ABSTRACT
We present a mathematical model for the problem of optimizing on-chip memory configurations to ensure minimal power consumption. Large memories consume more power than small ones, and the same relation holds for active and idle memories. Therefore, splitting memory configurations such that the most frequently accessed parts reside in separate but small memories can significantly reduce power consumption. The optimal selection of memory instances as well as an optimal mapping of application parts to these instances is a complex task. To solve this task, we have defined it as an integer linear programming model yielding the optimal solution. Given a set of benchmark applications, we show that using our model to split on-chip memory into multiple instances can yield power reductions of up to 80% for instruction memory and 73% for data memory. Furthermore, we present how our model can be used for system synthesis showing its application for HW/SW codesign.

Keywords
Integer Linear Programming, ILP, Low Power, On-chip Memory, SRAM, System-on-Chip, SoC

1. INTRODUCTION
The ubiquitous nature of embedded systems makes the need for the lowest possible power consumption a driving force for the design and development of embedded systems. Fortunately, these specialized systems often only perform a single task, which allows engineers to optimize specifically for that task without compromise. It has been shown that up to 60% or more of an embedded system’s power consumption is attributed to memory [8], which makes optimizing the memory sub-system an evident design goal.

A break down analysis of the energy consumption caused by reading from on-chip SRAM shows that less than one percent is consumed by the actual memory cells and about 90% by components such as precharge unit, sense amplifiers and address transition detection [4]. The energy consumption of these components is heavily affected by the overall size of the SRAM instance being accessed. One can make use of this to reduce the total energy consumption by splitting the on-chip memory into multiple instances such that frequently accessed segments of the address space reside in separate memory instances.

However, splitting the memory into multiple units requires an interconnect, e.g. a bus or a custom fabric, that forwards read and write requests to the individual memories. Obviously, this interconnect consumes energy itself, which with increasing number of memories can offset the benefits of the split memories. Furthermore, using multiple small memories instead of a single large one increases the required area and thus can become prohibitive. This makes selecting the optimal number and type of memories a non-trivial task.

It is also necessary to map address ranges to memory instances according to their access frequency to achieve the lowest possible power consumption. For example, if the most frequently accessed memory addresses are uniformly distributed over the application’s address space, frequent and infrequent addresses will inevitably be mapped to the same memory instances voiding any benefit of a split memory architecture.

In this work, we propose an integer linear programming (ILP) model solving the twofold problem of finding the optimal allocation of memory instances as well as an optimal mapping of address ranges to memories. The model is parameterized to allow user constraints such as limiting the maximum number of memory instances or the required area.

The rest of this work is organized as follows. Sec. 2 discusses existing research in the field of split memories before we declare our problem statement. Sec. 3 sets up the design space, which we use in our formal ILP model elaborated in Sec. 4. The application for system synthesis is illustrated in Sec. 5, and Sec. 6 provides results highlighting the benefits of our model. Finally, Sec. 7 concludes this paper.

2. RELATED WORK
Split memory configurations have been the subject of previous research. They have been investigated with different degrees of precision, ranging from algorithms suitable for
In the embedded domain, splitting memories is mainly used to reduce the power consumption, which Benini et al. [3] achieve by employing a recursive bi-partitioning algorithm. While the work claims optimality for its algorithm, this optimality is constrained by immutable the address space layout of the application, which can become a prohibitive limitation when the most accessed addresses are distributed. Mai et al. [8] mitigate the aforementioned limitation by allowing a reorganization of the memory layout. However, the problem is largely simplified to enable manual execution of the optimization process, which restricts the approach mainly to non-optimal solutions.

Srinivasan et al. [13] and [11] perform a combined optimization of memory and bus partitioning either for low power or for low chip area respectively. The combined problem increases the complexity, which constrains both works to heuristics. Furthermore, the address space is fixed, which limits the potential of the optimization.

Liu et al. [6] optimize the distribution of variables between phase change RAM and DRAM with the combined goal of a minimized power consumption, reduced number of writes to PRAM and optimized schedule of the respective application. The methodology is intended for DSP application and works on data flow graphs. Our approach solves a more general problem, as it takes an almost arbitrary number of memory types as input from which the solver may choose freely. Zhuge et al. [16] present another DFG based approach for split memory configurations; however, their method is intended to optimize the scheduling of applications and cannot be used for low power optimizations.

A problem closely related to finding an optimal split memory configuration can be found in the domain of scratch pad memories (SPM), which have proven to be an efficient replacement for caches in embedded systems [15]. Steinke et al. [14] use an ILP model to solve the problem of finding an optimal selection of memory ranges to be mapped to the SPM. However, the model has several limitations: it only considers a single on-chip memory of fixed size, and thus cannot be used for the synthesis of an optimal on-chip memory array; it lacks support for leakage power; and it does not differentiate between reading from and writing to memory. A more advanced ILP model is presented by Menichelli et al. [9] not only supporting CMOS leakage power but even low power modes for external memory, whenever the SPM is accessed. However, the model also assumes a single, fixed size scratchpad memory making it unusable for the synthesis of an optimal on-chip memory array.

Angiolini et al. [1] solve the problem of mapping address ranges to an SPM using dynamic programming. The approach allows for arbitrary small ranges to be mapped to the SPM increasing the complexity of the SPMs address decoder. However, this overhead is simplified by reducing the usable capacity of the SPM and only estimated. While seemingly similar, the work solves a different problem as, as it does not actually split the SPM but only partitions its locally by means of a complex address decoder.

Loghi et al. [7] optimizes SPMS for low power by splitting the SPM into subbanks while considering leakage power and supporting power gating of idle banks. The work only uses simplified traces, by segmenting the address space and execution time into fixed size chunks. The trace only records whether a memory segment has been accessed in a given time slice or not and not how often. The algorithm cannot change the number of subbanks according to its needs; however this is an important feature as our evaluation shows that the optimal number of partitions cannot be known in advance.

Balasa et al. [2] optimize the size of software managed SPMS by analysing an application’s algorithmic description and its usage of multidimensional datastructures. While allowing efficient usage of SPMS, the applicability of the approach is limited to applications in the signal processing domain. Furthermore, the methodology cannot be used to identify the optimal size of an SPM for a given application.

The main contribution of this work is a mathematical model to identify a low power memory configuration. The model differentiates between read and write accesses and supports leakage power. Furthermore, it considers the deselect power, which is caused by toggling input lines even when a memory is not selected. From an almost arbitrary list of memory types, the model selects an optimal number of memories and provides the necessary mapping of address space ranges to allocated memory to achieve the lowest power consumption.

The memory allocation, which yields an optimal low power on-chip memory configuration that can be used for system synthesis, and the optimal mapping of elf objects\(^1\) to memory instances, which enables optimal linker scripts, makes our model a valuable tool for system synthesis.

2.1 Problem Statement
Given is a set \( M = \{m_i | i \in [1, m]\} \) of different memory types and an application characterized as a set of application profiles \( P = \{p_i | i \in [1, p]\} \) describing the runtime behavior of the application. The problem is to find an allocation \( \alpha \) of memory instances and a binding \( \beta \) that schedules each application profile to exactly one memory instance such that \( \alpha \) and \( \beta \) yield the lowest power consumption of all possible allocations and bindings while satisfying area and user-defined constraints.

2.2 Preliminaries and Conventions
For the sake of readability and clarity, this work focuses on read only program memory (ROM) storing the application binary. However, the model is not limited to ROM and can also be used for data memory (RAM) with small modifications, which we point out whenever applicable.

We write \( \mathbb{N}_0 \) to represent the natural numbers including zero. Accordingly \( \mathbb{N}_0^n \) is the n-dimensional vector space over \( \mathbb{N}_0 \). For vectors, we use the \( L_1 \)-Norm, i.e. \( \forall \nu \in \mathbb{N}_0^n : ||\nu||_1 = \sum_{i=1}^{n} |\nu_i| = \sum_{i=1}^{n} \nu_i \).

\(^1\)e.g. global or local symbols and functions.
3. DESIGN SPACE

The basic elements of the design space are the individual memory types $m_i$, and the actual design space is composed of all possible combinations of one or more memory types, with multiple selections of individual memory types being possible. That is, all allocations $\alpha \in \mathbb{N}_0^n$ with $\alpha_i$ being the number of instances of memory type $m_i$.

Each individual memory type $m_i$ is defined by its set of physical properties. For ROM, the relevant parameters are:

- the area (given in technology size, e.g. mm$^2$) of silicon required by a single instance of that ROM;
- the size of the memory given in kilobytes;
- the current required to read from the memory (given in $\mu A/\text{MHz}$);
- the deselect current, which is consumed when the memory is not accessed (in $\mu A/\text{MHz}$); and
- the standby or leakage current, which is permanently consumed (in $\mu A$).

Both read and deselect current are given with respect to the operational clock frequency of the system. In addition to the parameters above, write current must be specified when optimizing a selection of RAM instances.

Note that the interconnect is not directly part of the design space. However, the interconnect contributes to the overall power consumption depending on the total number of memory instances $\|\alpha\|$. This will be modeled in Sec. 4.

3.1 Design Space Constraints

Obviously, the design space, as described above, contains an infinite number of possible solutions. Many of these candidates represent useless solutions because they do not provide enough memory for the application or simply require too much area. Therefore, we apply initial constraints to prune the design space of all infeasible solutions. We differentiate between inherent constraints that are necessary for the sake of a correct application execution and user constraints to remove solutions that do not meet the designers needs.

The model features two inherent constraints. The first constraint ensures that the embedded software can be executed on the target system. The constraint guarantees that enough memory is provided to hold the whole binary of the embedded software. That is the case if the sum of the sizes of all memory instances is greater than the size of the compiled binary. For RAM, the amount of required memory must be derived from profiling the application. The second inherent constraint ensures the correctness with regards to real time requirements. For that, the model is constrained by the minimal clock frequency that ensures all real time requirements. Again, this value must be determined by profiling the application.

The design space can be further reduced by two user constraints. The power reduction of multiple small memories comes at the cost of a larger area requirement not only caused by the now necessary interconnect but also by the memory instances themselves. To avoid a prohibitive area overhead, the total required area can be constrained by the designer. The second user constraint simply allows to limit the total number of memory instances.

While these are only the basic constraints that we have incorporated into our model, the parametrized design makes it easy to add further application specific constraints, e.g. to limit the solutions to certain memory organizations with a fixed column width.

3.2 Application Profiles

The dominant part of a memory’s power consumption is caused by accessing it, which depends highly on the executed application. For this reason, the behavior of an application must be modeled to control the optimization process of the ILP solver. As indicated in our problem statement (cf. Sec. 2.1), we describe an application in terms of a set $P$ of application profiles, with $|P| \geq 1$. An application profile describes the periodic behavior for the whole application ($|P| = 1$) or for individual parts thereof ($|P| > 1$). The more application profiles are used to describe an application, the better solutions can be found by the ILP solver.

An exemplary application profile is shown in Fig. 1 (a). Each profile assumes a fixed period, which can be divided into two parts: an active phase, called duty cycle, and an idle phase. While memory can only be accessed during the duty cycle, it is not necessarily accessed throughout the entire duty cycle as indicated by the individual peaks in Fig. 1 (a). However, the power consumption does not depend on the individual points in time when the memory is accessed but only on the duration of those accesses. This allows us to simplify the application profile by combining the individual short memory accesses and modeling them as a fraction of the duty cycle called access probability. Fig. 1 (b) shows the resulting, simplified application profile.

While application profiles are designed to support the periodic nature often found in embedded applications, they can easily be used to describe non-periodic behavior by setting the period to the total application runtime.

For RAM, the application profiles must be extended to also account for write accesses. That is, the access probability is replaced with a read probability and a write probability,
The average power work, we focus on reducing the average power consumption. Power consumption is a function over time, and thus, reducing power consumption is not a well-defined term. In this section, we will first establish a componentwise definition of the required energy $E_p$ for a single application profile, from which we derive the combined energy consumptions of all application profiles.

Due to the periodic nature of application profiles, the total energy consumption of a single profile only depends on the energy consumed in one period. As explained in Sec. 3, the power consumption of a memory instance depends on whether it is currently being accessed or not. When reading from memory, standby current plus read current is consumed, and otherwise standby current plus deselect current is consumed. In consequence, a single period of a profile $p$ consumes the sum of its read energy, deselect energy, and standby energy:

$$E_p = E_{\text{read}} + E_{\text{deselect}} + E_{\text{STDBY}}$$ (1)

The three components $E_{\text{read}}$, $E_{\text{deselect}}$, and $E_{\text{STDBY}}$ are defined as:

$$E_{\text{read}} = d_i \cdot p_{r_i} \cdot I_r(f) \cdot V \cdot t_p$$ (2)

$$E_{\text{deselect}} = (1 - d_i \cdot p_{r_i}) \cdot I_d(f) \cdot V \cdot t_p$$ (3)

$$E_{\text{STDBY}} = I_s(f) \cdot V \cdot t_p$$ (4)

with duty cycle $d_i$, access probability $p_{r_i}$, and period $t_p$ of the profile; $I_r$, $I_d$, and $I_s$ are read, deselect, and standby current, in part as a function of the operational frequency; and $V$ is the voltage of the memory.

It can be immediately be seen from Eq. 2 to 4 that the respective average power consumptions do not depend on the period $t_p$ but only on the duty cycle and the access probability. That is, $P_{\text{Read}} = d_i \cdot p_{r_i} \cdot I_r(f) \cdot V$ and $P_{\text{deselect}}$ and $P_{\text{STDBY}}$ accordingly.

To determine the overall power consumption of an application, all application profiles must be considered. Even though the individual profiles can have arbitrary periods and thus execute with different frequencies, the overall average power still only depends on the individual duty cycles and access probabilities and not on the periods. Again, the total energy consumption is the sum of read, deselect and standby current: $E_{\text{tot}} = E_{\text{READ}} + E_{\text{DESEL}} + E_{\text{STDBY}}$ with $E_{\text{READ}}$, $E_{\text{DESEL}}$, and $E_{\text{STDBY}}$. Note that we use upper-case indices for the combined energies and lowercase indices for individual profiles. The components are defined as follows.

$$E_{\text{READ}} = \sum_{i=1}^{P} E_{\text{read}_i} \cdot \frac{T}{t_p}$$ (5)

$$= \sum_{i=1}^{P} d_i \cdot p_{r_i} \cdot I_r(f) \cdot V \cdot \frac{T}{t_p}$$ (6)

$$= T \cdot \sum_{i=1}^{P} d_i \cdot p_{r_i} \cdot I_r(f) \cdot V$$ (7)

The combined deselect energy cannot be calculated as the sum of the individual deselect energy because deselect energy is only consumed, when no profile is accessing the memory.

$$E_{\text{DESEL}} = \left( T - \sum_{i=1}^{P} d_i \cdot p_{r_i} \cdot \frac{T}{t_p} \right) \cdot I_d(f) \cdot V$$ (8)

$$= \left( T - T \cdot \sum_{i=1}^{P} d_i \cdot p_{r_i} \right) \cdot I_d(f) \cdot V$$ (9)

$$= T \cdot \left( 1 - \sum_{i=1}^{P} d_i \cdot p_{r_i} \right) \cdot I_d(f) \cdot V$$ (10)

$$E_{\text{STDBY}} = T \cdot I_s \cdot V$$ (11)

Again, it can immediately be seen that the average power $P_{\text{avg}} = E_{\text{tot}}/T$ does not depend on the individual periods but only on the individual duty cycles and access probabilities, i.e.:

$$P_{\text{READ}} = \sum_{i=1}^{P} d_i \cdot p_{r_i} \cdot I_r(f) \cdot V$$ (12)

$$P_{\text{DESEL}} = \left( 1 - \sum_{i=1}^{P} d_i \cdot p_{r_i} \right) \cdot I_d(f) \cdot V$$ (13)

$$P_{\text{STDBY}} = I_s \cdot V$$ (14)

For RAM, the average power is increased by the write power $P_{\text{WRITE}}$ and the formula for $P_{\text{DESEL}}$ must be amended.

$$P_{\text{WRITE}} = \sum_{i=1}^{P} d_i \cdot p_{w_i} \cdot I_w(f) \cdot V$$ (15)

$$P_{\text{DESEL}} = \left( 1 - \sum_{i=1}^{P} d_i \cdot (p_{r_i} + p_{w_i}) \right) \cdot I_d(f) \cdot V$$ (16)

4. **ILP MODEL**
Table 1: ILP variables with description and type.

<table>
<thead>
<tr>
<th>Var</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>Allocation of memory types</td>
<td>$N_0^{[M]}$</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Binding of app profiles to mem types</td>
<td>${0,1}^{[P] \times [M]}$</td>
</tr>
</tbody>
</table>

In this section, we introduce the missing parts to complete the ILP model. We show how memory allocation and application binding are modeled and how they are used to constrain the ILP model. Furthermore, we detail the calculation of the average power consumption for a given allocation and binding, introduce our the interconnect model and, before we establish the target function to be minimized by the ILP solver. For the sake of easy lookup, Tables 1 and 2 list all variables and parameters used in our model.

To enable a split memory configuration, an interconnect fabric is required. With the number of connected memories, this fabric grows in complexity which translates to growing power consumption and area requirements. Since this growth is usually non-linear, we model both power consumption and area requirements. Since this growth is usually non-linear, we model both power consumption and area requirements. That is, if no values are provided for a certain piecewise definition enables incomplete specifications of power and area. That is, if no values are provided for a certain interconnect size, power and area will be predicted linearly. Let $P_F(n)$ and $A_F(n)$ denote the functions for the interconnect fabric’s power consumption and area requirements, respectively.

### 4.1 Memory Allocation

As defined in Sec. 2.1, one part of the ILP problem is finding an allocation $\alpha \in N_0^{[M]}$ with $M$ being the set of available memory types and $\alpha_i$ representing the number of instances of memory type $m_i$. Let $m = |M|$ be the number of memory types. In the ILP, this directly translates to the allocation being a vector alpha of non-negative integers:

$$\forall i \in [1, m]: \alpha_i \geq 0$$

(17)

The allocation $\alpha$ is used to enforce three of the constraints introduced in Sec. 3.1. To limit the total number of allowed memory instances, we introduce the parameter $\text{mems}_{\text{max}}$ in the ILP model, such that $||\alpha||_1 \leq \text{mems}_{\text{max}}$. This translates to the following constraint:

$$\sum_{i=1}^{[M]} \alpha_i \leq \text{mems}_{\text{max}}$$

(18)

Accordingly, the parameter $\text{area}_{\text{max}}$ limits the total area available for all memory instances and the interconnect. Let $A \in N_0^{[M]}$ represent the area requirements of the individual memory types, then the area can be constrained as follows:

$$A_F(||\alpha||_1) + \alpha \cdot A \leq \text{area}_{\text{max}}$$

(19)

$$\Leftrightarrow A_F(\sum_{i=1}^{[M]} \alpha_i) + \sum_{i=1}^{[M]} \alpha_i \cdot A_i \leq \text{area}_{\text{max}}$$

(20)

To ensure that only fast enough memories are used, the maximum operational frequency of the allocated memories is compared with the scalar parameter $f_{\text{min}}$. Let $F \in N_0^{[M]}$ be the vector of maximum supported memory frequencies, then the selection is constrained by Eq. 21. If a memory does not support the required frequency (i.e., $F_i < f_{\text{min}}$), the ILP solver is forced to set $\alpha_i = 0$ to fulfill Eq. 21 thus prohibiting the allocation of too slow memories.

$$\forall i \in [1, |M|]: \alpha_i \cdot F_i \geq \alpha_i \cdot f_{\text{min}}$$

(21)

### 4.2 Application Binding

We represent the binding as a binary matrix $\beta \in \{0,1\}^{[P] \times [M]}$, with the elements $\beta_{ij}$ indicating whether application profile $i$ has been mapped to memory type $j$ ($\beta_{ij} = 1$) or not ($\beta_{ij} = 0$).

To ensure a correct solution, each application profile must be bound to exactly one memory type. With $\beta_{ij} \in \{0,1\}$ and Eq. 22 each row $j$, representing the binding of profile $j$, is constrained to have a single occurrence of the value one.

$$\forall i \in [1, |P|]: \sum_{j=1}^{[M]} \beta_{ij} = 1$$

(22)

Furthermore, enough instances of a given memory type must be provided fitting all application profiles mapped to that memory type. Let $\sigma^P \in N_0^{[P]}$ and $\sigma^M \in N_0^{[M]}$ be the vectors representing memory required by application profiles and memory provided by memory types respectively. With this, we specify the memory requirements as follows.

$$\forall j \in [1, |M|]: \sum_{i=1}^{[P]} \beta_{ij} \cdot \sigma^P_i \leq \alpha_j \cdot \sigma^M_j$$

(23)
Note that neither Eq. 22 nor Eq. 23 explicitly binds application profiles to memory instances but only to memory types. This might result in an allocation and binding, where an application profile does not fit entirely in a single memory instance (e.g., 2 mem instances of size 3KB and 3 profiles requiring 2KB each). However, this does not impose a limitation nor affects the correctness of our model. In such a case, the instances of a given memory type must be arranged linearly and gapless with respect to the address space.

### 4.3 Optimization Goal

To calculate the actual power consumption of a given configuration, the power consumption formulae established in Sec. 3.3 must be amended to consider memory allocation and application binding. The read power consumption of a given memory type $j$ can be calculated solely using the binding:

$$P_{\text{read},j} = \sum_{i=1}^{n} \beta_{ij} \cdot d_i \cdot p_{ri} \cdot I_{s,j}(f) \cdot V$$  \hspace{1cm} (24)

For memory type $j$, deselected power is consumed whenever none of the mapped applications is accessing the memory. Therefore, deselected power depends on both, the applications bound to it and the number of allocated memories. The latter replaces the constant value 1 in Eq. 13.

$$P_{\text{desel},j} = \left( \alpha_j - \sum_{i=1}^{n} \beta_{ij} \cdot d_i \cdot p_{ri} \right) \cdot I_{s,j}(f) \cdot V$$  \hspace{1cm} (25)

The standby power must be simply weighed by the number of allocated instances.

$$P_{\text{stdby},j} = \alpha_j \cdot I_{s,j} \cdot V$$  \hspace{1cm} (26)

With the augmented power consumption for the individual components (Eq. 24 to 26) and the power consumption function $P_T(n)$ of the interconnect fabric, we can now postulate the cost function to be minimized.

$$P_{\text{tot}} = P_T(n) + \sum_{j=1}^{M} \left( P_{\text{read},j} + P_{\text{desel},j} + P_{\text{stdby},j} \right)$$  \hspace{1cm} (27)

$$= P_T(n) + \sum_{j=1}^{M} \left( \alpha_j \cdot I_{s,j} + \sum_{i=1}^{n} \beta_{ij} \cdot d_i \cdot p_{ri} \cdot I_{s,j}(f) \cdot V \right)$$  \hspace{1cm} (28)

The ILP solver’s goal is to minimize $P_{\text{tot}}$. Respecting the given constraints, the solver chooses suitable variable assignments for allocation $\alpha$ and binding $\beta$ such that no other candidate solution has a lower power consumption. This solves our problem statement as defined in Sec. 2.1.

### 5. SYSTEM SYNTHESIS

In this section, we briefly want to illustrate how the ILP model can be incorporated into a system synthesis flow optimized for low power. Since the model not only yields the optimal allocation of memories but also provides an optimal mapping of profiles to memory instances, our process is ideal for hardware/software codesign. As illustrated in Fig. 3, the general synthesis flow can be divided into a sequence of four steps: cross compilation of the application (a), extraction of application profiles (b), ILP solving (c), optimization of hardware and software domains (d,e).

In the first step, the application’s source files are cross compiled for the target architecture, and the resulting object files are linked to an executable binary. The binary is then analyzed to setup a list of application profiles. For each symbol in the binary, the name, starting address, and size are extracted. Symbols are either functions, global or local objects. Objects not only represent global variables but also stack and heap, for which we only extract the initial sizes as their eventual sizes is determined at runtime.

In the second step, an ISS simulator is used to complete the previously setup application profiles. The simulator tracks all instruction fetches to determine the number of executed instructions and cycles spent in each function to establish its duty cycle. The function’s period can either be determined by expert knowledge or be expressed relative to the overall application runtime if the application is not of periodic nature. The simulator also tracks all read and write accesses. If the accessed address falls into the range of an object, it is accounted for that object, otherwise, either the stack or heap are credited and the stack or heap size is updated accordingly.

The instruction memory profiles and the data memory profiles are then fed to the ILP solver, which solves two independent ILP problems, one for instruction memory and one for data memory. For the given set of parameters, the ILP solver produces an optimal selection and number of memory types as well as a mapping of symbols to memory instances (cf. Sec. 4), which can be used to optimize the hardware and software domains of an embedded system.

For the HW part, the memory array is built by instantiating the previously determined number of memory IP blocks and by synthesizing a parametrized interconnect based on the memory blocks’ address ranges. For the SW part, it must be ensured that symbols are located in the address ranges of the memory blocks they have been mapped to. For this,
the ILP’s symbol mapping is used to create a linker script with which the object files created in the first step are re-linked to generate an executable with an optimal address space layout.

6. EVALUATION

We evaluated the applicability of our approach using the Embedded Microprocessor Benchmark Consortium (EEMBC) MultiBench benchmark suite. To keep the simulation and profiling environment simple, we refrained from using benchmarks requiring file I/O, which left us with the five benchmarks. IP reassembly reflects the work of a network router when reassembling fragmented packets; IP check performs IP header validation; TCP base implements the most processing-intensive parts of the TCP protocol; MD5 performs checksum calculation; and Huffman implements the decoding algorithm commonly found in image and video compression standards.

To extract profiling data for instruction memory and data memory, we cross compiled the applications for the PowerPC architecture. The application execution was then simulated using the ppc405 instruction set simulator (ISS) from the SoClib platform [12], which we improved with logging capabilities as described in the previous Section (cf. Sec. 5).

We used CACTI 6.5 [10] to generate a set of 74 different memory types ranging from 512 Bytes to 8 MBytes. All memory characteristics were created for the 45nm technology node, based on the low standby power transistor models provided by the ITRS. For each memory size, multiple memory types with different number of sub banks were created.

To acquire power values for the interconnect, we designed a parametrized, multiplexer-based VHDL model. To minimize power consumption of inactive memories, the interconnect implements transparent latches to keep address lines from toggling. The model has been synthesized using the NanGate 45nm Open Cell Library for different number of memories and address ranges. Power simulations using actual memory access traces have been performed for the individual synthesized interconnects, and the resulting power consumption values form the piecewise linear function modeling the interconnect in the ILP.

For all applications, we performed separate optimizations for instruction memory and data memory. For both, the ILP model was solved 8 times with different limits for the maximum number of partitions, ranging from 1 to 8 instances. The resulting optimal power consumptions have been plotted in Fig. 4 with instruction memories on top and data memories below. For instruction memory, it can be clearly seen that already a split memory configuration of two instances causes a drastic power reduction (80.5% for IP reassembly). Increasing the number of partitions shows slight improvements for up to 4 memories (82.9% for IP reas.), but no application can benefit from more than 5 or more instances. With a still very good power reduction of 60.2%, the MD5 benchmark benefits the least from split memories.

For data memory, the power reduction improves more gradually, and all applications can benefit from up to 8 memory instances, with the 8 memory configuration further reducing the power consumption by up to 5% over a 7 memory configuration. The average power reduction of data memory is 60.8%, with a minimum of 37.8% for Huffman decoding and a maximum of 73.2% for IP check. While still yielding good results, splitting data memory was not as beneficial as splitting instruction memory, which we attribute to the large heap requirements of the applications. It is worth noting that a 6 or 7 memory configuration is not favorable for the IP reassembly benchmark, but allowing an 8 memory configuration eventually reduces the power consumption by another 5%. This highlights that the optimal number of memory partitions cannot be known in advance and should be a free variable in the optimization process.

To give better insight into power consumptions of the individual memory instances, Tab. 3 and Tab. 4 show detailed power consumption break downs for instruction and data memory respectively of two exemplary applications. For both applications, the monolithic solution is compared against the optimal solution found by the ILP solver. For the instruction memory of the IP reassembly benchmark (Tab. 3), it can be seen that only 9 of 241 functions are responsible for 95.1% of all instruction fetches. Putting these 9 functions into a low-power memory of only 512 Bytes size achieves the biggest reduction in power consumption. For the Huffman decoder, the situation is even more ex-

![Figure 4: Average Instruction Memory (top) and Data Memory (bottom) Power Consumptions for Different Benchmarks, each with Varying Limits for the Number of Allowed Memory Instances.](image-url)
8. REFERENCES

9. CONCLUSION

Table 3: Instruction Memory Power Consumption Details

Table 4: Data Memory Power Consumption Details

treme, where 4 functions account for 99.9% of the instruction fetches. This explains the large benefit that is already achieved when splitting into 2 memories.

For data memory (Tab. 4), the causes for power reduction are not as evident. This is because a large amount of memory reads and writes are target the very large heap (up to 4 Mbytes). Thus, the reduced power consumption is mainly achieved by splitting the memory into more sub-banks by spreading the heap across 5 smaller memory instead of one large memory. This effectively increases the number of sub-banks from 16 to 40, thus transparently segmenting the heap. The ILP model further reduces power by minimizing the total memory size and by assigning frequently accessed objects to smaller memories whenever possible. This is well evident in the MD5 case where the smallest memory is still accounted for about 30% of all reads and writes.

7. CONCLUSION

In this paper, we have provided a mathematical model to determine an optimal on-chip memory organization with respect to low power. The model is highly flexible allowing applications to be modeled with different degrees of precision and works with a large list of memory types. Achieving power savings of up to 80% for instruction memory and 73% for data memory in a set of industrial grade benchmarks proves the benefits of our model. Furthermore, we showed, how our model can be incorporated into an automated synthesis making it a valuable tool in low power HW/SW code-sign.

8. REFERENCES


