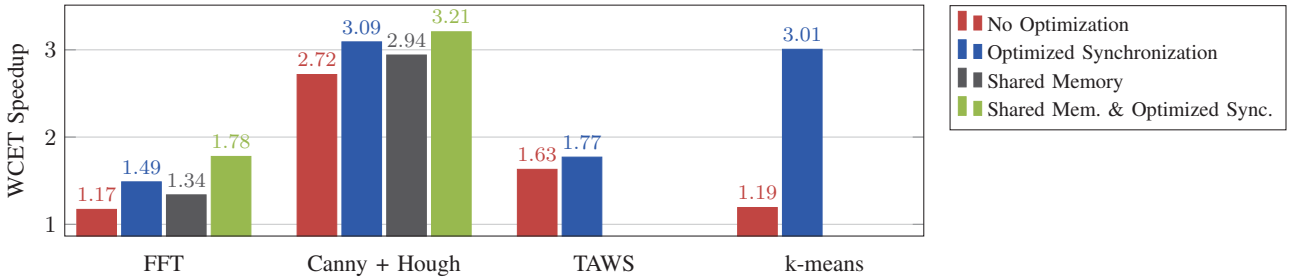


Fig. 5: WCET speedup results for different benchmark applications and optimization scenarios

TABLE I: Benchmark applications

Benchmark	Seq. WCET [cycles]	Description
FFT	7.699×10^8	32 768-point Fast Fourier Transform using the Cooley-Tukey algorithm
Canny + Hough	1.432×10^{11}	512x512 pixel Canny Edge Detector followed by a Hough-Transformation
TAWS	7.788×10^5	Terrain Awareness and Warning System from avionics domain
k-means	1.152×10^9	k-means clustering algorithm using 4096 3D vectors and 4 means

In all benchmarks, the optimized synchronization placement was able to significantly improve the WCET speedups. The largest impact can be observed for the k-means algorithm, where the unoptimized placement prevents parallel execution of major program parts. The clustering of redundant synchronization was especially effective for the FFT with shared memory, where it could reduce the number of FIFO operations from 78 to 25. The FFT also benefits the most from shared memory usage, which can be explained by the high costs of transmitting all transformed signals via message-based communication. For size reasons, most of the processed signal slices would anyway go to the off-chip memory, which makes data sharing even more beneficiary. In contrast to that, the tool-flow did not select any shared memory variables for the TAWS and k-means algorithms, because the faster on-chip memories were sufficient to store all data.

Overall, the presented approach could successfully speed up the WCET of all four benchmarks, while the two studied optimizations significantly contributed to these results.

VII. Conclusion

In this paper, we presented a WCET-aware compiler flow to generate well-predictable parallel C code from a scheduled sequential input program. The flow enables the efficient use of shared memories in platforms with mixed private and shared memory segments. A dedicated optimization step improves the positions of synchronization and communication operation on a fine-grained level and is able to remove redundant synchronization. Our experimental evaluation shows, that both shared memory usage and refined synchronization can significantly improve the WCET-speedup of four different practically relevant algorithms.

Altogether, our flow provides an essential building block for WCET-aware parallelizing compilers, which can help to enable design automation in real-time software development for multi-core systems. As a future work, we plan to evaluate the approach for a wider range of target platforms and to further investigate its potential to facilitate the use of multi-core processors in future hard real-time systems.

References

[1] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 416–430, Mar. 2000.

[2] J. Matějka *et al.*, "Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution," in *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'18. New York, NY, USA: ACM, 2018, pp. 11–20.

[3] B. Rouxel *et al.*, "Tightening contention delays while scheduling parallel applications on multi-core architectures," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 164:1–164:20, Sep. 2017.

[4] K. Didier *et al.*, "Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 24:1–24:27, Jul. 2019.

[5] S. Reder *et al.*, "Worst-case execution-time-aware parallelization of model-based avionics applications," *Journal of Aerospace Information Systems*, vol. 16, no. 11, pp. 521–533, 2019.

[6] T. Stripf *et al.*, "Compiling scilab to high performance embedded multicore systems," *Microprocessors and Microsystems*, vol. 37, no. 8, Part C, pp. 1033 – 1049, 2013, special Issue on European Projects in Embedded System Design: EPESD2012.

[7] D. Cordes *et al.*, "Automatic extraction of task-level parallelism for heterogeneous mpsocs," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 950–959.

[8] R. Leupers *et al.*, "Maps: A software development environment for embedded multicore applications," *Handbook of Hardware/Software Codesign*, pp. 917–949, 2017.

[9] T. Ungerer *et al.*, "Experiences and results of parallelisation of industrial hard real-time applications for the parmerasa multi-core," in *Submitted to the 3nd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands*, 2015.

[10] P. Alefragis *et al.*, "Mapping and scheduling hard real time applications on multicore systems - the argo approach," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Cham: Springer International Publishing, 2018, pp. 700–711.

[11] Milind Girkar and Constantine D. Polychronopoulos, "The hierarchical task graph as a universal intermediate representation," *International Journal of Parallel Programming*, vol. 22, no. 5, pp. 519–551, oct 1994.

[12] V. Sarkar and B. Simons, "Parallel program graphs and their classification," in *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 1994, pp. 633–655.

[13] O. Avissar *et al.*, "Heterogeneous memory management for embedded systems," in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '01. New York, NY, USA: ACM, 2001, pp. 34–43.