Static Allocation of Basic Blocks Based on Runtime and Memory Requirements in Embedded Real-Time Systems with Hierarchical Memory Layout

Philipp Jungklass
IAV GmbH, Gifhorn, Germany
philipp.jungklass@iav.de

Mladen Berekovic
Universität zu Lübeck, Institute of Computer Engineering, Germany
berekovic@iti.uni-luebeck.de

Abstract

Modern microcontrollers for safety-critical real-time systems use a hierarchical memory system to increase execution speed and memory capacity. For this purpose, flash memories, which offer high capacity at low transfer rates, are combined with scratchpad memories, which provide high access speed at low memory capacities. The main goal is to use both types of memory in such a way that their advantages are optimally exploited. The target is to allocate runtime-intensive code fragments with low memory requirements to the fast scratchpad memories. Previous approaches to separate program code on system memories consider the executed functions as the smallest logical unit. This is contradicted by the fact that not all parts of a function have the same computing time in relation to their memory usage. This article introduces a procedure that automatically analyses the compiled source code and identifies runtime intensive fragments. For this purpose, the translated code is executed in an offline simulator and the maximum repetition for each instruction is detected. This information is used to create logical code fragments called basic blocks. This is repeated for all functions in the overall system. During the analysis of the functions, the dependencies between them are also extracted and a corresponding call-graph with the call frequencies is generated. By combining the information from the call graph and the evaluation of the basic blocks, a prognosis of the computing load of the respective code blocks is created, which serves as base for the distribution into the fast scratchpad memories. To verify the described procedure, EEMBC's CoreMark is executed on an Infineon AURIX TC29x microcontroller, in which different scratchpad sizes are simulated. It is demonstrated that the allocation of basic blocks scales significantly better with smaller memory sizes than the previous function-based approach.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture

Keywords and phrases Memory Architecture, Memory Management, Real-time Systems

Digital Object Identifier 10.4230/OASIcs.NG-RES.2021.3

Acknowledgements Special thanks to Arne Bredemeier from Infineon.

1 Introduction

In modern systems with hard real-time requirements, microcontrollers are increasingly used, which implement a hierarchical memory layout with different memory technologies. The reason for this is that memories with a large capacity, such as flash, have a comparatively slow access time and a significantly reduced bandwidth in contrast to fast scratchpad memories based on Static Random-Access Memory (SRAM). In comparison, the disadvantage of SRAM is that it requires a lot of space on the wafer during manufacturing, which significantly

increases production costs. For this reason, the manufacturers of such microcontrollers try to combine these two types of memory efficiently with each other, thus exploiting the advantages of both technologies and reducing the disadvantages [13] [14].

Table 1 Relation of the memory sizes of microcontrollers for real-time systems [7] [15] [18] .

Microcontroller	**Flash-Memory / KB**	**SRAM-Memory / KB**
Infineon AURIX TC29x (3 processor cores)	Code: 8192 Data: 1024	Code: 96 Data: 600
Texas Instruments TMS320F2838x (5 processor cores)	Code: 1536 Data: -	Code: - Data: -
ST SPC58 family (3 processor cores)	Code: 6144 Data: 256	Code: 48 Data: 160

As table 1 shows, the storage capacity of SRAM-based memories is significantly smaller compared to flash memories. It should also be noted that all specifications refer to the respective overall system. Therefore it is important to note that, for example, the Infineon AURIX TC29x provides a relatively large SRAM memory of 96 KB for code execution, but this is divided among the three processor cores, so that only 32 KB is directly available for each core [7]. The great potential of these small but powerful memories is that the fast access times significantly increase the execution speed. In case of the already mentioned Infineon AURIX TC29x, code from the local SRAM memory of the corresponding processor core is executed 3.5 times faster than using the flash memory. Furthermore, by allocating local copies of selected code parts to the respective SRAM memory of the cores, the number of concurrent accesses can be reduced significantly, which is particularly relevant for real-time multicore systems [16] [12] [8]. Due to the limited memory capacity it is essential to evaluate which code fragments should be allocated in these memories. As these code fragments are needed more frequently, the increased execution speed has a greater influence on the performance of the overall system. For this reason, this article presents a procedure that analyzes the code at instruction level and allocates particularly runtime-intensive basic blocks to the fast scratchpad memories. For this purpose, the call frequency is set in relation to the memory consumption, with the goal of an optimized usage of the fast memory. The structure of this article is divided into six chapters. After the introduction, related work as well as the previous optimization approaches are presented. This is followed by a description of the developed concept, which is implemented practically in the fourth section. To prove the functionality, the presented method is applied to the CoreMark of EEMBC in the fifth chapter. Finally, the results are discussed and an outlook on future extensions is given.

2 Related Work

Previous work on optimizing the use of existing scratchpad memory can be divided fundamentally into two categories. In static distribution, the allocation of code and data takes place during development and is fixed at system runtime. The advantage of this approach is that a static allocation can more easily be mapped in a timing model. This is necessary for the calculation of the Worst-Case Execution Time (WCET), which is mandatory for the certification of a safety-critical real-time system. The disadvantage, in contrast, is the sometimes suboptimal utilization of the small scratchpad memory, which cannot realize its speed advantage optimally, depending on the execution path. Dynamic approaches try to

compensate this by adapting the contents of the scratchpad memory at runtime according to the program flow. It should be noted, that dynamic approaches cause an overhead due to copy processes and the associated administration, that should not be underestimated.

2.1 Static Allocation

In [10] a procedure for the static distribution of code and data into the available memory of a real-time multicore system is shown. For this purpose priorities are calculated for all functions and variables in the system, which result from the call frequency as well as the number of concurrent accesses. Depending on the priority, the code and data are distributed to the memories in the system and local copies are created if necessary. In contrast to this work, in the article, functions are defined as the smallest allocatable unit, whereby less frequently executed code paths are also allocated to the scratchpad memory. A further approach of static separation is discussed in [2]. The article describes a compiler strategy for distributing the stack and global variables into the Random-Access Memory (RAM) of the microcontroller. As boundary conditions for the procedure the renouncement of an Memory Management Unit (MMU) as well as the use of a Non-Uniform Memory Access (NUMA)-based memory management is demanded. Analogous to the method presented in this article, the execution speed shall be improved by an optimized memory distribution depending on the call frequency. In contrast, the analysis is limited to global variables and the separation of the stack. In the article [9] and [11] approaches for the optimization of explicit intercore communication in multicore systems are presented. The concepts are based on the assumption that shared variables should be allocated to the local scratchpad memory of those processor cores that access it more often. The two methods have a similar approach, but differ in their respective implementation. In contrast to the procedure described here, the concepts in the two articles are limited to the allocation of variables for intercore communication.

2.2 Dynamic Allocation

One possibility for the dynamic use of scratchpad memory is described in the approach [13]. For the presented procedure the functions are divided into basic blocks in the first step. In the second step, the basic blocks are determined, which are called in the Worst-Case Execution Path (WCEP). Then these basic blocks are allocated to the fast scratchpad memories, which increases the execution speed. In contrast to the approach in this article, the described concept uses dynamic memory management, which reloads the basic blocks as required. As a result, there is no evaluation or prioritisation of functions among themselves, which would be necessary for static memory management. Instead, an offline evaluation of the basic blocks is carried out to determine if they are qualified for reloading at runtime, because the copy time causes a significant overhead, which has to be compensated by the increased execution speed. In [19] a method is described which optimizes concurrent accesses to shared memory in real-time capable multicore microcontrollers. For this purpose the scheduling of the operating system is extended, whereby a task consists of three phases. In the first phase, all required data is loaded from the shared memory into the local caches of the processor cores before it is executed in the second phase. In the last phase, the generated results are copied back into shared memory. Although the procedure in this article focuses on caches, this method is also possible when using scratchpad memories. In contrast to the concept in this article, dynamic memory management is implemented here to reduce competing accesses. However, in order to reduce the number of concurrent accesses, all possible paths of a task must be completely copied to the local memory. Only in this case copying phases during execution

can be effectively prevented. However, this also copies paths of a task whose computing time has no significant influence on the runtime. The method presented in the article [4] describes a procedure that dynamically allocates basic blocks into scratchpad memory based on their call frequency. To reduce the overhead of dynamic memory management, basic blocks with a fixed size are used, which can compensate the problem of fragmentation. The problem is, fixed-size memory blocks are not always completely filled, which in effect reduces the efficiency of memory usage.

3 Concept

The concept described here is to identify code fragments within a function that have a high runtime and low memory requirements. For this purpose, the translated code is analyzed in an offline simulator and the minimum and maximum execution frequency is determined for each instruction. In addition, the memory requirements of each instruction and its execution time in clock cycles are extracted from the processor architecture description. This information can be used to create a table for each function, which contains all the data required to evaluate each instruction. The goal is to allocate the particularly computationally time-intensive areas of a function to the fast scratchpad memories of the microcontroller, which allows the low capacity of these high-performance memories to be used more efficiently. The procedure is illustrated using listing 1 as an example.

```
1  /* function to copy the message from
2  receive buffer in destination buffer */
3  uint32 CopyMessage(uint8* dest, uint32 length)
4  {
5    /* return value */
6    uint32 errorCode = TRUE; /* TRUE = 1 */
7
8    /* check the maximum buffer length */
9    if(length <= 32)
10   {
11     for(uint32 i = 0; i < length; i++)
12       {
13       /* copy the received byte from receive
14         buffer (rb) into destination buffer */
15       *dest++ = rb[i];
16     }
17   }
18   else
19   {
20     /* invalid length */
21     errorCode = FALSE; /* FALSE = 0 */
22   }
23   return errorCode;
24 }
25
```

Listing 1 Example CopyMessage: Source code (ANSI C)

The minimal example in the listing contains a function which copies the data from a global array `rb[]` byte by byte into a target memory, whose address is passed to the function as bytepointer `dest`. To avoid memory overflow, the requested variable `length` is checked for the maximum value of 32 before copying. Depending on the result of the check, the data is copied to the target memory or a corresponding error message is returned to the calling function. Using the C code shown here, it can already be estimated that the runtime is primarily dependent on the length of the data to be copied.

3.1 Machine Code Analysis

In the first step of the analysis, the source code for the target platform is compiled. The reason for this procedure is that modern compilers support a variety of optimizations that take into account architectural characteristics like jump predictions, superscalar architectures or errors in processor design. Therefore, depending on the configured optimization level, the translated machine code can differ massively from the original source code. To ensure that the available scratchpad memory can still be used efficiently, the translated machine code is therefore used as the basis for optimization.

In the example shown here, this is done for an Infineon AURIX TC29x, using the Tasking Compiler in version 6.2r2 with the optimization level `O0` as compiler. The result can be taken from the commented listing 2.

```
1   mov d2,#1                ;Move
2   mov d15,#32              ;Move
3   jge.u d15,d4, .L5        ;Jump if Greater Than or Equal
4   j .L17                   ;Jump Unconditional
5   mov d15,#0               ;Move
6   j .L14                   ;Jump Unconditional
7   movh.a a15,#@his(rb)     ;Move High to Address
8   lea a15,[a15]@los(rb)    ;Load Effective Address
9   addsc.a a15,a15,d15,#0;Add Scaled Index to Address
10  ld.bu d0,[a15]           ;Load Byte Unsigned
11  st.b [a4],d0             ;Store Byte
12  add.a a4,#1              ;Add Address
13  add d15,#1               ;Add
14  jge.u d15,d4,.L16        ;Jump if Greater Than or Equal
15  j .L7                    ;Jump Unconditional
16  j .L18                   ;Jump Unconditional
17  mov d2,#0                ;Move
18  j .L19                   ;Jump Unconditional
19  ret                      ;Return from Call
20
```

Listing 2 Example CopyMessage: Source code (machine code)

With the help of the Infineon TriCore Simulator (TSIM) the translated code is executed and combined with the information from the processor architecture description. The result is shown in the table 2.

As it can be seen in the table 2, the offline simulator analyzes all paths of the function and determines the minimum and maximum call frequency for each instruction. This information is combined with the memory requirements and the runtime. This combination of parameters is used to identify fragments that require a lot of computing power. Additionally, the WCEP is determined for each function. This procedure is particularly interesting for safety-critical real-time systems, because only the WCET is relevant for the evaluation of such systems.

3.2 Basic Block Prioritisation

After the identification of the relevant blocks, a prioritisation is carried out. In the first step, all basic blocks along the WCEP are evaluated regarding of their runtime. Due to the focus on real-time systems, these fragments receive the highest priority within the function. Subsequently, all other paths are examined for their optimization potential. After completion of this analysis, all basic blocks of this function have a priority based on their call frequency, with the WCEP having the highest rating. The formula (1) describes the calculation.

$$p_f(bb) = \sum_{i=0}^{i_{\max}} (e_{\max}(i) \cdot r_{\max}(i)) + p_{\text{WCEP}} \tag{1}$$

$p_f(bb)$	Priority of the basic block bb within the function f
$e_{\max}(i)$	Maximum execution freqency of instruction i
$r_{\max}(i)$	Maximum runtime of instruction i
p_{WCEP}	Additional priority for the WCEP

As the procedure presented here is a static memory allocation, a dynamic adjustment of the scratchpad contents at runtime is not possible. In fact of this, it is not sufficient to prioritise only the basic blocks within a function. It is also necessary to consider the call frequencies of the individual functions in the overall system in the evaluation. For this reason, all function jumps and their frequency are logged during the analysis of the functions and used to construct a call graph. On the basis of this information, the call frequency can be determined for each function, which is then included in the prioritisation of the basic blocks. The following formula (2) illustrates the procedure.

$$p_s(bb) = e_{\max}(f) \cdot p_f(bb) \tag{2}$$

$p_s(bb)$	Priority of the basic block bb within the system s
$e_{\max}(f)$	Maximum execution freqency of function f

Table 2 Example CopyMessage: memory usage, runtime and call frequency.

Instruction	**Memory usage /Byte**	**Runtime /Ticks**	**Execution frequency Min/Max /Ticks**
mov d2,#1	4	1	1/1
mov d15,#32	4	1	1/1
jge.u d15,d4, .L5	4	1	1/1
j .L17	4	1	0/1
mov d15,#0	4	1	0/1
j .L14	4	1	0/1
movh.a a15,#@his(rb)	4	1	0/32
lea a15,[a15]@los(rb)	4	1	0/32
addsc.a a15,a15,d15,#0	4	1	0/32
ld.bu d0,[a15]	4	1	0/32
st.b [a4],d0	4	1	0/32
add.a a4,#1	2	1	0/32
add d15,#1	2	1	0/32
jge.u d15,d4,.L16	4	1	0/33
j .L7	4	1	0/32
j .L18	4	1	0/1
mov d2,#0	4	1	0/1
j .L19	4	1	1/1
ret	4	4	1/1
Total	**72**	**22**	**10/300**

3.3 Basic Block Separation

The basic blocks are separated by jumps, which are already contained in the machine code. Only the address have to be manipulated. By this procedure both the runtime and the memory consumption can be kept constant. Ideally, most compilers generate entry and exit jumps during the translation of loops, which are suitable for this modification. In the example from listing 2 used here, the jump instructions in line 6 and line 16 are used for this purpose. Analogous to the procedure with loops, this is also done with branches, which are also implemented using conditional jump instructions. Depending on the architecture used, however, it must be taken into account that often only one branch path is reached by a jump. This becomes clear in listing 2, where in line 3 the conditional jump is executed. If the check is unsuccessful, the following line is executed, but without a jump. In fact of this, care must be taken during compilation that the runtime-intensive path is always reached via a jump, so that this can be allocated to the fast scratchpad memory if necessary. Applying the presented concept to the example shown here results in all instructions from line 7 to line 16 inclusive being allocated to the fast scratchpad memory. By this procedure the required memory can be reduced from 72 bytes, when allocating the complete function, to 36 bytes. Despite the 50% reduction in memory requirements, 290 of the 300 required instructions are executed from scratchpad memory in the case of WCEP.

4 Implementation

The offline simulator for the analysis of the translated machine code basically consists of the components shown in figure 1. As input variables, the simulator receives a description of the microcontroller, which contains the addresses of the memories, the addressing types and the instruction set. On the other hand the translated machine code is entered into the simulator. Both input files are then processed by a parser, which converts the information into an uniform format. This input parser is intended to enable easy expandability for further microcontroller architectures or compilers. In the next step, the code is analyzed using the concept described in chapter 3 and then the priority for each basic block is calculated. Using the knowledge gained, an optimized memory allocation can be calculated in the next-to-last step with the help of the allocation block. At last, a modified copy table as well as the corresponding linker script will be created using two generators.

4.1 Basic Block Separation

After the complete analysis of all functions in the overall system and their prioritisation, the jump instructions in the compiled machine code are manipulated to separate the basic blocks. In general, modern microcontrollers use an instruction set that provides a large number of such instructions. These jump instructions can be divided into two categories. The first category consists of absolute jumps, which refer directly to an address in memory. In contrast, the second variant always uses relative jumps in relation to the current address. For example, a relative jump can refer to an address that is 20 bytes further in memory. The advantage of these relative jumps is their reduced memory requirement, since no complete address has to be stored. The disadvantage is their reduced range, since it is only possible to refer to a much smaller memory area. In order to implement the method presented in this article, it must be possible to implement a jump to the scratchpad within a function allocated in the flash memory. Depending on the microcontroller used, absolute jumps are absolutely necessary for this, since the two types of memory often use different address ranges. For the Infineon AURIX TC29x used in this article, the absolute addresses for the different memories are listed in the table 3.

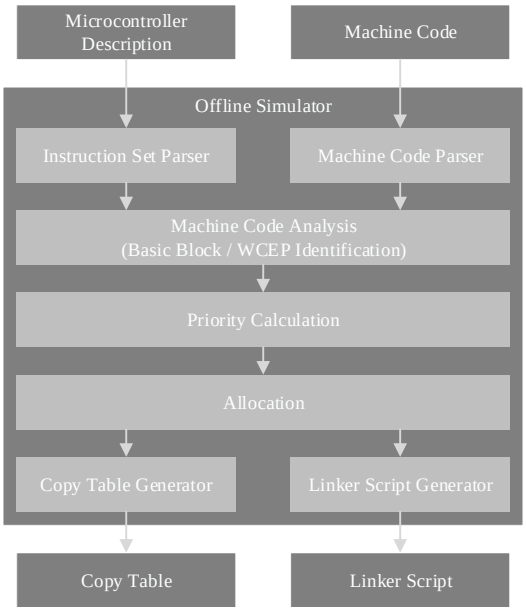

Figure 1 Offline Simulator - Detailed Design.

Table 3 Infineon AURIX TC29x - Code Memory Addresses [7].

Memory	Address Range	Size/KB
Code Scratchpad (CPU0)	0x7010.0000-0x7010.7FFF	32
Code Cache (CPU0)	0x7010.8000-0x7010.FFFF	32
Flash	0x8000.0000-0x807F.FFFF	8192

The relative jumps of the TriCore architecture of the AURIX use one byte for addressing, which allows a maximum jump width of 512 bytes with an addressing granularity of two bytes. As a result, it is only possible to switch between the flash memory and the scratchpad with an absolute jump. Due to the fact that a relative jump requires less memory space compared to an absolute jump, this creates a problem because the two types of addressing cannot simply be replaced. There are three different approaches to solve this problem, which are explained in the following sections [6].

4.1.1 Absolute Jumps

Modern compilers offer the option to avoid relative jumps in memory by various configuration settings, whereby only absolute addresses are used. The advantage of this variant is that the used addresses in the memory can be easily replaced. The disadvantage is the increased memory consumption of this method, which results from the exclusive use of absolute addresses. One way to reduce this problem is the specific use of compiler commands in the source code, whereby the use of absolute addresses is only applied at defined positions.

4.1.2 Jump Table

If the compiler that is used, does not provide an option to disable relative jumps, an additional jump table in memory can be used. For this purpose, a table containing the absolute addresses is stored in memory near the function to be optimized. The relative jumps within the function

are manipulated in such a way that they refer to the correct lines within the table where the absolute address is located. The advantage of this implementation is that no support from the compiler is required. In contrast, the jump table occupies additional memory and each jump into the fast scratchpad requires an additional relative jump into the table before the actual target address is reached. This increases the runtime, which must be taken into account in the evaluation of the separation.

4.1.3 Memory Reservation

Another possibility is to reserve additional memory space directly next to a relative jump instruction. This can be achieved by integrating so-called zero-operations already in the C code. After compiling the source code, the memory space of the relative jump and the zero-operation is used to integrate an absolute jump. Similar to the jump table, this procedure does not require any support from the compiler. However, this variant requires a complex analysis to integrate the required zero-operations at the correct position in the source code.

4.2 Basic Block Allocation

To copy the selected basic blocks into the scratchpad memory of the microcontroller, the copy table must be extended accordingly. Using the copy table, the microcontroller copies the required functions and variables into the system's RAM during the startup code. Since this table is automatically created by the compiler during the compilation process, this causes a problem. In general, most compilers only allow complete functions and variables to be allocated to the scratchpad memory. For this reason the copy table is extended by a script to include the corresponding entries. In addition, the required memory is reserved in the linker script by means of a dummy section, with an adjusted size according to the required memory.

5 Experimental Results

The CoreMark of EEMBC in version 1.0 is used to prove the functionality. The reason for choosing the CoreMark as benchmark is that it is available in open source and has already been used in other publications. This circumstance makes it easier to relate the results of this article to the overall context of previous publications. Furthermore, the CoreMark was developed with a special focus on the evaluation of embedded microcontrollers and also offers multicore support.

The microcontroller used is an Infineon AURIX of the first generation, type TC29x with three processor cores, which is operated with a clock frequency of 200 MHz. The Infineon AURIX family uses the proprietary TriCore architecture, which is a modified Harvard architecture. Therefore each processor core has two interfaces, one for data and one for code. Each of these interfaces has two memories, one is a cache and one is a scratchpad RAM. The communication between the cores and the connection to the global memories is done via a crossbar. The basic structure can be seen in the figure 2. The memory sizes are described in the table 4.

For compiling the source code of the CoreMark, the TASKING compiler in version 6.2r1 for the TriCore architecture is used. The translation process of the this compiler is done in three steps. In the first step the C code is compiled into the Src format, which is then translated into the machine code by the assembler in the second step. At last the code is combined by the linker and allocated to the corresponding memories. The configurations for all steps used in this article can be taken from the table 5 [17].

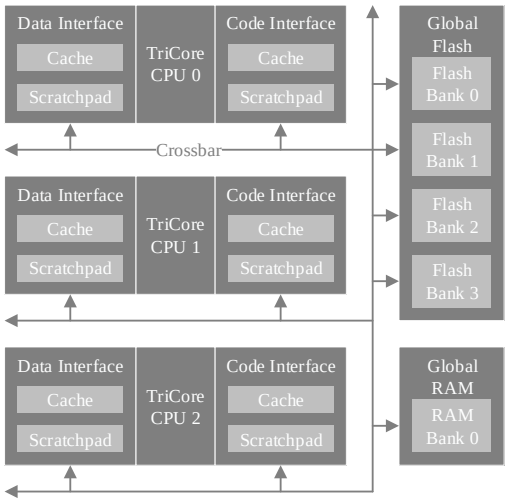

Figure 2 Infineon AURIX TC29x - Basic Memory Layout [7].

For the manual validation of the generated results of the offline simulator all optimization levels of the compiler as well as the linker are deactivated. This configuration generates entry and exit jumps for loops, which are required for the separation of the basic blocks in this test series. Another special feature of the assembler configuration is the avoidance of relative jumps. With the settings made in this experiment, all jumps are implemented absolutely, which, as already described in chapter 4, is one way to separate the program code [17]. The previous approach to distributing source code to the available memory in the system is based on the call frequency and the memory consumption of the respective functions. For this purpose, the maximum number of times the individual functions are called during a defined period of time is determined. The call frequency is used to calculate their priority and the functions with the highest priority are allocated to the fast SRAM memory until it is filled completely. For this comparison, this method is taken as a reference. Therefore, in the first step, the maximum call frequency and its memory consumption is determined for each function of the CoreMark. The table 6 shows the number of calls at 20,000 iterations for each function, whereby the ordering already corresponds to the priorities calculated. For better overview, only the functions that are called during the benchmark measurement are listed.

In the table 7, the decomposition into basic blocks is carried out for three functions as an example. For the purpose of better overview, only those basic blocks are shown that are relevant for the proposed optimization. The remaining instructions within this

Table 4 Infineon AURIX TC29x - Memory Dimensioning [7].

Category	Memory	Size/KB
Local	Data Scratchpad	240
	Data Cache	8
	Code Scratchpad	32
	Code Cache	32
Global	Flash	8192
	SRAM	32

Table 5 TASKING Compiler 6.2r1 - Configuration.

Utility	Configuration
C-Compiler	-O0
Assembler	-OgS
Linker	-O0

Table 6 EEMBC CoreMark 1.0 - Call Frequency and Memory Usage (Functions).

Function Name	Call Frequency	Size /Byte
ee_isdigit	78400000	44
core_state_transition	20480000	644
crcu8	11680008	92
crcu16	5840004	32
crc16	5240004	16
calc_func	4442360	240
cmp_idx	4161115	76
core_list_find	4120000	106
core_list_reverse	4080000	38
cmp_complex	2221180	44
crcu32	1280000	36
matrix_sum	320000	128
matrix_add_const	160000	74
core_bench_state	80000	422
matrix_test	80000	272
matrix_mul_matrix_bitextract	80000	162
matrix_mul_matrix	80000	146
matrix_mul_vect	80000	112
matrix_mul_const	80000	74
core_bench_matrix	80000	52
core_list_mergesort	60001	290
core_bench_list	40000	440
core_list_remove	40000	46
core_list_undo_remove	40000	40
Total		3626

function are only executed once per function call. In fact of this they cannot be used for further decomposition. For each of the basic blocks, the repetition during the entire benchmark execution is specified, which is calculated from the call frequency of the function and repetitions within the function. In addition, the memory requirements for each basic block in the scratchpad are also specified.

All basic blocks in the table 7 are loops which, due to their repetitions, cause a high computing load with low memory consumption. The basic blocks of the matrix_sum function are special because they are nested loops. However, since the analysis of the machine code is performed at the instruction level, the offline simulator detects this structure due to the different execution frequencies.

In the measurement, which is shown in figure 3, the presented method is set in relation to the previous approach. To evaluate the two allocation strategies, the CoreMark is executed with different scratchpad sizes. In the function-based approach, the functions are allocated to the fast memory in the order of their priority, which is shown in table 6. For a comparison of the two approaches, the distribution of the basic blocks is done according to a similar scheme. The advantage of the basic block allocation is the finer granularity. Due to the fact that functions often contain execution paths that are rarely processed, function-based allocation

Table 7 EEMBC CoreMark 1.0 - Call Frequency and Memory Usage (Basic Blocks).

Function Name	Basic Block Call Frequency Total	Size /Byte
crcu8	93440064	72
matrix_sum	2880000	14
	25920000	90
core_bench_list	4080000	192
	1120000	32
	1160000	32

is less efficient when using the small scratchpad memory. The difference is particularly noticeable at the beginning of the measurement series, where the execution time decreases significantly faster with the basic block-based method. The disadvantageous course of the function-based procedure is caused by the fact that the function core_state_transistion is called frequently, but consumes a lot of memory. As a result, additional functions can only be allocated to the scratchpad once its size exceeds 800 bytes. Analogous to the staged sequence of the function-based allocation, a similar paragraph for the basic block-based procedure can be seen at 1 KB. This is a relatively large basic block that is unsuitable for further decomposition. By using better distribution algorithms, however, the process could be even better balanced. This should be evaluated in further investigations. Furthermore, it can be seen in the course that with increasing size of the scratch pad, the two methods converge, since all frequently used instructions are available in the fast memory. As the CoreMark with the current settings requires 3626 bytes of memory, the measurement results are identical from this scratchpad size on.

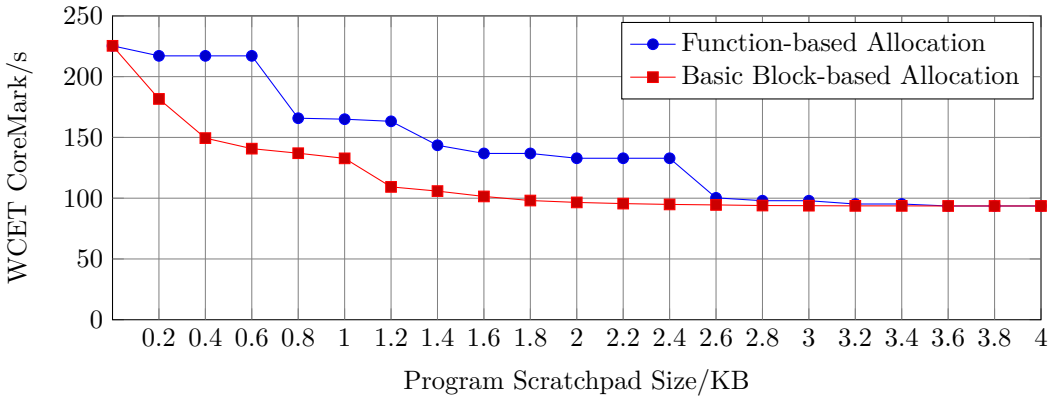

Figure 3 Execution Time of the CoreMark 1.0 with different Scratchpad Allocation Methods.

6 Discussion

The main goal of this work is to use the local scratchpad memory more efficiently for the program code. Especially in systems with hard real-time requirements these memories provide a fast and deterministic way to increase the execution speed. However, the table 1 shows that the size of the flash memory is often larger than the program scratchpad by a factor of 85, as shown in the Infineon AURIX TC29x. As this relation already illustrates, the optimal use of

the scratch pads is therefore essential for reaching full performance. The method presented in this article shows a way how the low capacity of the scratchpad memory can be utilized better. The results on the AURIX platform in chapter 5 clearly show that, in contrast to the function-based approach, the use of basic blocks scales better with reduced memory sizes. However, this is at the cost of a slightly increased memory requirement in flash memory for all variants of separation. This is due to the fact that for absolute jumps, compared to the relative jumps, the complete address must be stored in memory. This restriction only exists for Instruction Set Architecture (ISA) that support relative jumps. A further problem is currently the analysis of the machine code. Particularly in safety-critical systems, interrupts and external signals are analyzed and it is necessary to react accordingly. Due to these unpredictable input variables, a realistic estimation of the maximum call frequency of functions is almost impossible. Furthermore, there are states in every system which are mutually exclusive, which effects the call frequency of functions. These complex correlations are extremely difficult to extract and require further investigations for a better evaluation of the WCET [5] [1]. In previous studies, the code is translated with the optimization level O0, whereby only absolute addresses for the jump instructions are generated. By this procedure optimizations are deactivated too, which have a significant influence on the execution speed. Therefore the goal is to consider further optimization levels and compilers in future extensions to achieve more realistic results. For the measurements performed so far, only the CoreMark from EEMBC was used, which only represents a small number of use cases. For this reason, a wide range of benchmarks will be ported in future research so that many different memory access types and patterns in the allocation can be analyzed and taken into account. In this context, the previous approaches to an optimized storage strategy will be ported and compared with the concept presented in this article. The intention is to give an overview which distribution method achieves the best results with which memory access pattern. A potential extension is the integration of caches in the distribution of the basic blocks. In contrast to the dynamic use of scratch pads, the address calculation for caches is done by the microcontroller. Thus, the overhead that would normally occur when calculating addresses in software can be avoided. However, the use of caches in real-time systems is associated with significantly higher effort in timing analysis, which in turn makes allocation more difficult [3]. Furthermore, support for other microcontroller architectures is planned for future work. The focus will be on multicore architectures, as these will benefit even more from local scratch pads due to concurrent access to shared memory. By optimizing the use of these core-exclusive memories, timing anomalies resulting from concurrent accesses could be prevented even more effectively. For this purpose, the analysis and prioritisation of the basic blocks would have to be extended accordingly [14].

References

1 N. Akram, Y. Zhang, S. Ali, and H. M. Amjad. Efficient task allocation for real-time partitioned scheduling on multi-core systems. In *2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 492–499, 2019. doi:10.1109/IBCAST.2019.8667139.

2 Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.

3 Marco Caccamo, Marco Cesati, Rodolfo Pellizzoni, Emiliano Betti, Roman Dudko, and Renato Mancuso. Real-time cache management framework for multi-core architectures. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 2013.

4 Zhong-Ho Chen and Alvin Su. A hardware/software framework for instruction and data scratchpad memory allocation. *ACM Transactions on Architecture and Code Optimization*, 7, April 2010. doi:10.1145/1736065.1736067.

5 C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann. Syswcet: Whole-system response-time analysis for fixed-priority real-time systems (outstanding paper). In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 37–48, 2017.

6 Infineon Technologies AG, 81726 Munich, Germany. *TriCore TC1.6P & TC1.6E Core Architecture*, February 2012.

7 Infineon Technologies AG, 81726 Munich, Germany. *AURIX TC29x B-Step User's Manual V1.3*, December 2014.

8 Philipp Jungklass and Mladen Berekovic. Effects of concurrent access to embedded multicore microcontrollers with hard real-time demands. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–9, 2018.

9 Philipp Jungklass and Mladen Berekovic. Intercore-kommunikation für multicore-mikrocontroller. In *Tagungsband Embedded Software Engineering Kongress 2018*, 2018.

10 Philipp Jungklass and Mladen Berekovic. Memopt: Automated memory distribution for multicore microcontrollers with hard real-time requirements. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS)*, 2019.

11 C. Kachris, G. Nikiforos, V. Papaefstathiou, X. Yang, S. Kavadias, and M. Katevenis. Low-latency explicit communication and synchronization in scalable multi-core clusters. In *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, pages 1–4, 2010.

12 Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastava. Wcet-aware dynamic code management on scratchpads for software-managed multicores. *Real-Time Technology and Applications - Proceedings*, 2014:179–188, October 2014. doi:10.1109/RTAS.2014.6926001.

13 Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison. *Institut de Recherche en Informatique et Systèmes Aléatoires - Publication Interne No 1818*, January 2006.

14 Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 220–229. IEEE Press, 2015.

15 STMicroelectronics, Schiphol, Amsterdam, Niederlande. *SPC58xEx/SPC58xGx 32-bit Power Architecture microcontroller for automotive ASILD applications - Reference Manual*, August 2018.

16 V. Suhendra, T. Mitra, A. Roychoudhury, and Ting Chen. Wcet centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium*, pages 10 pp.–232, 2005. doi:10.1109/RTSS.2005.45.

17 TASKING BV, Spoetnik 50, 3824 MG Amersfoort, Netherlands. *TASKING VX-toolset for TriCore User Guide*, ma160-800 (v6.2) edition, December 2016.

18 Texas Instruments Incorporated, Post Office Box 655303, Dallas, Texas 75265. *TMS320F2838x Microcontrollers Technical Reference Manual*, May 2019.

19 G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 2016. doi:10.1109/TC.2015.2500572.