A Hardware-assisted Approach for Non-invasive and Fine-grained Memory Power Management in MCUs

Michael Kuhn, Patrick Schmid and Oliver Bringmann
Embedded Systems, University of Tübingen, Germany

***Abstract*—The energy demand of embedded systems is crucial and typically dominated by the memory subsystem. Off-the-shelf MCU platforms usually offer a wide range of memory configurations in terms of overall memory size, which may differ in the number of memory banks provided. Split memory banks have the potential to optimize energy demand, but this often remains unused in available hardware due to a lack of power management support or require significant manual effort to leverage the benefits of split-banked memory architectures. This paper proposes an approach to solve the challenge of integrating fine-grained power management support automatically, by a combined hardware/software solution for future off-the-shelf platforms. We present a method to efficiently search for an optimized code and data mapping onto the modules of split memory banks to maximize the idle times of all memory modules. To non-invasively put memory modules into sleep mode, a PC-driven power management controller (PMC) autonomously triggers transitions between power modes during embedded software execution. The evaluation of our optimization flow demonstrates that memory mappings can be explored in seconds, including the generation of the necessary PMC configuration and linker scripts. The application of PC-driven power management enables active memory modules to remain in light sleep mode for approximately 13% to 86% of the execution time, depending on the workload and memory configuration. This results in overall power savings of up to 24% in the memory banks, in terms of static and dynamic power.**

***Index Terms*—Energy efficiency, Low-power electronics, Memory architecture, Embedded systems, Computer architecture**

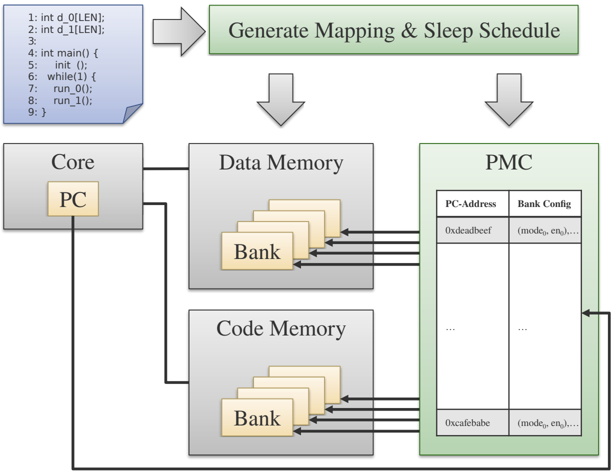
I. Introduction

Smart embedded devices permeate throughout many aspects of industrial processes, as well as our personal lives. Those devices are often microcontroller-based small-footprint devices which process data from a few numbers of sensors and optionally drive some actuators or send the processed data to other devices. Two aspects are important to consider when planning and designing sensor devices on a large scale – the development cost and the energy costs for powering the devices.

Application-specific power management offers great potential for low-energy computing in embedded devices, in controlling the power modes of specifically enhanced architectural components by the use of many different techniques (i.e., adjusting clock frequencies, clock gating, power gating, body biasing, ...) on different domains of the system (i.e., processor core, periphery, memory subsystem, ...). Typically, the software running on these devices can make use of the power-management techniques by triggering Power Modes (PMs) provided by a Power Management Controller (PMC). Since modern sensor applications frequently rely on memory-intensive data-processing algorithms, the impact of the memory subsystem is becoming a dominant factor on the overall energy demand of the system and should therefore be the focus of optimization efforts. For many embedded products, application-specific hardware/software design is too costly, so off-the-shelf Micro Controller Units (MCUs) are used. However, many MCUs have only very few physical memory banks, which limits the possibilities to apply power management, or have no, or only limited, possibilities for fine-granular power management for the on-chip memory, even if the memory banks are divided into multiple physical memory modules. For example, the Ambiq Apollo4 Blue Plus has three different directly accessible memory banks (TCM, Extended SRAM, Shared SRAM), and offers the possibility to configure which memories should be powered down in dependence on the system's power state. Therefore, it would be beneficial, that off-the-shelf ultra-low power MCUs implement the system memory with split banks, where the PMs of each memory module can be triggered individually. The RP2040 and RP2350 MCUs, used in the Raspberry Pi Pico family, have already implemented this architecture. Each module's power state in the split main system memory bank can be controlled manually via a configuration register [1], [2]. The remaining issue with their solution is, that the software code has to be instrumented with additional statements to trigger PMs switches of the memory. Depending on the built-in memory, putting it in a sleep mode is profitable after only a few CPU clock cycles. To apply power management to existing hardware on the granularity of only a few clock cycles, the software code has to be altered heavily, which is not only inconvenient, but also changes the run time and energy demand of the software.

In order to implement a non-invasive and fine-grained memory power management, we propose a combined hardware/software solution for future off-the-shelf platforms. An overview of our solution is shown in Fig. 1. This solution entails the extension of existing MCU architectures by integrating a PMC, that is configured prior to the runtime of the target application, and then driven by the MCU's Program Counter (PC) to autonomously issue power mode changes. To complement that, we present a software toolkit that is designed to adapt the target software automatically to this architecture by generating an optimized memory mapping and a sleep schedule for the PMC. Code and data objects are mapped to the memory modules in a manner that maximizes the idle time for each memory module of the System-on-Chip (SoC). The mapping

This work has been partly funded by the German Federal Ministry of Education and Research (BMBF) in the project Scale4Edge under reference no. 16ME0122-140.



```
int d_0[LEN];
int d_1[LEN];

int main() {
    init ();
  while(1) {
    run_0();
    run_1();
}
```

Generate Mapping & Sleep Schedule
Core
PC
Data Memory
Bank
Code Memory
Bank
PMC
PC-Address
Bank Config
0xdeadbeef
$(\text{mode}_0, \text{en}_0), \ldots$
...
...
0xcafebabe
$(\text{mode}_0, \text{en}_0), \ldots$

Fig. 1. PC-driven memory power management with an application-specific sleep schedule

is realized via an automatically generated linker script. Then, a PMC sleep schedule is created with the purpose of using the idle times to issue sleep modes, thus minimizing the memory on-time. The generated configuration is included in the target software along with the necessary PMC setup-code.

II. Related Work

Existing work, tackling the optimization of the energy demand of a systems' memory, mostly relies on the fact, that the energy demand of Static Random Access Memories (SRAMs) typically varies with the size, which is especially useful on systems with a memory hierarchy. For example, in [3], the authors combine task scheduling for real-time applications on multicore systems with an efficient use of local and shared memories. However, optimization approaches for systems that feature a flat memory hierarchy, can be categorized into approaches targeting systems with or without Scratchpad Memories (SPMs). A great part of the existing literature is based on the concept of SPMs, which are not in the scope of the present work. The method described in [4] involves dynamically copying instruction data into SPM during execution. In contrast, [5] focuses on optimizing the data mapping for pipelined streaming applications. The authors of [6] propose a mapping scheme for code and data that performs comparable to hardware caches. Additionally, [7] examines an efficient data mapping to a hybrid SPM composed of SRAM and non-volatile memory to minimize energy demand. These papers mainly focus on the optimization of the performance, not the energy efficiency. In contrast, [8] is focused on the energy demand by an efficient data allocation on a scratchpad consisting of volatile and non-volatile memory.

There are also some papers which focus on a classic MCU memory hierarchy, consisting only of separated code and data memories. Pallister et al. propose in [9] a method to find and apply an energy-efficient mapping of parts of the instruction code from the code memory, realized with flash, to the SRAM, exploiting the fact that read accesses to the SRAM demand less energy than accesses to the flash memory. The optimization is implemented on binary Basic Block (BB) level, therefore each relocated BB causes the need for rewriting jump targets in the target software. Power management is not discussed in this paper. Ozturk et al. [10] propose an approach to fit software to unequally sized banked memory architectures with the help of an ILP solver in order to apply power management to the memory DRAM banks, with the objective of saving energy. To achieve this, they employ data migration, compression and replication, which results in significant code modifications to the target software. Furthermore, a customized compiler is required. Steinfeld et al. [11] propose to split the main system memory in equally sized banks for event-driven applications and found that in terms of the energy demand, the split is beneficial with an adjusted mapping. However, it is not discussed how transitions to sleep modes are realized. Transitions from sleep to active modes are triggered by memory accesses. Therefore, the approach is invasive in terms of the execution time. Strobel et al. propose in [12] an approach to optimize the energy demand for embedded systems, by co-designing the size and amount of memory modules together with the mapping of parts of the software to these memories. Their proposed strategy is based on the concept of holding more frequently used data in smaller memory modules. This approach is further developed in [13], where the optimization of static power is achieved through the implementation of power management in a second optimization step. The authors present the potential for scheduling power mode changes at the granularity of clock cycles but dismiss this approach as impractical.

III. Considerations and General Approach

In order to provide embedded system developers a tool to automatically generate an energy-optimized variant of their software that targets MCUs with split memory banks, we propose a solution that involves a PC-driven PMC in hardware, as well as a toolkit that optimizes the software to minimize the overall memory on-time. For the remainder of this paper, we assume that the software is running in a bare-metal configuration, that is, without any dynamic task scheduling, but in a traditional main loop. In contrast to existing work, such as [13], the main optimization goal of our approach is to minimize the overall energy demand caused by static power. The rationale for focusing on static power, as opposed to dynamic power, is that with shrinking semiconductor fabrication scales, the static share of the overall power demand of circuits is growing to the point of eventually becoming a dominant part. Moreover, equally sized memory modules can be assumed for general purpose hardware, in which case an optimization of the dynamic power demand is not applicable by adjusting the memory mapping. Nonetheless, in our evaluation, we also consider the effects of our optimization on the dynamic power.

In contrast to existing work, such as [9], we focus on relocation of code and data at the source-code level for an efficient memory mapping. This means that entire functions and data structures are allocated to the available memory modules. Benefits of this approach are a reduced complexity of the relocation process, and the possibility to work with unmodified standard compiler toolchains. This also means that there is no necessity for specialized tooling to rewrite the source or binary code. In contrast to the approach proposed by Ozturk et al. [10], we consider static mappings of code objects, which eliminates the necessity for data migration at runtime. The generation of a linker script is sufficient to produce the optimized mapping, if the compiler is configured to place every function and data element into a separate section.

To the best of our knowledge, all existing approaches require the explicit instrumentation of PM transition statements within the software code, to apply a scheduled transition to a sleep mode. If a memory module in a sleep mode state is accessed, the processor pipeline has to be stalled, in order to wait for the memory to return to an active state. Alternatively, another explicit statement in the software code has to be introduced. However, such invasive statements limit the potential for optimization, since every single transition has to be instrumented in the source or binary code. In many applications, it is preferable to have a non-invasive approach. Therefore, we favor an approach that is capable of effecting power mode transitions without instrumenting the target software code. This objective can be accomplished by integrating a PC-driven PMC into the hardware, that autonomously monitors the software execution on the processor core, and triggers the PM transitions on predefined instruction addresses.

A. Memory Power management

We assume the built-in memory modules $m \in \mathcal{M}$ have several power modes $p \in \mathcal{P}$, where $\mathcal{P}$ denotes the set of all available power modes for the memory and $\mathcal{M}$ all available SoC memory modules. Each memory has one active mode, and at least one sleep mode. For each power mode p, the memory macro has an average static power rating $P_{static}^{m,p}$. Entering and exiting a sleep mode p from the active mode leads to a switching penalty $E_{switch\text{-}penalty}^{m,p}$, i.e., the energy that is required during the switching operation. These characteristics can be derived from data sheets or measurements.

We define the energy savings for memory module m in the time interval $T_i = (t_i, t_i')$, with $|T_i| = t_i' - t_i$, where sleep mode p is applied as:

$$E_{saving}^{m,p} = \left[P_{static}^{m,active} - P_{static}^{m,p} \right] \cdot |T_i|. \tag{1}$$

To maximize the benefits of memory power management, sleep modes should be used whenever the power savings in an interval T_i outweigh the PM switching penalty:

$$E_{saving}^{m,p} > E_{switch\text{-}penalty}^{m,p}. \tag{2}$$

For each power mode, we derive the break-even time, from (2):

$$t_{break\text{-}even}^{m,p} := |T_i| = \frac{E_{switch\text{-}penalty}^{m,p}}{P_{static}^{m,active} - P_{static}^{m,p}}, \tag{3}$$

as the duration of a sleeping interval, where the energy savings are equal to the incurred switching penalty. Therefore, a power mode p should be applied for a memory module $m \in \mathcal{M}$ during a time interval T_i in-between two consecutive memory accesses at timestamps t_i and t_i', when:

$$|T_i| > t_{break\text{-}even}^{m,p}. \tag{4}$$

B. Mapping optimization

The sleep mode schedule is generally dependent on the specific memory mapping, since the access patterns to the memory modules change with the mapping. We define a set $\mathcal{O}$, which includes all function and data objects of the target software. A code mapping:

$$\mathcal{X} = \{(o,m)|o \in \mathcal{O}\} \subset \underbrace{\mathcal{O} \times \mathcal{M}}_{\text{all possible assignments}} \tag{5}$$

is a set of tuples that assigns each object to exactly one memory module $m \in \mathcal{M}$. For each object $o \in \mathcal{O}$, we define

$$\mathcal{I}_o = \{T_0, T_1, \dots, \}, \tag{6}$$

as the set of time-intervals between consecutive memory accesses to the object o. Given a mapping $\mathcal{X}$, we define the time-intervals where the memory module m is not accessed by the processor core as:

$$\mathcal{I}_m^{\mathcal{X}} = \bigcap_{o \in \{o'|(o',m') \in \mathcal{X} : m'=m\}} \mathcal{I}_o. \tag{7}$$

For example, assuming a mapping $\mathcal{X} = \{(o_0, m), (o_1, m)\} \subset \{o_0, o_1\} \times \{m\}$, with $\mathcal{I}_{o_0} = \{(0,5)\}$ and $\mathcal{I}_{o_1} = \{(2,8)\}$, the set of time-intervals where no access on memory m takes places is then defined as:

$$I_m^{\mathcal{X}} = \mathcal{I}_{o_0} \cap \mathcal{I}_{o_1} = \{(0,5)\} \cap \{(2,8)\} = \{(2,5)\}.$$

Our optimization goal is to find a mapping $\mathcal{X} \in \mathcal{Z}$, which minimizes the overall static energy for all memory modules, where $\mathcal{Z}$ is the set of all possible mappings. The objective function is defined as:

$$\underset{\mathcal{X} \in \mathcal{Z}}{\arg\min} \sum_{m \in \mathcal{M}} \sum_{T_i \in \mathcal{I}_m^{\mathcal{X}}} P_{static}^{m,p} \cdot |T_i| + E_{switch\text{-}penalty}^{m,p}. \tag{8}$$

Our approach to solve the optimization problems is discussed in Section IV. During the minimization process, for each memory macro m in every time interval $T_i \in \mathcal{I}_m^{\mathcal{X}}$ we determine the optimal PM $p \in \mathcal{P}$:

$$\underset{p \in \mathcal{P}}{\arg\max}\, (P_{static}^{m,active} - P_{static}^{m,p}) \cdot |T_i| - E_{switch\text{-}penalty}^{m,p}, \tag{9}$$

that provides the lowest average energy. Since $|\mathcal{P}|$ is typically very small, finding the optimal PM is generally fast.

C. PC-driven Power Management Controller

In order to realize the non-invasive power management, we propose the following architecture for a PC-driven PMC, an overview is shown in Fig. 2. The PMC is configured by the target software itself, usually once at the beginning of the software execution. The configuration is transmitted via an APB interface, and contains a mapping from PC addresses to a PM configuration for each memory module. Each configuration consists of a power mode and a corresponding enable-bit for each memory module. The enable-bit allows triggering power mode transition on a subset of memory modules and facilitates the generation of a power mode schedule. During program execution, the PMC observes the PC from the processor core. The PC is then marked as valid if no pipeline flush is detected.

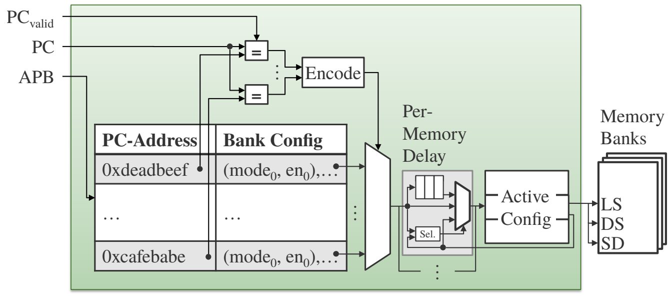


Fig. 2. PC-driven Power Management Controller

If the PC is valid and matches with an entry in the configured PC table, the configuration will be applied to the memory power control signals. In pipelined processor cores, it is not always possible to change the state of a memory module into a sleep-mode at the same time a PC match is registered. This is because the current instruction in execution might access exactly this memory module and can require multiple clock cycles before the execution is actually finished. Our PMC solves this problem is by a configurable delay register, which is only used for transitions to sleep modes, but not for transitions to the active mode. The length of the delay register can be easily adapted, allowing the PMC to be used with different pipeline lengths. As a consequence of this implementation, the PC can be routed from an early pipeline stage.

IV. Implementation of Optimization Flow

Our optimization flow consists of a preparation step 1) and the optimization steps 2) and 3), as outlined in Fig. 3. The inputs are a software project, implemented in the C programming language, and a definition of relevant hardware specifications. This includes a description of the target's memory configuration and the number of available configuration entries on the PMC.

1) Generation of Time-stamped Memory Trace: The preparation for both optimization steps is to build the software project from its source code, and link a target binary with a default linker script, ignoring the memory bank split. The target binary is then passed on to an Instruction Set Simulator (ISS), for which we use ETISS [14]. Prior to the simulation with the ISS, we determine Basic Block (BB) timings similar as in [15]. We extended ETISS to simulate the timing behavior on BB-level, along with the functional simulation. In addition to that, we modified the ISS to output a time-stamped trace, containing all executed instructions, including memory addresses of accessed instructions and data.

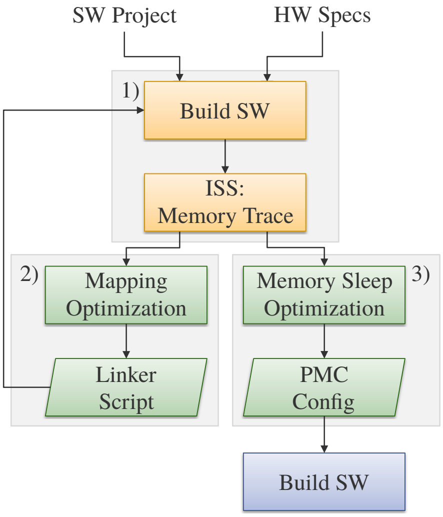


Fig. 3. Optimization Flow

2) Memory Mapping Optimization: In the first optimization step, the time-stamped trace is analyzed for the potential sleep intervals between memory accesses. All considered mappings of code objects to the respective memory modules are then evaluated with the potential energy savings of every sleep mode in all potential sleep intervals, respecting the inflicted switching penalties. The best mapping is found according to the description in Section III-B. The optimized mapping is then passed to a template-based linker script generator, that locates all mapped function and data sections to the determined memory modules.

3) Memory Sleep Schedule Optimization: We prepare the second optimization step, by reconfiguring the software project to use the newly generated linker script, and a second ISS-simulation with a newly built target binary. To establish a sleep schedule for the software project, we analyze the new ISS-trace. The power-mode schedule is generated by iterating over the memory trace and comparing the timestamps of instructions to the access times of the memories' sleep intervals. We then schedule the power mode transitions according to the maximized savings, as defined in Section III-A. The transitions are associated with specific instruction addresses, to mark the points during execution where transitions are triggered in time. To realize non-invasive transitions between power modes without introducing additional wait cycles, the transitions are aligned with memory access instructions, and their execution in the processor pipeline. Based on this schedule, we generate a new header file containing the PMC configuration. We then reconfigure the software project to include this configuration header. Finally, we perform a build of the optimized software.

A. Efficient Mapping Optimization

As already stated in reference [13] an exhaustive search in all parameter combinations of this optimization problem is infeasible in general, due to the exponential number of combinations represented by $|\mathcal{M}|^{|\mathcal{O}|}$. That is, depending on the number of code objects and memory modules, the number of possible combinations quickly exceeds typical amounts of workstation system memory and leads to excessive run times. Nevertheless, a few key facts reduce the number of combinations to be examined to a more realistic number, which is why we were able to find mapping solutions for the benchmarks we used in a short time. The number of memories is limited in practice, due to the fact that smaller memories have a larger overhead regarding area and power, in general. Additionally, common MCUs have separate code and data memory banks to avoid the von Neumann bottleneck, so we consider the respective

TABLE I
Relative Time spent in Power Modes for Code Memory

Benchmark	**Mode**	**Code Memory Banks Sleep [%]**			
		0	1	2	3
matmult	LS	0.00	0.00	0.00	0.00
	SD	0.00	100.00	100.00	100.00
compress	LS	0.00	0.00	0.00	0.00
	SD	0.00	100.00	100.00	100.00
st	LS	0.00	0.00	0.00	0.00
	SD	0.00	100.00	100.00	100.00
nbody	LS	0.00	0.00	0.00	0.00
	SD	0.00	100.00	100.00	100.00

TABLE II
Relative Time spent in Power Modes for Data Memory

Benchmark	**Mode**	**Data Memory Banks Sleep [%]**			
		0	1	2	3
matmult	LS	0.00	0.00	0.00	83.38
	SD	0.00	0.00	0.00	0.00
compress	LS	0.00	38.78	0.00	15.30
	SD	0.00	0.00	100.00	0.00
st	LS	78.74	0.00	0.00	93.39
	SD	0.00	100.00	100.00	0.00
nbody	LS	81.73	0.00	0.00	3.86
	SD	0.00	100.00	100.00	0.00

memory modules for code and data objects. Moreover, in some cases, the size of a code object may exceed the capacity of a single memory module, then multiple modules have to be pooled to one virtual module, which reduces the number of targeted memory modules further. If required, the number of code objects to map can be reduced by bundling small data objects into virtual symbols. This is because, in contrast to large structures of linear data, these objects do not imply longer periods of usage of the same memory module. To reduce the number of mappings before evaluation, we remove impossible mappings, due to the space limitations of each memory module. We also eliminate equivalent mappings, which is particularly beneficial in the case of equally sized memory modules.

V. Evaluation

The optimization approach is validated through a prototype of the PMC, that we have implemented at Register-transfer Level (RTL) and integrated into the PULPissimo platform [16]. We configured the platform to contain one four-stage pipelined RISC-V core of the type CV32E40P, from which we extract the PC of the *ID*-stage. A separate signal marks the value of the PC as valid, if the *ID*-stage is not flushed due to branch mispredictions. Our PMC is configured for two delay cycles of transitions to sleep modes, so they can be scheduled at the address of memory access instructions to the respective memory module. For our design, we use the 22FDX FD-SOI technology from Globalfoundries as synthesis target technology, with standard cells and low-leakage memory macros from Synopsys. The PULPissimo platform uses three memory banks, two equally sized private memory banks of the processor core, and a shared memory bank that can be used by peripheral platform components. The shared memory bank is excluded in our analysis, as memory access patterns of peripheral platform components are not part of this work. The private memory banks are connected to the processor core with separate interfaces, one for the instruction code and one for data accesses. All code and data objects are loaded from a read-only memory to the respective memory banks in a boot stage, before execution. We split the private memory banks into four equal-sized modules each, and examine two memory configurations, shown in TABLE III.

A. Full system evaluation

We conducted experiments of the complete optimization flow with benchmarks from the BEEBS Suite [17], compiled with the riscv-gnu-toolchain in version 2.5.0, provided along with the PULPissimo platform. We applied our optimization flow, described in Section IV, on four benchmarks, representing various applications. The resulting software contains a specific power-mode schedule for each memory module. For our analysis, we choose the memory configuration depending on the data memory usage of the benchmarks, and a memory module can be in either of the three following power modes: (1) Active, (2) Light Sleep (LS) or (3) Shut Down (SD). Our tooling ensures that memory modules that are actually used during the execution, are set to LS in the appropriate periods between accesses. Memory modules that are not used, are set to SD with the first applied PMC entry. Following the application of our optimization flow, we simulated the execution of the benchmarks on our MCU platform at the RTL to get exact execution timings and to rule out accesses to sleeping memory modules. The RTL simulation generates a record of all power mode transitions, which is used to determine the cumulative time spent in all available power modes and the number of power mode transitions. The total static power dissipation and savings are then derived using the memory model described in Section III-A. TABLE III shows the memory configuration and number of used PMC configuration entries for the executed benchmarks, along with the resulting absolute and relative energy savings compared to the baseline where all memory modules are always active.

The code size of the executed benchmarks did not exceed the size of one memory module, consequently all but one code memories were shut down during the entire execution, as shown in TABLE I. As a result, our subsequent analysis is focused on the data memory. The relative time spent in each power mode per data memory module is shown in TABLE II. Only *matmult* allocates all available data memory modules, *compress* allocates three, and *st* and *nbody* allocate two modules each. In order to provide meaningful data for the evaluation of the effectiveness of the PC-driven PMC, which enables the use of as many light sleep intervals as possible, we show the relative energy savings gained through the light sleeps on allocated memories in TABLE IV. The allocated memory modules remain in light

TABLE III
SYSTEM SPECIFICATIONS, OPTIMIZATION FLOW RUNTIME AND ENERGY SAVINGS (BASED ON STATIC POWER)

Benchmark	**Specs**		**Flow Runtime [s]**		**Energy [nJ]**				**Energy Savings [%]**		
	Memory	PMC Entries	Overall	Mapping	Baseline[a]	LS Savings	LS Sw. Penalty	SD Savings	Total	LS	SD
matmult	8 x 16 KiB	11	12.22	0.26	296.1002	4.2121	0.6380	34.4489	12.84	1.21	11.63
compress	8 x 16 KiB	14	15.46	0.81	297.4506	1.9554	1.2052	46.1413	15.76	0.25	15.51
st	8 x 64 KiB	43	21.88	2.11	381.4798	26.8095	3.8827	130.4798	40.21	6.01	34.20
nbody	8 x 64 KiB	31	62.29	3.99	3820.7744	140.2182	12.6441	1306.8426	37.54	3.34	34.20

[a] All memory banks are always on

TABLE IV
ENERGY SAVINGS BY LS ON ALLOCATED DATA MEMORIES

Benchmark	**Allocated Memories**		**LS Savings**	
	Baseline [nJ]	Avg. LS [%]	Abs. [nJ]	Rel. [%]
matmult	148.0501	20.96	3.5741	2.41
compress	111.5440	13.52	0.7502	0.67
st	95.3700	86.07	22.9268	24.04
nbody	955.1936	42.80	127.5741	13.36

TABLE V
TOTAL ENERGY SAVINGS ON ALL MEMORIES
(BASED ON STATIC AND DYNAMIC POWER)

Benchmark	**Relative Savings [%]**	
	Gross	**Net**
matmult	23.50	21.64
compress	26.13	24.32
st	8.81	6.40
nbody	8.73	7.00

sleep mode for approximately 18% to 86% of the execution time. This results in an average reduction in static power consumption in active modules of up to 24%.

We also conducted a gate-level simulation and power estimation with the four benchmarks to compare the savings with the energy demand of the PMC itself. For this purpose, we synthesized two hardware designs, each with the same memory configuration as shown in TABLE III. The design with the smaller memory modules employs a PMC with 16 configuration entries, while the other design utilizes a PMC with 128 entries. Although the goal of optimization is to minimize static power, it also affects dynamic power and its contribution to total energy demand. Therefore, in addition to the results shown in TABLE III and TABLE IV, we also included the effects of the dynamic power. TABLE V shows the relative energy savings of the two memory banks during execution. The first column shows gross savings of up to 26%. The second column shows the net savings, i.e., the gross savings minus the energy demand of the PMC itself, with up to 24%.

B. PMC power evaluation

To determine how the average power consumption depends on the number and size of the comparisons, we conducted a comparative analysis of different hardware configurations. We therefore simulated a sequence of programming all PMC configuration entries, followed by a sequence of changing PC values. The average power is determined with a gate-level simulation of the standalone PMC module and a subsequent power estimation. The plot in Fig. 4 shows the average power of the PMC, relative to the power of the 32-Bit PMC with 128 entries, depending on the number of PMC configuration entries for two input sizes of the PC. The bit width is a consideration in this context, given that it affects the size of the comparisons with the instruction addresses in the PMC. The bit width results from the actual memory address range. The configuration of 18 bit corresponds to four code memory modules with 64 KiB each. The results show a considerable influence of the PMC size on the average power consumption of the PMC and a less strong influence of the bit width of the comparisons. Even though the results in TABLE V show that the benefits of the current implementation of the PMC already outweigh the energy overhead, it is clear that future implementations could reduce the overhead even further.

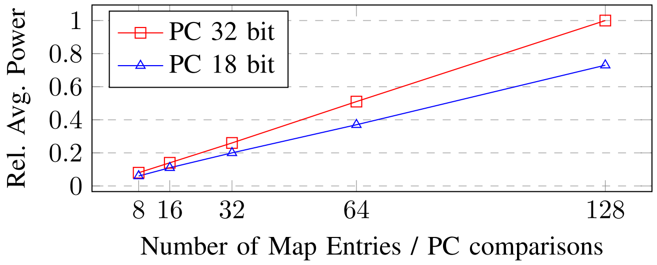


Fig. 4. Comparison of the relative average power of the PMC module, depending on the number of map entries and the PC bit width.

VI. CONCLUSION

We presented an approach to provide an automated flow for non-invasive and fine-grained power management in an MCU environment. This was achieved by developing a method to efficiently search for an optimized mapping of software code objects to individual modules of split memory banks, in conjunction with a hardware solution for autonomous power management. The results of our evaluation show that the proposed PC-driven PMC has the potential to fully exploit the benefits of using sleep modes when applicable. We have also shown that our approach provides energy savings of up to 24%, sufficient to justify its use. For future work, we consider the usage of execution contexts for power mode selection, and improving the energy-efficiency of the PMC hardware module.

References

[1] *RP2040 Datasheet*, Raspberry Pi Ltd, May 2024.

[2] *RP2350 Datasheet*, Raspberry Pi Ltd, Aug. 2024.

[3] C. Fu, G. Calinescu, K. Wang, M. Li, and C. J. Xue, "Energy-Aware Real-Time Task Scheduling on Local/Shared Memory Systems," in *2016 IEEE Real-Time Systems Symposium (RTSS)*. Porto, Portugal: IEEE, Nov. 2016, pp. 269–278.

[4] A. Janapsatya, A. Ignjatovic, and S. Parameswaran, "A Novel Instruction Scratchpad Memory Optimization Method based on Concomitance Metric," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, 2006, p. 6.

[5] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES '06*. Seoul, Korea: ACM Press, 2006, p. 401.

[6] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 472–511, May 2006.

[7] Y. Li, J. Zhan, W. Jiang, and J. Yu, "Energy optimization of branch-aware data variable allocation on hybrid SRAM+NVM SPM for CPS," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. Limassol Cyprus: ACM, Apr. 2019, pp. 236–241.

[8] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Data Allocation Optimization for Hybrid Scratch Pad Memory With SRAM and Nonvolatile Memory," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 6, pp. 1094–1102, Jun. 2013.

[9] J. Pallister, K. Eder, and S. J. Hollis, "Optimizing the flash-RAM Energy Trade-off in Deeply Embedded Systems," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 115–124.

[10] Ozturk, Ozcan and M. Kandemir, "ILP-Based energy minimization techniques for banked memories," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 1–40, Jul. 2008.

[11] L. Steinfeld, M. Ritt, F. Silveira, and L. Carro, "Low-Power Processors Require Effective Memory Partitioning," in *Embedded Systems: Design, Analysis and Verification*, G. Schirner, M. Götz, A. Rettberg, M. C. Zanella, and F. J. Rammig, Eds. Berlin, Heidelberg: Springer, 2013, pp. 73–81.

[12] M. Strobel, M. Eggenberger, and M. Radetzki, "Low power memory allocation and mapping for area-constrained systems-on-chips," *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, Jul. 2016.

[13] M. Strobel and M. Radetzki, "Power-mode-aware Memory Subsystem Optimization for Low-power System-on-Chip Design," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5, pp. 1–25, Oct. 2019.

[14] D. Mueller-Gritschneder, K. Devarajegowda, M. Dittrich, W. Ecker, M. Greim, and U. Schlichtmann, "The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping," in *Proceedings of the 28th International Symposium on Rapid System Prototyping Shortening the Path from Specification to Prototype - RSP '17*. Seoul, South Korea: ACM Press, 2017, pp. 79–84.

[15] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *Proceedings of the 48th Design Automation Conference on - DAC '11*. San Diego, California: ACM Press, 2011, p. 486.

[16] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: An ultra-low-power pulpissimo SoC in 22nm FDX," in *2018 IEEE SOI-3D-subthreshold Microelectronics Technology Unified Conference (S3S)*, 2018, pp. 1–3.

[17] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," 2013. [Online]. Available: https://arxiv.org/abs/1308.5174