Efficient Automated Code Partitioning for Microcontrollers with Switchable Memory Banks

MICHAL CISZEWSKI and KONRAD IWANICKI, University of Warsaw

Switching active memory banks at runtime allows a processor with a narrow address bus to access memory that exceeds ranges normally addressable via the bus. Switching code memory banks is regaining interest in microcontrollers for the Internet of Things (IoT), which have to run continuously growing software, while at the same time consuming ultra-small amounts of energy. To make use of bank switching, such software has to be partitioned among the available banks and augmented with bank-switching instructions. In contrast to the augmenting, which is done automatically by a compiler, today the partitioning is normally done manually by programmers. However, since IoT software is cross-compiled on much more powerful machines than its target microcontrollers, it becomes possible to partition it automatically during compilation.

In this article, we thus study the problem of partitioning program code among banks such that the resulting runtime performance of the program is maximized. We prove that the problem is $\mathcal{NP}$-hard and propose a heuristic algorithm with a low complexity, so that it enables fast compilation, and hence interactive software development. The algorithm decomposes the problem into three subproblems and introduces a heuristic for each of them: (1) Which pieces of code to partition? (2) Which of them to assign to permanently mapped banks? and (3) How to divide the remaining ones among switchable banks? We integrate the algorithm, together with earlier ones, in an open-source compiler and test the resulting solution on synthetic as well as actual commercial IoT software bases, thereby demonstrating its advantages and drawbacks. In particular, the results show that the performance of partitions produced by our algorithm comes close to that of partitions created manually by programmers with expert knowledge on the partitioned code.

CCS Concepts: **•Computer systems organization →Embedded software; •Software and its engineering →Compilers;**

Additional Key Words and Phrases: bank switching, code banking, trampoline function, switchable memory, partitioned memory, code partitioning, heuristic algorithm, microcontroller, MCU, Internet of Things, IoT

ACM Reference format:
Michal Ciszewski and Konrad Iwanicki. 2017. Efficient Automated Code Partitioning for Microcontrollers with Switchable Memory Banks. *ACM Trans. Embedd. Comput. Syst.* 16, 4, Article 114 (May 2017), 26 pages.
DOI: 10.1145/3055511

1 INTRODUCTION

The Internet of Things (IoT) vision, in which surrounding physical objects are fully-fledged members of the global network [42], is highly demanding for embedded compilers. On the one hand, the devices built into the physical objects to give these objects sensing, actuation and computing

K. Iwanicki was supported by the Polish Ministry of Science and Higher Education with a scholarship for outstanding young scientists.

Authors' addresses: M. Ciszewski and K. Iwanicki, Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warszawa, Poland. E-mails: mc305195@students.mimuw.edu.pl, iwanicki@mimuw.edu.pl.

DOI: 10.1145/3055511

capabilities, and to connect them to the Internet have to be tiny, unobtrusive, inexpensive, and ultra-low power. This entails heavy constraints on the resources of such wireless micro-devices, particularly on their memory and processing capabilities. On the other hand, the devices have to run software implementing not only their core application logic but also a lot of extra functionality, notably a suite of standardized protocols at several network layers, so that each device can appear to the outside world as an ordinary Internet host. The amount of software necessary for supporting these protocols as well as other nonfunctional issues, like security, manageability, and interoperability, is significant already and is likely to grow in the future [20]. In this light, an important role of embedded compilers is to facilitate cramming large volumes of software into extremely resource-constrained microcontrollers of IoT devices.

A major problem in this respect is the memory constraints of the microcontrollers. More specifically, while a few kilobytes of data memory are typically sufficient for a program comprising an application and a complete operating system with an IPv6 stack [11, 18], a binary image of such a program can already today exceed a hundred of kilobytes of code memory. What is more, in contrast to data memory (RAM), which is commonly expected to be low [17], the requirements for code memory (ROM) are likely to grow even further with subsequent emerging standards and nonfunctional issues. This is particularly motivated by the fact that ROM, in contrast to RAM, draws virtually no power in sleep. Such a growing demand on code memory, however, is problematic because many popular microcontrollers for IoT utilize architectures with 16-bit code address buses, which normally preclude addressing more than 64 kilobytes of code. There are examples where this has already been a serious practical issue [10, 19, 22].

Microcontroller manufacturers approach the problem in various ways, one of them being to employ *bank switching* for code memory. With bank switching, the address space of a microcontroller can be extended arbitrarily without widening the address bus. This is achieved by partitioning the physical memory into so-called *banks*, such that at any time only a subset of the banks can be accessed; in contrast, to access another bank, the bank has to be activated explicitly—*switched*—instead of some currently active one. The main advantage of this approach is that, being decoupled from the address space size, the bus can stay narrow. As a result, the microcontroller's die size and power consumption can remain ultra low, which is crucial for IoT applications. A major disadvantage is in turn that the code has to be partitioned into the available ROM banks and augmented with bank switching instructions. The quality of the partition is vital as it determines the code-size and runtime overheads.

Therefore, considering the present and expected volumes of IoT software repositories, an embedded compiler should assist programmers in dividing their code among banks. Ideally, it should automatically partition the code, such that its size and runtime overheads are minimized. In practice, however, as we discuss in more detail in the next section, even though bank switching is a classic technique, efficient automated partitioning of code into banks has started gaining interest only recently. Its support in embedded compilers for IoT microcontrollers thus leaves room for improvement.

In this article, we present a novel approach to automated code partitioning for microcontroller architectures with switchable ROM banks. We consider architectures with eager switching of (a subset of) banks, that is, ones in which a switchable bank is changed immediately whenever the value of an appropriate microcontroller register changes. Our goal is to automatically generate partitions that result in as few bank switches at runtime as possible, so that the performance of a partitioned program can be high. We formalize the problem and prove that it is $\mathcal{NP}$-hard, which implies that, unless $\mathcal{P} = \mathcal{NP}$, only heuristic algorithms are practical. With this observation, we devise an algorithm with a low complexity, so that it generates partitions quickly, thereby allowing for interactive software development. The algorithm decomposes the problem into three subproblems and provides a heuristic solution for each of them: (1) Which pieces of code to partition? (2) Which of them to assign to permanently mapped banks? and (3) How to divide the remaining ones among switchable banks? We implement the algorithm in an open-source compiler for IoT software, together with other existing heuristic algorithms. To the best of our knowledge, the compiler is the first one to support such extensive out-of-the-box program partitioning. We then use the compiler to empirically compare the algorithms on synthetic as well as commercial IoT software. The comparison demonstrates the advantages and drawbacks of our solution.

The rest of the article is organized as follows. Section 2 surveys related work. Section 3 formalizes the problem and analyzes its complexity. Section 4 introduces our algorithm. Section 5 describes its implementation. Section 6 evaluates the algorithm and compares it with existing solutions. Finally, Section 7 concludes.

2 RELATED WORK

As mentioned previously, bank switching is a classic technique of extending a device's address space without widening its address bus. Its origins can be traced back to minicomputers [5]. It was then employed also in microcomputers, such as BBC Micro [6], where it allowed for using so-called sideways (paged) ROM banks, ZX Spectrum +3 [1], where it was necessary for fully utilizing the available 128 KB of RAM and 64 KB of ROM within a common 64 KB address space, and Commodore 64 [43], where its goal was to overcome the overlap between RAM and ROM sections, to name just a few examples. Likewise, bank switching underlay the Expanded Memory Specification (EMS) for IBM-compatible computers [27] and its successor, the eXtended Memory Specification (XMS) [28].

At that time, however, efficient automated partitioning of code into banks was hardly feasible. This was because computing a partition of a program that maximizes the program's runtime performance involves analyzing a graph comprising potentially all instructions in the program, thereby requiring a considerable amount of RAM. Moreover, the problem itself is also computationally expensive, as we prove in Section 3. Consequently, for the early bank-switched architectures, program partitioning was typically done manually by the programmers, and its quality depended on their expertise.

Bank switching was then abandoned in personal computers in favor of architectures with wider memory buses. Adopting new architectures with 32-bit buses, such as ARM Cortex-M3 [3], Cortex-M0 [2], or the more recent Cortex-M0+ [4], is thus also one of the approaches taken by microcontroller manufacturers to overcome the problem of insufficient code address space for IoT software. Apart from the larger address space, another advantage of this approach is a potentially more efficient instruction set. The drawback, however, is that, due to the additional complexity, these architectures are often more power-consuming [23]. Moreover, switching to these novel architectures entails considerable financial costs and problems with porting volumes of existing software, developing new device drivers, and thoroughly testing everything [15], not to mention reworking hardware designs. In general, completely switching a microcontroller architecture may be a radical undertaking in established companies.

Therefore, another approach taken by microcontroller manufacturers is extending the address bus in existing architectures, albeit as much transparently as possible. An example is Texas Instruments' MSP430X architecture [39], whose goal is to widen the code address bus from 16 to 20 bits compared to MSP430, while maintaining backward compatibility and low power consumption. However convenient for users, such an approach has its drawbacks as well, such as different pointer widths for data and code, greater stack usage, especially during interrupts, and a few more [32]. In other words, full transparency is not achieved.

For these reasons, bank-switched architectures are yet another option offered by microcontroller manufacturers, often together with the previous two. Such architectures can be simple and ultra low-power but they completely give up transparency with respect to the address bus size, and hence require managing ROM bank accesses in software. Today, however, we are far better equipped to automate such management. This is because off-the-shelf machines, on which the embedded IoT software is developed, offer memory and computing capabilities that are larger by at least a factor of 10^5 and 10^3, respectively, compared to the target bank-switched IoT microcontrollers, on which the software has to run. As a result, automating bank switching can be delegated to embedded cross-compilers. In particular, they will have enough memory and processing power to store and analyze an entire image of a program for a microcontroller, so as to partition it into banks and insert the necessary bank-switching instructions.

To date, however, little work has been done on compiler-driven automated bank switching for code memory. More specifically, some embedded compilers, such as GCC for MH68HC11 [7] or SDCC for MCS51 [12], offer memory models that allow for compiling bank-switched programs. The granularity of switching is that of a single function: each function must fit entirely in some bank, and a bank switch can take place only upon a function call and return. To this end, either individual functions must be marked by a programmer as callable between banks or all of them are marked automatically by the compiler. In addition, the programmer must assign the functions to banks with special pragmas. The compilers in turn generate bank-switching instructions for so marked functions but make no attempts to optimize the marking and allocation, in some cases even not verifying whether they are correct.

In contrast, completely automated marking and partitioning of functions into banks have started gaining interest only recently. To start with, due to the lack of this functionality in embedded compilers, the developers of Contiki, a popular open-source operating system for IoT devices, created a custom code partitioning tool for banked-switched architectures based on 8051 [37], such as the CC253x system-on-chip (SoC) solutions from Texas Instruments, Inc. [40]. The tool assumes that all functions are marked as callable between banks and uses a bin packing heuristic to fill up the available banks before the code is linked into a final binary image. Li et al. [26], in turn, devised a heuristic that tries to reduce the size of the program binary. To this end, based on a control flow graph analysis, it greedily assigns functions to banks to minimize the number of bank change instructions inserted to the binary. Mengting et al. [30] later formulated the same code-minimization problem in terms of integer linear programming and presented an algorithm that combines a heuristic for the Minimal k-Cut problem, to obtain an initial, potentially incorrect partition, with a tabu search heuristic, to gradually correct the solution.

The heuristics by Li et al. and Mengting et al. both assume architectures with *lazy* bank switching, such as in PIC16F7X [31], where the next bank to activate can be chosen (and changed potentially many times) without actually using the memory in the bank. This is done via a bank selection register that conceptually extends the code address bus but is relevant only for specific instructions, such as function calls. In contrast, the aforementioned CC253x SoCs, targeted by the bin packing tool, employ *eager* bank switching, where a bank selection register points at a currently active bank. Consequently, any change to the value of the register causes an immediate bank switch, that is, dynamic memory remapping. Lazy and eager bank switching are thus two different models with their own advantages and drawbacks. In particular, it is typically a function call that accumulates the runtime overhead in eager switching, whereas in lazy switching this overhead is due to bank selection instructions, which are dispersed in the code. All in all, due to such differences, the corresponding goals of and approaches to function marking and partitioning in the two models may differ.

In contrast to the aforementioned heuristics, our work targets the eager bank switching model. We prove that in this model the problem of automated function marking and efficient partitioning is $\mathcal{NP}$-hard. We introduce a novel heuristic that improves over the aforementioned bin packing heuristic by, first, automating the function marking process and, second, minimizing also the runtime overhead incurred by function partitioning. We empirically show that our approach outperforms the bin packing heuristic. Similarly, we adapt the heuristics by Li et al. and Mengting et al. to the eager model to compare them with our solution. We are not aware of any previous similar results for the eager bank switching model. What is more, the presented work also opens up a number of possibilities for future research.

Finally, from an even broader perspective, architectures with ROM bank switching belong to a large class of memory-partitioned architectures, which also includes, among others, architectures for digital signal processing (DSP) or graphical processing units for general-purpose computing (GPGPUs), in which the goal is to optimize access to physical RAM banks. A considerable body of research regarding these architectures aims to maximize parallel data access. For example, by enabling single-instruction parallel RAM access on dual-bank DSPs, one can improve the performance of embedded programs [24, 33, 35, 45]. In contrast, avoiding access conflicts to RAM banks on GPUs is crucial in high-performance GPGPU [8, 34]. A similar effect can be achieved with appropriately using heterogeneous memory, such as scratchpad vs. external RAM in embedded systems [25, 41] or the memory hierarchy in GPGPUs [8, 34]. These problems are different from the ones we consider in this article, though. Likewise, while there are also solutions for accessing RAM banks beyond what would be allowed with a given address bus [36], such solutions target data memory, with different problems and different optimization techniques than in the case of efficient automated code partitioning for IoT micro-devices.

3 PROBLEM FORMALIZATION AND ANALYSIS

We start presenting our work by discussing in more depth the problem it addresses. First, we explain the eager bank switching model, with the aforementioned CC253x SoCs [40] serving as running examples. Second, we formalize the model and the problem of efficiently marking and partitioning functions into banks. Third, we prove that the complexity of the problem justifies heuristic solutions.

3.1 Eager Bank Switching

In architectures with banked-switched code memory, the *program counter register*, pointing the currently executed instruction, is narrow, like the bus itself. In effect, at any time, only a fraction of the program is addressable by the microcontroller through the program counter. Therefore, to enable running the entire code, the addressable space is divided into p *areas* of fixed sizes, and likewise, the ROM is divided into k *banks* of corresponding sizes, where $p \leq k$. At any time, only up to p of the k ROM banks are mapped to some of the areas—are *active*—and can thus be accessed via the program counter. Some of these banks can be mapped permanently to particular areas, so that the code they store can always be accessible or can have certain absolute addresses, such as the code of interrupt handlers or bootloader traps. For each other area, in turn, the active bank can be switched at runtime by a *bank mapping register* corresponding to this area. In the eager bank switching model, any change to the bank mapping register for an area immediately changes the ROM bank mapped to this area.

As an illustration, consider the memory architecture of the aforementioned CC253x SoCs, depicted in Fig. 1. In CC253x, the program counter is 16 bits wide, which implies that it can address only 64 KB of program code, whereas the ROM has 128 or 256 KB, depending on a particular microcontroller. To enable utilizing the entire ROM, the addressable space is divided into two areas of 32 KB each,

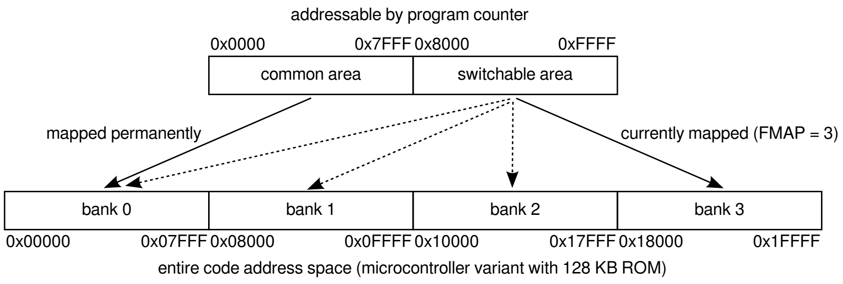

Fig. 1. The banked-switched memory architecture of CC253x SoCs from Texas Instruments, Inc. [40].

and likewise, the ROM is divided into 4 or 8 banks of 32 KB. Bank 0 is permanently mapped to the lower area, so-called *common area*, from address `0x0000` to address `0x7fff`, and is thus referred to as the *common bank*. Any of the remaining 3 or 7 banks (but also bank 0) can in turn be mapped to the upper area, so-called *switchable area*, from address `0x8000` to address `0xffff`. The active bank in the switchable area is controlled by an 8-bit bank mapping register, `FMAP`, of which only 3 bits are meaningful. Any change to `FMAP` immediately changes the bank mapped to the switchable area. The effective code address space of a program thus stretches from address `0x00000` to address `0x1ffff` or `0x3ffff`, depending on whether a particular microcontroller features 4 or 8 ROM banks, respectively.

A microcontroller executes the instruction pointed by the program counter, that is, an instruction in a currently mapped bank. To execute an instruction in an unmapped bank, the bank must be first mapped to some area in place of the bank currently mapped to that area, switched. Although, as we propose in Section 7, such switching can be done at a finer granularity, today's compilers do it upon function invocations. The consequences are twofold. First, while some subtle exceptions are possible, in principle, each function must fit entirely in one bank. Second, an invocation of function `foo` from within function `bar` can incur a different runtime overhead if `foo` resides in the same bank as `bar` or in a permanently mapped bank than if `foo` resides in a different bank: in the former case, the invocation can be realized with a single `call` instruction of the microcontroller, whereas in the latter, one or more bank switches may be necessary. The number of inter-bank calls should thus be kept as low as possible.

For example, the aforementioned popular C compiler for CC253x, SDCC [12], implements an inter-bank function invocation with calls of two so-called *trampoline functions*. The `banked_call` trampoline function saves the active bank, extracts the bank number and in-bank offset from the address of the target function, changes the bank active in the switchable area, and jumps to the in-bank offset of the target function. The `banked_ret` function, in turn, restores the previously active bank and returns back to the address of the invocation. A trampoline-based inter-bank invocation is thus much more expensive than an intra-bank or common-bank invocation. In particular, in contrast to 1 `lcall` instruction for an intra- or common-bank invocation, the `banked_call` trampoline function itself contains 11 microcontroller instructions, not to mention additional instructions necessary to invoke it in place of the target function. As a side note, the trampoline functions must reside in the common bank.

To enable running a program by a bank-switched microcontroller, the functions constituting the program must be partitioned into the available ROM banks, such that each function fits entirely into the bank it is assigned to. Moreover, all functions that are assigned to banks not permanently mapped and that are invoked by other functions from different banks must be marked as callable between banks, so that the compiler can generate inter-bank invocation code for their calls. Since inter-bank function calls incur much more runtime overhead than intra- or common-bank ones, the partition must strive to minimize the number of such calls at runtime: functions calling each other frequently should preferably be stored in the same banks, and moreover, functions called frequently globally should be assigned to permanently mapped banks. As marking is done trivially given a partition, the rest of this article thus studies the problem of automated efficient partitioning of program functions into banks.

3.2 Formal Problem Model

Let us formally model the problem. To this end, we assume the following notation for natural numbers throughout the rest of this article: $\mathbb{N} = \{0, 1, 2, \ldots\}$; $\mathbb{N}^+ = \mathbb{N} \setminus \{0\} = \{1, 2, \ldots\}$; $\mathbb{N}_k = \{0, 1, \ldots, k-1\}$.

Let $G = (V, E)$ be a directed graph for a program given as input. Each of its vertices, $v \in V$, corresponds to a single function from the program, and there exists an edge, $e = \langle v, u \rangle \in E$, for each call of the function corresponding to vertex u from within the body of the function corresponding to vertex v. Note that E can thus be a multiset if some function contains multiple calls of another function, and consequently, G can be a multigraph. All in all, G is often referred to as a *function call graph* for the program.

Each vertex $v \in V$ is given a weight, $\sigma(v) > 0$ ($\sigma : V \to \mathbb{N}^+$). The weight denotes the code size in bytes of the function corresponding to v in the binary image of the program. For simplicity, in this section, we assume that this size does not change when the functions called by this function are assigned to different banks.

Similarly, each edge $e = \langle v, u \rangle \in E$ has a weight, $\omega(e) \geq 0$ ($\omega : E \to \mathbb{N}$). The weight reflects how frequently the function corresponding to v invokes the given call instruction for the function corresponding to u: the larger it is, the more frequent the calls. The weight can be measured, for instance, as the number of such calls in a given time period, so that depending on how the period is selected, slightly different optimization goals may be possible. From the perspective of the complexity analysis, however, it is not important what the edge weights denote precisely but only that they reflect some runtime frequency of the corresponding function calls to which the minimized runtime overhead is proportional. Note also that we could now collapse all edges from v to u into a single edge with an aggregate weight, which in effect would turn E from a multiset into a plain set. We do not do this to keep our model general: the current definition of E preserves the original structure of the call graph even with edge weights.

In the partitioning problem, we are given k banks, numbered from 0 to $k-1$, of which $k' \in \mathbb{N}_k$ first ones are permanently mapped, and we have to assign each of the program's functions to one of these banks. Our goal is thus to partition the set of all vertices, V, into at most k subsets $V_0^\pi, \ldots, V_{k-1}^\pi$, so that $V_0^\pi \cup \cdots \cup V_{k-1}^\pi = V$ and $V_i^\pi \cap V_j^\pi = \varnothing$ for any $i \neq j$. Formally, we represent such a partition by $\pi : V \to \mathbb{N}_k$, such that, for a given vertex, $v \in V$, $\pi(v)$ is equal to the index of the bank to which the function corresponding to v has been assigned. In this notation, we thus have $V_i^\pi = \{v \in V | \pi(v) = i\}$ for all $i \in \mathbb{N}_k$.

Some functions, such as interrupt handlers or bootloader traps, may be pre-assigned to specific banks. We denote the subset of vertices pre-assigned to bank i as V_i^α, and all such vertices as $V^\alpha = V_0^\alpha \cup \cdots \cup V_{k-1}^\alpha$. By analogy to π, we formally represent such an initial partition as $\alpha : V^\alpha \to \mathbb{N}_k$, so that $V_i^\alpha = \{v \in V^\alpha \subseteq V | \alpha(v) = i\}$. We thus have to partition only the remaining vertices, $V \setminus V^\alpha$. This implies that in the final partition we must not change the initial bank assignment, that is, we must have $\pi(v) = \alpha(v)$ for all $v \in V^\alpha$, which is equivalent to $V_i^\alpha \subseteq V_i^\pi$ for all $i \in \mathbb{N}_k$.

Each bank i has a limited size (capacity), which we represent by $S : \mathbb{N}_k \to \mathbb{N}$. Consequently, the aggregate code size of the functions assigned to this bank must not exceed its capacity. Formally, the partition, π, must satisfy $\sum_{v \in V_i^\pi} \sigma(v) \leq S(i)$ for all $i \in \mathbb{N}_k$. Note that by feeding the partitioning algorithm with a smaller number of banks or their lower capacities compared to the actual values, one may influence the final partition, thereby trading off code size for runtime overhead.

To quantify, in turn, the runtime overhead, we need to capture the fact that it is larger for inter-bank calls than for intra-bank calls and calls of functions in the permanently mapped banks. To this end, we introduce an additional cost metric, $C : (V \to \mathbb{N}_k) \to E \to \mathbb{N}$, where $C(\pi)(e)$ denotes the runtime cost of a given function call instruction, $e \in E$, measured, for instance, in cycles of the microcontroller or micro-Jules of energy, depending on the partition, $\pi : V \to \mathbb{N}_k$. The metric can be defined arbitrarily, based on the specification of the microcontroller and the particular bank-switching routines employed by the compiler. For example, we can define C as follows: given a call instruction corresponding to $e = \langle v, u \rangle \in E$ and partition $\pi : V \to \mathbb{N}_k$, $C(\pi)(e) = C_L$ if $\pi(v) = \pi(u)$ (i.e., intra-bank call) or $\pi(u) \in \{0, \ldots, k'\}$ (i.e., permanently mapped bank call), and $C(\pi)(e) = C_H$ otherwise (i.e., inter-bank call), for some C_L and C_H such that $C_L \ll C_H$. Note that in this way $k' \in \mathbb{N}_k$ is implied by C, so it does not have to be an explicit part of the input to the problem.

With these definitions, to minimize the runtime overhead of the program, the partition, π, has to minimize the aggregated runtime cost of all function calls, taking into account that some of these calls are invoked more often than others. Formally, π should minimize the following sum: $\sum_{e=\langle v,u\rangle \in E}[\omega(e) \cdot C(\pi)(e)]$. To explain, for each call instruction in the program, represented by edge $e \in E$, of the function represented by vertex $u \in V$ from within the function represented by vertex $v \in V$, we take the number of invocations of this instruction at runtime during some period, $\omega(e)$, and multiply it by the runtime unit cost of the instruction, which is specified by C, and hence depends also on the function partition, π.

3.3 Complexity Analysis

Theorem 3.1 shows that the so-defined partitioning problem is $\mathcal{NP}$-hard, which implies that, unless $\mathcal{P} = \mathcal{NP}$, only heuristic solutions are practical.

THEOREM 3.1. *The following partitioning problem is $\mathcal{NP}$-hard: GIVEN (1) a directed multi-graph, $G = (V, E)$, (2) a vertex weight function, $\sigma : V \to \mathbb{N}^+$, (3) an edge weight function, $\omega : E \to \mathbb{N}$, (4) a number, $k \in \mathbb{N}^+$, (5) a size limit function, $S : \mathbb{N}_k \to \mathbb{N}$, (6) a cost function, $C : (V \to \mathbb{N}_k) \to E \to \mathbb{N}$, (7) a subset of V, V^α, and (8) an initial partition, $\alpha : V^\alpha \to \mathbb{N}_k$, COMPUTE a partition, $\pi : V \to \mathbb{N}_k$, SUCH THAT (a) for all $v \in V^\alpha$, $\pi(v) = \alpha(v)$, (b) for all $i \in \mathbb{N}_k$, $\sum_{v \in V_i^\pi} \sigma(v) \le S(i)$, where $V_i^\pi = \{v \in V | \pi(v) = i\}$, and (c) $\sum_{e=\langle v,u\rangle \in E}[\omega(e) \cdot C(\pi)(e)]$ is minimal among all valid partitions, that is, ones satisfying (a) and (b), OR ELSE inform that no valid partition exists.*

PROOF. We prove the theorem by reducing a classic $\mathcal{NP}$-complete problem, the Minimum Bisection (MB) problem [13], to our problem. In MB, given an undirected and unweighted graph, $G' = (V', E')$, with an even number of vertices (i.e., $|V'| = 2n$ for some $n \in \mathbb{N}^+$), the objective is to produce a vertex partition, $\pi' : V' \to \mathbb{N}_2$, that divides the vertices into two equal-size subsets (i.e., $\left|\{v \in V' | \pi'(v) = 0\}\right| = \left|\{v \in V' | \pi'(v) = 1\}\right| = n$), so that the number of edges connecting two vertices from different subsets, $\left|\left\{e = \{v, u\} \in E' | \pi'(v) \ne \pi'(u)\right\}\right|$, is minimal.

The reduction requires an algorithm for transforming $G' = (V', E')$, an input to MB, into $G = (V, E)$, σ, ω, k, S, C, V^α, and α, which together can act as an input to our problem. The algorithm can be as follows. As the set of vertices, V, we simply take V'. Similarly, for each undirected edge in E', $e = \{v, u\}$, we create two directed edges in E: $e_1 = \langle v, u \rangle$ and $e_2 = \langle u, v \rangle$. As the vertex and edge weight functions, σ and ω, we take the following ones: $\sigma(v) = 1$ for all $v \in V$ and $\omega(e) = 1$ for all $e \in E$. Since we need to only bisect the graph, $k = 2$. Considering the expected number of vertices in each of the two subsets, n, and the selected vertex weight function, σ, we take a constant size limit function: $S(i) = n$ for all $i \in \mathbb{N}_2$. As the cost function, in turn, we take $C(\pi)(e) = [1 \text{ if } \pi(v) \neq \pi(u)$ where $e = \langle v, u\rangle$ or 0 otherwise$]$. Finally, we take an empty initial partition, so $V^\alpha = \varnothing$ and $\alpha = \varnothing$. All in all, this algorithm for transforming an input to MB into an input to our problem requires time polynomial in G'.

Consequently, we just have to prove that a solution obtained for our problem with such a transformed input can be transformed back into a solution to MB also in polynomial time. To this end, we prove that (i) with such an input to our problem, there always exists a valid partition (i.e., the OR ELSE clause in the theorem is never used) and (ii) a partition being a solution to our problem is also a solution to MB.

To prove (i), that is, that our problem always has at least one valid partition for an input transformed from MB, recall that, according to the theorem, a valid partition, $\pi : V \to \mathbb{N}_2$, must satisfy two conditions: (a) for all $v \in V^\alpha$, $\pi(v) = \alpha(v)$ and (b) for all $i \in \mathbb{N}_2$, $\sum_{v\in V_i^\pi} \sigma(v) \leq S(i)$, where $V_i^\pi = \{v \in V | \pi(v) = i\}$. Condition (a) holds trivially for any partition π because the previous transformation algorithm selects $V^\alpha = \varnothing$ and $\alpha = \varnothing$. Let us thus reformulate condition (b) given the transformation algorithm. First, from the definition of σ in the algorithm, we get: for any $i \in \mathbb{N}_2$, $\sum_{v\in V_i^\pi} \sigma(v) = \sum_{v\in V_i^\pi} 1 = |V_i^\pi|$. Second, in the algorithm: $S(i) = n$ for all $i \in \mathbb{N}_2$. Combining these, we get: $|V_0^\pi| \leq n$ and $|V_1^\pi| \leq n$. At the same time, since $|V| = 2n$, we must have $|V_0^\pi \cup V_1^\pi| = 2n$. Therefore ultimately, for an input transformed from MB, condition (b) is equivalent to $|V_0^\pi| = |V_1^\pi| = n$. In other words, any partition that divides the $2n$ vertices of G into two subsets of n vertices is valid, which proves (i).

Regarding (ii), in turn, let us first observe that we have just also showed that for an input transformed from MB, a partition, $\pi^\star : V \to \mathbb{N}_2$, being a solution to our problem, is also a bisection of G (i.e., $|V_0^{\pi^\star}| = |V_1^{\pi^\star}| = n$). Since $V = V'$, $\pi^\star$ is also a bisection of G'. Consequently, to prove (ii), we just have to show that if $\pi^\star$ minimizes $\Phi(\pi) = \sum_{e=\langle v,u\rangle\in E}[\omega(e) \cdot C(\pi)(e)]$, then it also minimizes $\Psi(\pi) = \left|\left\{e = \{v,u\} \in E' | \pi(v) \neq \pi(u)\right\}\right|$. To start with, from the definition of ω in the transformation algorithm, we get: $\Phi(\pi) = \sum_{e=\langle v,u\rangle\in E} C(\pi)(e)$. From the definition of C in the algorithm, we get in turn: $\Phi(\pi) = \sum_{e=\langle v,u\rangle\in E}[1 \text{ if } \pi(v) \neq \pi(u) \text{ or } 0 \text{ otherwise}] = \sum_{e\in\{\langle v,u\rangle\in E|\pi(v)\neq\pi(u)\}} 1 = \left|\{e = \langle v,u\rangle \in E | \pi(v) \neq \pi(u)\}\right|$. Now, recall that in the transformation algorithm, we create E from E' by adding two directed edges to E for each undirected edge from E', that is, $E = \bigcup_{e=\{v,u\}\in E'} \left\{e_1 = \langle v,u\rangle, e_2 = \langle u,v\rangle\right\}$. This implies that $\Phi(\pi) = \left|\{e = \langle v,u\rangle \in E|\pi(v) \neq \pi(u)\}\right| = 2\left|\left\{e = \{v,u\} \in E'|\pi(v) \neq \pi(u)\right\}\right| = 2\Psi(\pi)$. Consequently, if partition $\pi^\star$ minimizes $\Phi(\pi)$, it also minimizes $2\Psi(\pi)$, and hence $\Psi(\pi)$, which proves (ii). We are thus able to transform a solution to our problem back to a solution to MB in polynomial time with respect to G' (actually, in constant time).

With so-defined polynomial-time reductions, if we had a polynomial-time algorithm for solving our problem, we would be able to solve MB also in polynomial time. However, MB is proven to be $\mathcal{NP}$-complete [13], which implies that it cannot be solved in polynomial time unless $\mathcal{P} = \mathcal{NP}$. This means that, unless $\mathcal{P} = \mathcal{NP}$, no polynomial-time algorithm exists for our problem: the problem is $\mathcal{NP}$-hard. □

4 ALGORITHM

Since the problem of partitioning program functions into banks is $\mathcal{NP}$-hard, our algorithm decomposes it into three subproblems and introduces a heuristic solution for each of them: (1) Which functions to partition? (2) Which of them to assign to the permanently mapped banks? and (3) How to assign the remaining ones to the remaining banks? The three heuristics are run sequentially, as three phases of the algorithm. They make the following assumptions.

4.1 Assumptions

To start with, the source (or intermediate) code of all functions to be partitioned must be available. In particular, all invocations of other functions within each such function are analyzed by the algorithm to compute a partition, and the partition determines the protocol of these invocations (e.g., inter-bank vs. intra-bank), and hence the final binary image of the function. If no source or intermediate code is available for a function, this function, together with the functions on whose invocation protocols it depends, must be pre-assigned manually to appropriate banks. In practice, third-party libraries or drivers that are closed-source may need to be pre-assigned to correct banks.

Furthermore, the functions to be partitioned must be invoked explicitly in the code, that is, they should not be called by anonymous pointers. If, in turn, a function is called by pointer, its invocation protocol must be fixed, and thus the function must be manually either preassigned to a permanently mapped bank or marked as callable between banks. To this end, it should be possible to identify which functions may be called by pointers in the code. This is normally possible in C. What is more, some languages for IoT software, like NesC [14], go even a step further, by strongly discouraging calls by pointers in the programming constructs they offer.

For each function to partition, it must also be possible to determine the size of the function's binary image (i.e., σ in the model) as well the expected runtime frequency of the function's invocations (i.e., ω in the model). For preassigned functions, the image size should be given; for the ones to partition, it can be computed by recompiling the functions. We discuss how to implement this efficiently in Section 5. The runtime frequencies, in turn, can be obtained based on actual profiling results or some heuristic algorithms. Again, we return to these implementation details in Section 5.

Regarding the microcontroller model, the number of banks, including permanently mapped ones (i.e., k and k' in the model), as well as their size limits (i.e., S in the model) must be obtained from the microcontroller specification. Using the same reasoning, as the call cost function (i.e., C in the model), we need not consider an arbitrary one but rather a function that reflects the implementation of bank switching in microcontrollers and compilers. More specifically, we assume a function that defines an invocation of function `foo` as inter-bank if and only if `foo` is assigned to a switchable bank and there exists another function that is assigned to a different bank and that invokes `foo`. This is exactly what today's compilers implement: if a function is marked as callable between banks, its invocation always follows the inter-bank protocol, irrespective of the bank of the caller. For simplicity, we assume that the cost of such a call is one, whereas the cost of other calls is zero. In other words, our algorithm aims to minimize the runtime number of calls done with the inter-bank protocol.

To sum up, based on the source (or intermediate) code of a program, a function call graph is constructed. Since this construction can be done trivially in linear time, we omit its description here. Given the call graph and under the previous assumptions, our algorithm produces a partition of the functions into banks or raises an exception if it is unable to do so. The whole process has a low, polynomial-time complexity, which is crucial from the perspective of interactive software development. Any partition produced by the algorithm is valid in the sense of Theorem 3.1 but it need not guarantee the minimal runtime cost of function invocations, even though the heuristics have been designed specifically to keep this cost low given the previous assumptions.

4.2 Selecting Functions to Partition

Although the goal of the algorithm is to partition *all* functions that are not pre-assigned, in practice there is also another option: replicating some functions in multiple banks. For example, if function w is invoked by functions v and u, which are assigned to different banks, that is, $\pi(v) = i \neq j = \pi(u)$, then we can replicate w as w_i and w_j in both the banks and modify v and u such that they invoke respectively w_i and w_j. In effect, both these invocations become intra-bank. However, the cost is potentially more bank space occupied. Furthermore, replicating just w may be insufficient to significantly improve the runtime performance, as w may itself invoke other functions, which invoke other functions, and so on, such that many of these functions may need to be replicated to notice a performance difference. Finally, allowing for replication would lead to a significant increase in the complexity of the problem because one would have to decide not only on the function placement but also on the number of their replicas.

For these reasons, our algorithm does use function replication but in a way ensuring that its complexity is acceptable. More specifically, in the first-phase heuristic of the algorithm, the function call graph, $G = (V, E)$, is modified by forcing selected functions to be inlined, that is, to have their bodies inserted instead of their invocations in the code of other functions. To this end, for each function, $v \in V$, it is checked whether inlining this function would be beneficial, and if so, v is inlined in the place of each invocation $e = \langle u, v \rangle \in E$ from within any other function, $u \in V$. To avoid checking the same function multiple times, the heuristic processes the functions in a particular order, so that when function v is considered for inlining, all functions it invokes directly or indirectly have already been processed. The order is obtained by topologically sorting G and reversing the resulting order. Edges leading to cycles, which correspond to calls between (mutually) recursive functions, are simply ignored during the sorting, that is, when sorting we consider a maximum $E^{\preceq} \subseteq E$ such that $(V, E^{\preceq})$ does not contain any cycles. As long as self-recursive calls are not inlined, ignoring cycle-edges does not influence the correctness of inlining; the number of inlined functions need not be optimal, though. Nevertheless, this is not a problem because the entire algorithm is heuristic anyway and, what is more, (mutually) recursive functions are virtually nonexistent in IoT software. Algorithm 1 contains the pseudocode of the heuristic.

```
1  $(E^{\preceq}, V^{\preceq}) \leftarrow ($
         a maximum subset of $E$ such that $(V, E^{\preceq})$ is a directed acyclic multigraph,
         $V$ sorted in a reverse topological order with respect to the edges in $E^{\preceq}$
     $);$
2  **foreach** $v \in V^{\preceq}$ **do**
3      **if** *beneficial to inline* $v$ **then**
4          **foreach** $u \in V$ **such that** $e = \langle u, v \rangle \in E$ **and** $u \neq v$ **do**
5              insert the body of $v$ in the place of call $e$ within $u$;
6              $E \leftarrow E \setminus \{\langle u, v \rangle\}$;
7              $E \leftarrow E \cup \{\langle u, w \rangle | \langle v, w \rangle \in E\}$;
8          **end**
9          **if** $\langle v, v \rangle \notin E$ **then**
10             remove $v$ from the program;
11             $V \leftarrow V \setminus \{v\}$;
12             $E \leftarrow E \setminus \{\langle v, w \rangle \in E\}$;
13         **end**
14     **end**
15 **end**
```

Algorithm 1: Global function inlining.

An open design option is the check whether inlining a given function is beneficial (line 3 of the pseudocode). In particular, in our implementation, a function is inlined if the binary size of its body is smaller than a threshold given as a compiler option. An alternative implementation could in turn consider the total execution time of the function's invocation, depending on whether the function is inlined. Although computing this time may be infeasible in a general case, in IoT software, especially pieces with real-time guarantees, many functions have easily computable execution times. There are also other inlining criteria that may be considered, such as the function's invocation frequency, the number of other functions from which the function is invoked, or the number of the function's parameters, to name some examples. In any case, the check must also ensure that repeated inlining does not yield a function whose size exceeds a threshold, as this could complicate or even preclude partitioning.

Overall, the heuristic reduces the number of functions to partition by eliminating many small ones. Importantly, this operation has a low complexity. The topological sorting (line 1 of Algorithm 1) is done in a time bounded by $O(n + m)$, where $n = |V|$ and $m = |E|$ [9]. There are $O(n)$ checks of whether inlining a function is beneficial (line 3) and at most $O(m)$ actual inlining actions (line 5), yielding $O(n + m)$ as the total time complexity also for the main loop (lines 2–15). Although updating the call graph as a result of inlining (especially line 7) may suggest a higher complexity, note that an inlined function is small, which implies that the number of invocations it contains, and hence number of the edges in E that need to be modified upon its inlining, is limited by a constant. All in all, the total complexity of the heuristic is $O(n + m)$. In other words, the inlining phase of our algorithm can be fast.

Finally, although inlining is normally supported by embedded compilers, it is typically performed for individual compilation units (e.g., C files). In contrast, global inlining, as in our heuristic, is still rather uncommon. What is more, by introducing an explicit inlining phase, we can better control the inlining criteria. The subsequent phases of our algorithm thus operate on the call graph modified by Algorithm 1.

4.3 Assigning Functions to Permanently Mapped Banks

The second phase of the algorithm targets permanently mapped banks. These banks are treated specially because functions assigned to them are always addressable via the program counter, and hence their invocations do not require any bank switching, as such being inexpensive. Taking this into account, to minimize the aggregate runtime cost of all function invocations, our phase-two heuristic should assign to the permanently mapped banks those functions that are invoked most frequently by all other functions, that is, those functions $v \in V$ for which $F(v) = \sum_{e=\langle u,v\rangle \in E} \omega(e)$ is larger than for others. Since such functions may also have different sizes, $\sigma(v)$, it is more beneficial to select smaller functions first, in other words, to assign to the permanently mapped banks those functions $v \in V$ that maximize the following utility metric $U(v) = \frac{F(v)}{\sigma(v)}$.

For a single bank, i, such a general selection problem is known as the 0-1 Knapsack problem, which is also proved to be $\mathcal{NP}$-hard [13]. In our case, the weight values for the knapsack items, $\sigma(v)$, are natural numbers, so a simple algorithm that uses dynamic programming could be employed to produce an exact solution. However, the time complexity of this algorithm is $O\big(n \cdot S(i)\big)$ and the memory complexity is $O\big(S(i)\big)$, where $n = |V|$, that is, the algorithm is exponential in its input: the number of bits describing the bank size, $S(i)$. With k' banks, this complexity would be even higher. What is more, even an optimal assignment of the permanently mapped banks need not yield an optimal global assignment, a simple proof of which we leave for an interested reader. For these reasons, we resort to a much faster and memory efficient greedy heuristic, with pseudocode presented in Algorithm 2.

The heuristic first computes the utility value for each function. It then sorts the functions such that the functions with higher utility values will be considered first for assignment. Subsequently, it iterates over the functions in this order, considering each one for assignment to some bank. More

```
1  foreach $v \in V$ do
2      $U(v) \leftarrow \frac{1}{\sigma(v)} \cdot \sum_{e=\langle u,v\rangle \in E} \omega(e)$
3  end
4  $S \leftarrow V$ sorted in the descending order of values of $U$;
5  foreach $v \in S$ do
6      $i \leftarrow$ a permanently mapped bank with the least amount of free space that can hold additional
         $\sigma(v)$ bytes or $null$ if such a bank does not exist;
7      if $i \neq null$ then
8          $\pi(v) \leftarrow i$;
9          $V \leftarrow V \setminus \{v\}$;
10         $E \leftarrow E \setminus \{\langle u, w\rangle \in E | u = v \text{ or } w = v\}$;
11     end
12 end
```

Algorithm 2: Assigning functions to permanently mapped banks.

specifically, the function is assigned to a bank that has the least amount of free space but still enough to hold the function.

The complexity of computing the utility values (lines 1–3 of the pseudocode) is $O(n + m)$ as each vertex and edge of G must be visited exactly once. The vertex sorting (line 4) can be completed in time bounded by $O(n \cdot \log n)$. Finally, the assignment itself (lines 5–8) visits each vertex exactly once, searching for the most occupied permanently mapped bank with enough free space to hold the vertex. Therefore, if the free space in the banks is stored in a balanced binary tree, the complexity of the entire assignment is $O(n \cdot \log k')$. Removing the edges for assigned vertices (line 10) will in turn not exceed $O(m)$ in total for all vertices. All in all, the time complexity of the phase-two heuristic is $O(n \cdot \log k' + n \cdot \log n + m)$. In other words, like the phase-one heuristic, this heuristic is also fast. What is more, if there is only one permanently mapped bank ($k' = 1$), as in the aforementioned CC253x, we can modify the algorithm slightly to guarantee that the aggregate utility of the resulting partition is no less than half of the utility of an optimal partition [44]. Finally, note that n and m in this section describe the size of the call graph after the global inlining, which typically should be smaller than the original call graph. However, it may happen that no functions are inlined by the first-phase heuristic. Nevertheless, the big O notation is correct even if n and m describe the size of the original call graph.

4.4 Assigning Functions to Remaining Banks

In the third and final phase of the algorithm, all functions that have not been partitioned yet are assigned to switchable banks. The main idea is to allocate entire sets of "related" functions to the same bank: all functions from such a set are assigned simultaneously to a bank that has enough free space to hold them. To this end, however, two issues must be addressed first: (1) How to determine the sets of "related" functions? (2) How to efficiently compute the aggregate bank space occupied by such a set?

Regarding the first issue, intuitively, "related" functions are those that call each other frequently, so that if they are allocated to the same bank, the number of inter-bank calls is small. To simplify identifying such functions, we disregard the direction of edges in G, thereby making use of the fact that the runtime cost of inter-bank invocations is symmetric. Under this assumption, we consider functions "related" if they belong to the same K-connected components of G.[1] A K-connected

[1]In this notation, K is independent of the number of banks, k.

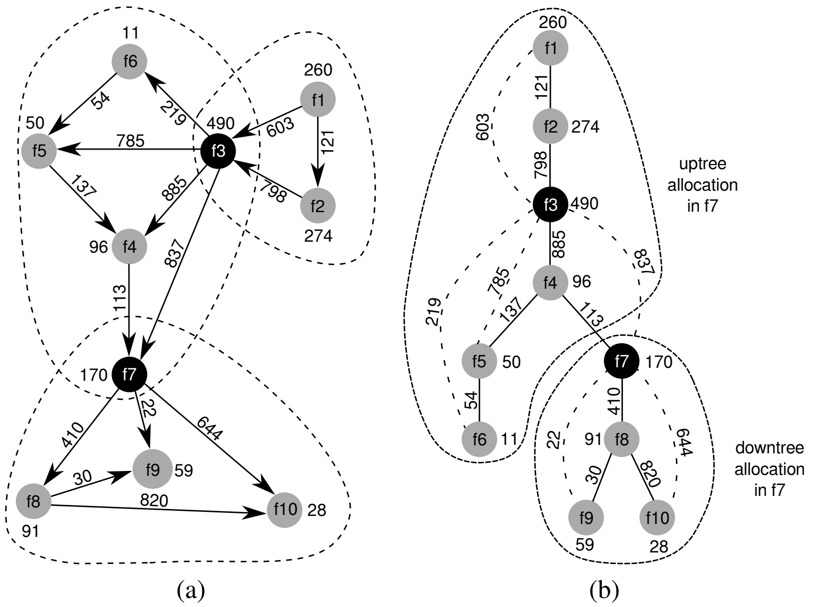

Fig. 2. Biconnected components of a sample graph (a) and selected allocations (b).

component of an undirected graph is a maximum subgraph such that to partition the subgraph, one has to remove at least K of its edges. In other words, functions belonging to the same K-connected component contain many call instructions for each other and fewer calls for other functions, and hence are likely to yield few inter-bank calls.

Computing K connected components for an arbitrary value of K can be done in polynomial time with Minimum Cut or Maximum Flow algorithms [21, 38] but the polynomial degree is high. There exists, however, a linear algorithm for $K = 2$, that is, for finding biconnected components in an undirected graph [16]. Consequently, although our heuristic can be generalized to an arbitrary K, for efficiency it employs biconnected components.

A vertex (function) can belong to one or more biconnected components of a graph. Vertices that belong to more than one biconnected component are called *cut vertices* because removing such a vertex partitions the graph. Put differently, only functions corresponding to cut vertices can be called by functions from more than one biconnected component. What is more, if we look at a graph, $G^{\ddagger}$, created from G by adding one vertex in $G^{\ddagger}$ for each biconnected component in G and one vertex for each cut vertex in G and connecting these vertices in $G^{\ddagger}$ as implied by their connectivity in G, then $G^{\ddagger}$ is a tree, which has been an inspiration for the design of our heuristic. Figure 2(a) illustrates these concepts on a sample graph. There are three biconnected components in the graph, marked with dashed lines. There are also two cut vertices, $f3$ and $f7$, each of which belongs to exactly two different biconnected components.

Cut vertices are important because due to their properties they are used by our heuristic as candidate points for partitioning sets of functions. More specifically, after identifying biconnected components and cut vertices in G, the heuristic builds a spanning tree over G with some vertex acting as the root. To be precise, it builds a spanning forest containing potentially more than one tree because G need not be connected. It then considers two allocations for each cut vertex v in every tree of the forest. A *down-tree allocation* at v is the set of functions represented by all vertices from the subtree rooted at v. An *up-tree allocation* at v is in turn the set of functions represented by all vertices in the tree that do not belong to the subtree rooted at v. Vertex v itself belongs to that of the two sets whose aggregated weight of edges with v is larger. Figure 2(b) presents a sample spanning tree, consisting of the solid edges, for the function call graph from Fig. 2(a); the dashed edges, in turn, correspond to those edges in the original graph that do not belong to the tree. Both allocations at vertex $f7$ are also marked. The up-tree allocation consists of vertices $f1$, $f2$, $f3$, $f4$, $f5$, and $f6$ while the down-tree allocation of vertices $f7$, $f8$, $f9$, and $f10$. The cut vertex $f7$ belongs to the down-tree allocation because the sum of weights with this allocation, equal to $22 + 410 + 644 = 1076$, is greater than the sum for the up-tree allocation, equal to $113 + 837 = 950$.

The heuristic assigns to banks entire allocations for selected cut vertices. To this end, it must be able to quickly determine whether an allocation fits in a bank, which brings us to the second of the aforementioned issues: efficiently computing the total size of the functions in an allocation. To provide such functionality, we use the *consecutive property* of pre-order tree numbering: if we visited the vertices of the spanning tree in which we search for allocations in pre-order, giving a subsequent number to each visited vertex, then all d tree descendants of a vertex with number j, including the vertex itself, would have numbers $j, j + 1, \ldots, j + d - 1$. Because of this property, we can reduce the problem of computing the total size of functions in a down-tree or up-tree allocation to computing the sum of values associated with numbers from a given range. More specifically, we associate the pre-order number of vertex v with the value of $\sigma(v)$. In particular, in Fig. 2(b), the pre-order number of a vertex is the number in the name of the corresponding function. Given these numbers we can efficiently compute the allocation sizes for v. For instance, the size of the selected down-tree allocation in Fig. 2(b) can be computed as the sum of values for range: 7, 8, 9, 10 whereas the size of the up-tree allocation, as a difference between the sum of values for the full range (the entire tree) and the size of the down-tree allocation.

In order to efficiently compute a sum of values associated with ranges of consecutive numbers, we use a data structure known as a *range tree* [29]. It is a (nearly) complete binary tree whose leaves correspond to individual numbers (ranges of length 1) with appropriate values whereas each internal node corresponds to a range being the union of the ranges of its children with the associated value equal to the sum of the values of the children. Such a structure allows us to query and update allocation sizes in time bounded by $O(\log n)$, which is crucial to the performance of the algorithm.

Having resolved the two issues, the phase-three heuristic becomes a relatively simple iterative algorithm. At any time, it maintains a spanning forest for the call subgraph comprising unassigned functions and a range tree for computing allocation sizes for these functions. In each iteration, it considers down-tree and up-tree allocations for every cut vertex, chooses the best one, and assigns all functions from the allocation to a bank. If no such allocation fits into any bank, then it performs an alternative partitioning procedure. In any case, immediately after assigning a function to a bank, the function is removed from the graph and the spanning forest, and zero is associated with its pre-order number in the range tree. This is necessary to correctly compute the sizes of future allocations. Pseudocode of the entire heuristic is shown in Algorithm 3.

The algorithm leaves several design decisions open. The first one is related to the spanning tree (forest) over the function call graph (line 2): it can be a minimum or maximum tree with respect to the edge weights represented by ω, or a tree obtained by a bread-first or depth-first search, as in the algorithm for computing biconnected components (line 1), to name just a few examples. The second open decision is the method of ranking candidate allocations (line 10). For instance, allocations with greater total function sizes or containing more frequently called functions can be preferred. The third option is the alternative partitioning procedure (line 13). Examples of such procedures include, among others, allocation of individual functions in the order of visiting them during a depth- or

```
1  $(B, C) \leftarrow$ (the biconnected components of $G$, the cut vertices of $G$);
2  $F \leftarrow$ a spanning forest for $G$;
3  $R \leftarrow$ a range tree with pre-order visit numbers of $F$ mapped to function set sizes;
4  while $F \neq \varnothing$ do
5      $best \leftarrow null$;
6      foreach $v \in C$ do
7          $(cand_D, cand_U) \leftarrow$ (down-tree allocation at $v$, up-tree allocation at $v$);
8          $(size_D, size_U) \leftarrow$ (
                   the total size of $cand_D$ computed with $R$,
                   the total size of $cand_U$ computed with $R$
            ) ;
9          $(bank_D, bank_U) \leftarrow$ (
                   a bank with at least $size_D$ free space or $null$ if there is none,
                   a bank with at least $size_U$ free space or $null$ if there is none
            );
10         $best \leftarrow$ the best of $\{best, (cand_D, size_D, bank_D), (cand_U, size_U, bank_U)\}$;
11     end
12     if $best = null$ then
13         $best \leftarrow$ select an alternative partition;
14     end
15     if $best = null$ then
16         terminate the algorithm with an “unable-to-partition” error;
17     end
18     perform the assignment represented by $best$;
19     remove $best$ from $G, B, C, F$ and $R$;
20 end
```

Algorithm 3: Assigning remaining functions to switchable banks.

breadth-first search or considering down-tree and up-tree allocations at all vertices instead of only at cut vertices. Finally, the choice of the bank for an allocation (line 9) can also be customized, for instance, if the allocation fits in more than one bank, then a bank with the minimum or maximum amount of free space can be selected, or alternatively the last assigned bank can be preferred.

Due to space constraints, however, in the remainder of this article, notably for the experimental evaluation, we assume the following variants for these options. First, as the spanning trees, we utilize depth-first search trees constructed by the algorithm for computing biconnected components. Second, we prefer allocations with a greater aggregate size of functions. Third, as the alternative partitioning method, we select assigning vertices in the order of visiting them during a depth-first search. Finally, as preferred candidate banks, we choose those with the smallest amount of free space.

Again, an advantage of the heuristic is its computational complexity. As mentioned previously, identifying biconnected components and cut vertices (line 1) can be done in time $O(n + m)$ [16]. Constructing a spanning forest (line 2) requires $O(m \cdot \log n)$ time with Kruskal's algorithm for minimal/maximal spanning trees or $O(n + m)$ time for breadth-/depth-first search trees [9]. The initialization of the range tree (line 3) takes $O(n \cdot \log n)$ time [29]. In the pessimistic case, the main while loop (line 4) assigns a single vertex per iteration, thereby requiring $O(n)$ iterations. The inner loop (line 6) iterates over all cut vertices, the number of which may approach $O(n)$ in some graphs. The dominating operations in this loop are the computation of total function sizes (line 8), whose complexity is $O(\log n)$ thanks to the range trees, and bank selection (line 9), which can be done in time bounded by $O(\log k)$ with balanced search trees. The alternative partitioning methods proposed by us (line 13) have their complexity bounded by $O\big(n \cdot (\log k + \log n)\big)$. Finally, the remaining operations in the main loop (lines 15–19) in total in all iterations of the loop can be realized in time $O\big(m + n \cdot (\log k + \log n)\big)$. All in all, the total time complexity of the phase-three heuristic is bounded by $O\big(n^2 \cdot \log(k \cdot n) + m \cdot \log m\big)$,[2] where n and m describe the graph size after the two previous phases. As we show in Section 6, this relatively low complexity makes this phase-three heuristic fast in practice.

5 IMPLEMENTATION

Let us present how to realize the proposed solutions, simultaneously introducing our implementation, which not only is publicly available[3] but has also been deployed in a production toolchain of at least one IoT company, as we elaborate in Section 6. First, we give a general view of a compilation process employing our algorithm. Second, we describe methods of estimating function sizes (σ in the earlier model) and runtime invocation frequencies (ω in the model). Finally, we discuss the implementations of existing algorithms, which we developed for comparison purposes with our algorithm.

5.1 Overview

A partitioning algorithm can be implemented in two forms: as an external tool or directly in a compiler generating target machine code. Due to various constraints, our implementation resembles the external tool approach. Nevertheless, since a direct implementation in a compiler can be more efficient, we discuss both the approaches here.

When integrated in a compiler, a partitioning algorithm can be run between generating intermediate code and target machine code, provided that this is done for the entire program. Alternatively, if the entire program is not compiled at once, the heuristic can be run just before linking object files into a complete binary program image. However, in the latter case, individual functions in the image, notably call instructions, will have to be patched to reflect the final invocation protocols, which depend on the partition. In either case, the result of the compilation process is a binary image for each bank. An example of a compiler for IoT microcontrollers that could be extended to support automatic function partitioning is the aforementioned SDCC [12].

Our implementation follows in turn the external-tool approach. More specifically, it has been made part of the 8051 backend in wNesC, a compiler for the NesC language for IoT software [14]. In NesC, all files comprising a program are translated to a single C file, which is then compiled with a C compiler, like the aforementioned SDCC. Our heuristic is run in between. First, it estimates sizes of all functions in the C program and simultaneously selects functions to inline by running SDCC. The inlining policy can be selected with a compiler option; by default, functions of size less than or equal to 20 bytes are inlined. However, for the evaluation described in the next section we changed this limit: we decided to inline functions not larger than 30 bytes. We selected this value as a result of a dedicated test performed to determine a boundary that minimizes the size of the generated binary code. After that, a partitioning heuristic, again selected with a compiler option, is run to assign the remaining C functions to banks, which results in a separate C file per bank. Finally, these C files for subsequent banks are compiled with SDCC. The process is presented in Fig. 3. The details, in turn, can be found in the aforementioned online repository of wNesC.

[2]Making different decisions at the open points may affect this complexity.
[3]https://github.com/mimuw-distributed-systems-group/wNesC-Compiler

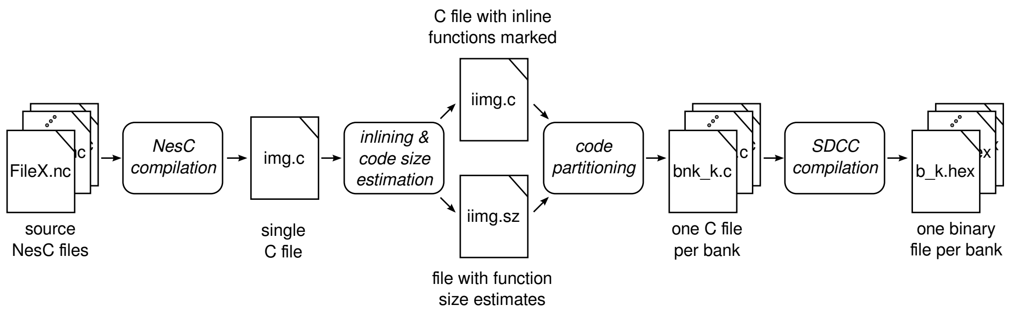

Fig. 3. The compilation process with partitioning heuristics in our implementation.

5.2 Estimating Function Sizes

When deploying our algorithm, an important aspect is the ability to estimate function sizes in the target binary image, so that no bank capacity is exceeded in a partition. If our solutions are integrated into a compiler, the compiler can generate machine code at any time. Moreover, such compilation can be done for individual functions, as function is the unit of compilation in C. Therefore, whenever the size of a function may have changed, in particular, as a result of a change to the invocation protocol of other functions, the function can be recompiled to obtain its new size estimate.

In contrast, in our implementation, the situation is more complex. Since wNesC does not generate machine code, relying in this respect on SDCC, our implementation must run SDCC to obtain function size estimates. For efficiency, this is performed by multiple threads whose cardinality can be controlled by the user. Likewise, the number of functions whose sizes are to be estimated in a single batch (a separate C file) is selected in an adaptive manner to minimize the number of SDCC executions. The estimation phase is naturally combined with inlining, as explained in Section 4.2, but the algorithm differs slightly for efficiency reasons. For the same reasons, functions that are not inlined are compiled only once to obtain the upper bounds on their machine code size, that is, their size as if all calls they make were inter-bank. All in all, due to keeping the partitioning software separate from the compiler that generates machine code, function size estimation in our implementation is quite elaborate. For more details, we refer the interested reader to the aforementioned repository of wNesC.

5.3 Estimating Function Runtime Frequencies

Another implementation-dependent aspect is obtaining estimates for runtime function invocation frequencies. This can be done by profiling generated machine code of the program, for instance, by associating a counter with each function invocation and incrementing this counter whenever the invocation is executed. After such profiling, the counters can be downloaded to a file and utilized as the frequency estimates during the next compilation of the program. Such an approach is feasible even on the resource-constrained IoT microcontrollers [20]. Moreover, it should give reasonable results because IoT software is largely event-driven, and hence, by profiling it for a sufficiently long time, we should be able to capture the expected function invocation frequencies. As an alternative, the programmers can be offered a set of compiler pragmas corresponding to runtime frequency classes. With these pragmas, they would be able to mark pieces of code that are invoked more often than others.

To further simplify the process, we have implemented a heuristic that produces estimates—even without any external input—by analyzing the source code. The heuristic can thus be utilized as a first approximation of function invocation frequencies or can be combined with the two previous methods, in particular, if the estimates obtained by these methods are sparse. Essentially, it analyzes the control flow graph of the program and gives higher frequencies to function invocations within loops and lower ones to functions in conditional statements. This analysis is performed globally and starts from root functions (e.g., the main function or interrupt handlers) or other functions with well defined frequencies. For each loop a function call is nested in, directly or indirectly, the frequency of this call is multiplied by $l \geq 1$. Therefore, the final frequency of the call is proportional to l^n, where n is the number of the loops (the global nesting level) of the call. Likewise, for each conditional instruction a call is nested in, the frequency of this call is multiplied by $\frac{1}{c} \leq 1$. If the call is reachable via multiple control flow paths, the frequencies computed for these paths are combined. The details can be found in the source code. Due to space constraints here, for the evaluation experiments in the next section, we assume just a single configuration: $l = 8, c = 2$. Other configurations can be selected with a compiler option. What is also important is that the compiler does not rely on any specific estimation mechanisms, which makes it possible to replace them at will, for instance, in effect of future research developments.

5.4 Implementations of Other Heuristics

Apart from ours, wNesC features the three other partitioning algorithms mentioned in Section 2: bin packing (denoted *BinPack*), the algorithm by Li et al. [26] (denoted *MaxGains*), and the tabu search algorithm by Mengting et al. [30] (denoted *TMSearch*). In fact, those algorithms were implemented in wNesC before we devised our algorithm because automated code partitioning had been requested by wNesC adopters as an important functionality of the compiler. The algorithms differ in their optimization goals, though. Whereas our algorithm maximizes the runtime performance, the bin packing algorithm tries to maximize bank space utilization while the other two aim at minimizing the size of the target binary image resulting from partitioning. Moreover, MaxGains and TMSearch required adaptation from their original lazy bank switching model to the eager model supported by wNesC. Therefore, let us describe the implementations of the algorithms in more detail.

The original bin packing heuristic implemented in the Contiki tool [37] partitions entire C compilation units (object files) instead of individual functions. In contrast, to improve bank space utilization, in BinPack a finer granularity is possible: individual functions are assigned to banks. They are ordered from the largest to the smallest and partitioned in this order. To maximize memory utilization, a function is assigned to a bank in which it fits and which has the smallest amount of free space.

The other two algorithms, MaxGains and TMSearch, were originally designed for the lazy bank switching model. In contrast, wNesC targets architectures implementing the eager model. Since these two models inherently differ, the two algorithms were adapted to the eager model to produce best possible results. In particular, the requirements for calling a function without bank switching overheads in the SDCC realization of code banking were taken into consideration and used in the adaptation instead of the corresponding assumptions in the lazy model. Moreover, in the case of MaxGains, changes were also introduced to assign frequently called functions to permanently mapped banks. For more details, refer to the aforementioned wNesC sources.

All in all, the implementations of the algorithms aim to make empirical comparisons as fair as possible. To also illustrate their development complexity, Table 1 presents the lines of code corresponding to each of the algorithms in wNesC. It is worth noting that the number of lines for our heuristic is so large because we implemented many variants for the open design decisions, which can be chosen with compiler options.

Table 1. Lines of Code for Each Algorithm in wNesC

Algorithm	Lines of code
BinPack	121
MaxGains	439
TMSearch	1680
our algorithm	3118

6 EVALUATION

We empirically evaluated the implemented partitioning algorithms on actual IoT wireless sensor devices employing the aforementioned CC2531 SoCs from Texas Instruments, Inc. [40]. In this section, we present the test software we utilized for the experiments, followed by the experimental results for the algorithms.

6.1 Test Applications

We tested the algorithms on two types of programs: synthetic applications and actual commercial IoT applications. All the applications were written in NesC and were compiled with the freshest versions of wNesC and SDCC.

The purpose of the synthetically-generated applications is to illustrate how the algorithms perform on function call graphs with specific structures that may be problematic to partition. To begin with, *Star* is an application in which functions are organized in a star: there is one function that calls all other functions. In *Paths*, the functions are organized into 25 paths, each path consisting of 24 vertices, such that each internal function on a path invokes the next function. In *Tree*, the function call structure is an almost complete binary tree with 685 vertices. In *Cycles*, the call graph contains 139 cycles of lengths 1, 2, 3, and 4 corresponding to (mutually) recursive functions. In *Random*, the call graph is a random connected directed acyclic graph with 105 vertices and at least 5% of all possible directed edges. Finally, in *Clique*, the functions are organized into a clique of 38 vertices. For more details, we refer the interested reader to the source code of these applications.[4]

The second application suite aims to demonstrate how the partitioning algorithms perform on actual IoT software. It comprises proprietary applications of an IoT company that has deployed the presented solutions in their production toolchain.[5] More specifically, *Blink* is a small test application that toggles with different frequencies three LEDs of a device, as such containing only basic operating system facilities without the network stack. *NetStackDemo* is a demo application of the company's IPv6 stack in which every wireless device generates IPv6 packets and processes packets received from other devices over the radio. In *UDPEcho*, each node acts as an echo client and server that can exchange UDP datagrams with other nodes or hosts in the Internet. Finally, in *MonitoringApp*, each device periodically reads connected sensors and sends the readings in UDP packets to the Internet, possibly with some storage and cryptography in between. It is the fundamental application used by the company in a number of commercial deployments. Although the source code of the applications is proprietary, and hence cannot be published, Table 2 gives an overview of the statistical properties of the sources that are important from the partitioning perspective. Likewise, Fig. 4 gives a more detailed picture of the most involved of the applications.

[4]http://students.mimuw.edu.pl/%7Emc305195/code-banking/synthetic-apps.zip
[5]http://invinets.com

Table 2. Properties of the Real-world Test Applications

	Blink	NetStackDemo	UDPEcho	MonitoringApp
Number of functions	481	1397	1582	1988
Number of function calls	529	1998	2411	2899
Avg. in/out calls per function	1.100	1.430	1.524	1.458
Std. dev. in calls per function	1.213	2.726	2.414	2.528
Std. dev. out calls per function	1.749	3.308	2.839	2.534
Avg. function size [B]	17.37	49.25	58.44	48.60
Std. dev. function size [B]	46.40	117.61	123.33	106.07
Avg. function frequency estimate	2.48	4.02	4.05	4.17
Std. dev. function frequency estimate	4.26	10.71	12.12	13.75

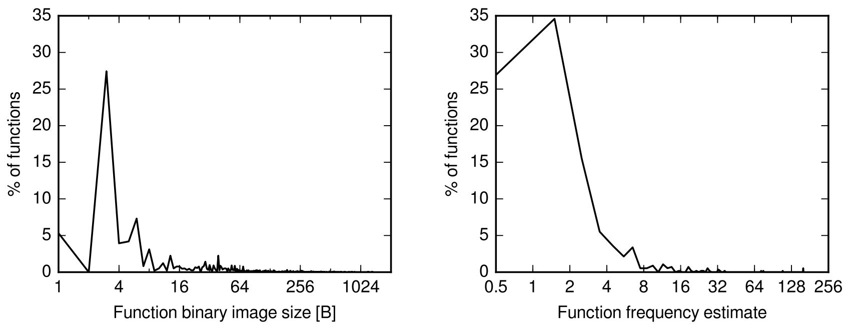

Fig. 4. A histogram of function sizes (left) and frequency estimates (right) for *MonitoringApp*.

6.2 Results

The results of the experiments can be found in Fig. 5. Apart from the implementation of our algorithm, we tested the aforementioned implementations of the three existing algorithms: BinPack, MaxGains, and TMSearch. As an additional reference, in the case of the real-world applications, we also show results for manual partitioning done by developers with expert knowledge of the applications when the automated partitioning algorithms were yet not implemented in wNesC. The quality of all partitions is measured with several metrics, of which the most relevant three are depicted in Fig. 5.

The first one is *average count of banked calls*, where a banked call is a function call following the inter-bank invocation protocol. This metric is thus a proxy of the degradation of the runtime performance caused by bank switching overheads: the smaller its value, the better the performance. It was measured at runtime by counting all banked calls during nineteen 2-second periods directly after turning on a sensor device and computing an arithmetic mean of these counts. We selected this interval specifically to illustrate the calls both during program initialization and during a sufficient period of normal operation. Choosing a shorter interval instead would favor the calls during initialization whereas a longer one would favor the periodic calls.

Our algorithm outperforms the other ones for both synthetic Fig. 5(a) and real-world Fig. 5(b) applications, except for the synthetic *Cycles* application, for which TMSearch performs the best. Furthermore, among all algorithms, its results come the closest to those for the manual partition by experts. It is also interesting to note that for some test applications, some algorithms fail to provide partitions that guarantee correct runtime operation. More specifically, partitions generated by BinPack and TMSearch for *NetStackDemo* caused the program to crash and made it impossible

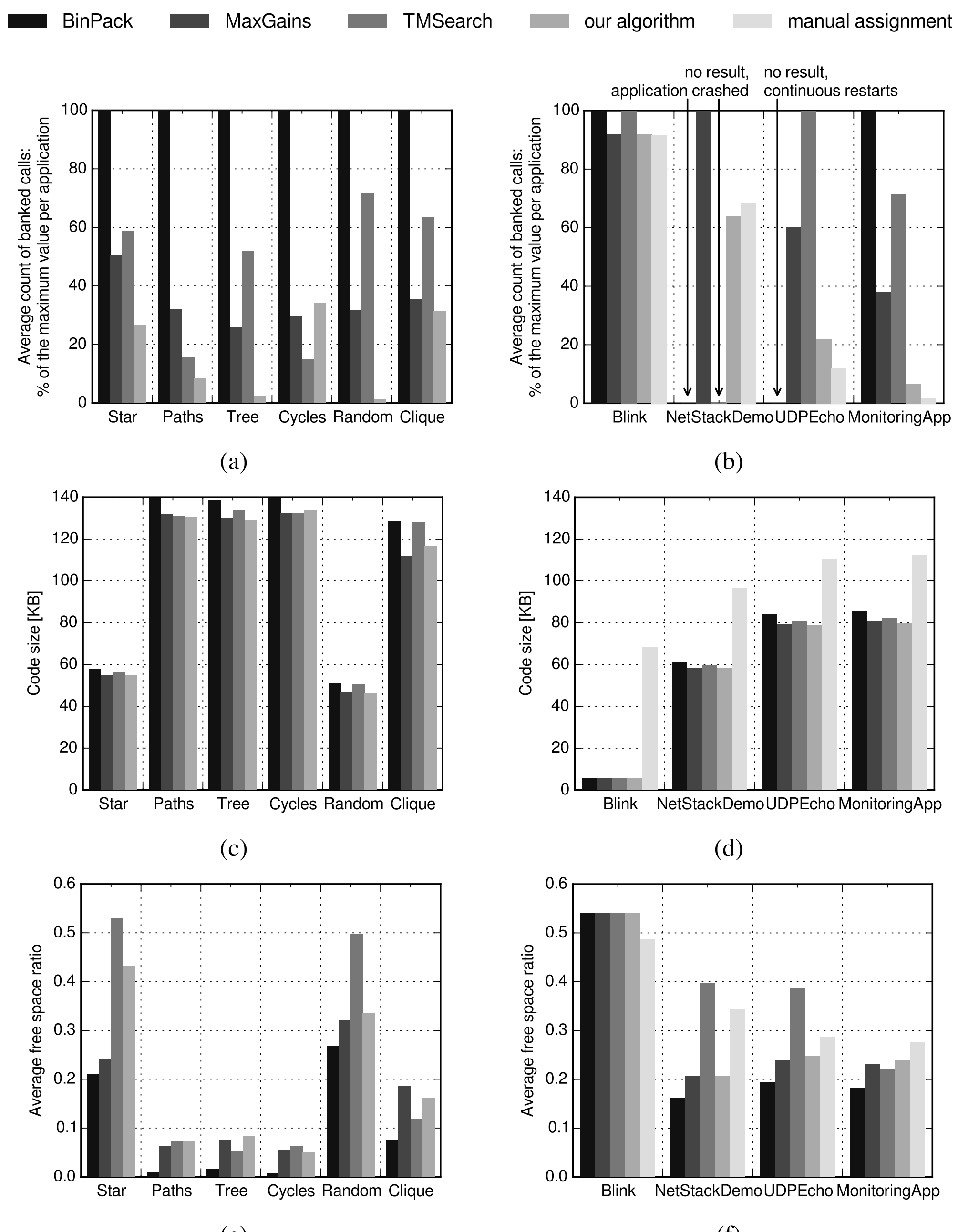

Fig. 5. The results of the tests: average counts of banked calls, normalized per application, for synthetic (a) and real-world (b) applications, code size for synthetic (c) and real-world applications (d) and, finally, average free space ratio for synthetic (e) and real-world (f) applications.

to compute the average count of banked calls. This could happen because of a stack overflow that could occur for these partitions but we were unable to verify the exact cause. Similarly, after programming the test microcontroller with the binary image of *UDPEcho* generated by BinPack, the microcontroller was repeatedly restarting, so again we were unable to properly calculate the number banked calls. All in all, however, there is enough evidence that our algorithm generates partitions with fewer banked calls.

The second considered metric is *code size*. It is the total size in bytes for all banks of the final binary image of an application partitioned by an algorithm. It thus incorporates the code overhead of all bank-switching instructions that are necessary for inter-bank function invocations. The smaller the code size is for a partition of a program, the better, as this potentially allows for adding extra functionality to the program in the future without exceeding the total available bank space.

The results for this metric are shown in Fig. 5(c) for the synthetic and Fig. 5(d) for the real-world applications. It can be observed that the code size does not differ much for the algorithms, except for manual assignment. This is because with manual assignment, unused functions are not removed, as in the inlining phase for automated partitioning. Overall, our algorithm is again better than others for the greatest number of applications; it is worse, albeit just slightly, only for *Cycles*, *Clique*, and *NetStackDemo*.

The last presented metric is *average free space ratio*. It is the arithmetic mean of unused ratios for banks to which at least one function is assigned, where an unused ratio for a bank is the ratio of the amount of the free space remaining in the bank after partitioning to the capacity of the bank. This metric thus measures the utilization of the space in banks that do store something: the lower it is, the better, as a lower value implies leaving less unused space. However, since it does not measure overheads of bank switching, it is not as important as average count of banked calls and code size.

Let us analyze Fig. 5(e) and (f) with results for, respectively, the synthetic and real-world applications. BinPack has the lowest value of this metric for all applications, which implies that it is the best in this case. TMSearch is in turn the worst algorithm: it has the greatest value of this metric for the greatest number of applications (i.e., *Star*, *Cycles*, *Random*, *NetStackDemo* and *UDPEcho*). We can conclude that our algorithm is neither the best nor the worst in this case, which need not be viewed as its disadvantage, considering its results for the other two, more important metrics. The manual assignment is in turn almost always worse than a partitioning algorithm in most cases. This can be interpreted as an advantage of automated code partitioning against manually assigning functions into the banks.

Finally, in Table 3 we also present the time necessary to generate a partition for each of the applications with each of the algorithms. Partitions for all test applications were generated by all algorithms, except for TMSearch, in less than 168 milliseconds, which can be considered fast. Moreover, for the real-world applications, they were generated at least twice as fast by all heuristics except for TMSearch: in less than 83 milliseconds. In contrast, TMSearch in most cases requires several seconds to produce a partition. Such an amount of time is problematic for interactive development but it can be reduced by changing the parameters for halting conditions of TMSearch. However, with the parameter values that guarantee faster execution, the results for the previous three metrics become worse. Our algorithm is relatively fast in practice even for large applications such as *Paths*, *Tree*, *Cycles* and *MonitoringApp*.

All in all, the results suggest that the presented algorithm performs well. In the two most important metrics, it outperforms all other algorithms for most of the applications. Moreover, the algorithm is better than manual assignment in terms of code size and also comes closer to it than any other algorithm for average count of banked calls. From the practical point of view, in turn, it is important that the algorithm generates partitions relatively fast, even if a program to partition is large.

Table 3. Time Necessary to Generate Partitions for the Test Applications (in Seconds)

Test application	Algorithm			
	BinPack	**MaxGains**	**TMSearch**	**our algorithm**
Star	0.017	0.033	3.756	0.071
Paths	0.025	0.044	11.848	0.136
Tree	0.024	0.167	19.413	0.105
Cycles	0.022	0.035	7.401	0.135
Random	0.022	0.041	2.066	0.06
Clique	0.014	0.059	3.286	0.077
Blink	0.011	0.016	0.683	0.045
NetStackDemo	0.019	0.03	3.154	0.064
UDPEcho	0.018	0.047	7.223	0.082
MonitoringApp	0.022	0.046	6.624	0.075

7 CONCLUSIONS AND FUTURE WORK

To sum up, automated code partitioning is gaining interest in the context of software for IoT microcontrollers with switchable ROM banks. We proved that the problem of generating a partition of a program that maximizes the program's runtime performance is $\mathcal{NP}$-hard. We also demonstrated that practical partitioning solutions can be integrated into today's compilers, nevertheless. In particular, our solution—based on three heuristics that address three distinct subproblems—is fast, can outperform simple bin packing and other algorithms, and comes close to partitions created manually by the programmers with expert knowledge on the behavior of the program.

However, the presented work is just one step toward truly understanding the problem. For instance, the heuristics proposed to date all operate with functions as partitioning units, which is partly because of the architecture of embedded compilers. It would be interesting to investigate whether operating with smaller units, such as basic blocks, could enable even more efficient partitions. Likewise, while our partitions try to minimize simply the runtime number of inter-bank calls, it may be worthwhile to explore finer-grained metrics, which consider the actual differences in the costs of inter- and intra-bank calls. Finally, it may also be interesting to study whether our heuristics could be applied to architectures with lazy bank switching. In any case, there are more examples of similar possible next research steps.

ACKNOWLEDGMENTS

K. Iwanicki was supported by the Polish Ministry of Science and Higher Education with a scholarship for outstanding young scientists. The authors would like to thank the anonymous ACM Trans. Embedd. Comput. Syst. reviewers, whose feedback has helped to improve the quality of this article. The authors are also grateful to P. Horban and M. Marschall, employees of InviNets Sp. z o.o., who integrated the compiler into the company's toolchain.

REFERENCES

[1] Amstrad. 1987. *ZX Spectrum +3: Manual.* Amstrad PLC, Brentwood, UK. http://www.worldofspectrum.org/ZXSpectrum128+3Manual/

[2] ARM Limited. 2009. Cortex-M0: Technical Reference Manual. Revision r0p0, ARM DDI 0432C (ID113009). (Nov. 2009).

[3] ARM Limited. 2010. Cortex-M3: Technical Reference Manual. Revision r2p0, ARM DDI 0337H (ID032710). (Feb. 2010).

[4] ARM Limited. 2012. Cortex-M0+: Technical Reference Manual. Revision r0p0, ARM DDI 0484B (ID041812). (April 2012).

[5] C. Gordon Bell and Allen Newell. 1971. *Computer Structures: Readings and Examples*. McGraw-Hill Inc.,US. 668 pages.

[6] Andrew C. Bray, Adrian C. Dickens, and Mark A. Holmes. 1983. *The Advanced User Guide for the BBC Microcomputer* (3 ed.). The Cambridge Microcomputer Centre.

[7] Stephane Carrez. 2003. GNU Development Chain for 68HC11&68HC12. http://www.gnu.org/software/m68hc11/m68hc11_doc.html. (2003).

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. 2008. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel and Distrib. Comput.* 68, 10 (2008), 1370–1380. DOI: http://dx.doi.org/10.1016/j.jpdc.2008.05.014 General-Purpose Processing using Graphics Processing Units.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press. 1312 pages.

[10] Wei Dong, Chun Chen, Jiajun Bu, and Wen Liu. 2014. Optimizing Relocatable Code for Efficient Software Update in Networked Embedded Systems. *ACM Trans. Sen. Netw.* 11, 2, Article 22 (July 2014), 34 pages. DOI: http://dx.doi.org/10.1145/2629479

[11] Mathilde Durvy, Julien Abeillé, Patrick Wetterwald, Colin O'Flynn, Blake Leverett, Eric Gnoske, Michael Vidales, Geoff Mulligan, Nicolas Tsiftes, Niclas Finne, and Adam Dunkels. 2008. Making Sensor Networks IPv6 Ready. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*. ACM, New York, NY, USA, 421–422. DOI: http://dx.doi.org/10.1145/1460412.1460483

[12] Sandeep Dutta. 2015. SDCC Compiler User Guide (SDCC 3.5.1). Revision 929. (Aug. 2015).

[13] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman. 340 pages.

[14] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 1–11. DOI: http://dx.doi.org/10.1145/781131.781133

[15] Mateusz Grabowski, Michal Marschall, Wojciech Sirko, Maciej Debski, Marcin Ziombski, Przemyslaw Horban, Szymon Acedanski, Marcin Peczarski, Dominik Batorski, and Konrad Iwanicki. 2015. An Experimental Platform for Quantified Crowd. In *Computer Communication and Networks (ICCCN), 2015 24th International Conference on*. 1–6. DOI: http://dx.doi.org/10.1109/ICCCN.2015.7288377

[16] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM* 16, 6 (June 1973), 372–378. DOI: http://dx.doi.org/10.1145/362248.362272

[17] Jonathan W. Hui. 2008. *An Extended Internet Architecture for Low-Power Wireless Networks - Design and Implementation*. Ph.D. Dissertation. University of California, Berkeley, Berkeley, CA, USA.

[18] Jonathan W. Hui and David E. Culler. 2008. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*. ACM, New York, NY, USA, 15–28. DOI: http://dx.doi.org/10.1145/1460412.1460415

[19] Konrad Iwanicki. 2016. RNFD: Routing-Layer Detection of DODAG (Root) Node Failures in Low-Power Wireless Networks. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 1–12. DOI: http://dx.doi.org/10.1109/IPSN.2016.7460720

[20] Konrad Iwanicki, Przemyslaw Horban, Piotr Glazar, and Karol Strzelecki. 2014. Bringing Modern Unit Testing Techniques to Sensornets. *ACM Trans. Sen. Netw.* 11, 2, Article 25 (July 2014), 41 pages. DOI: http://dx.doi.org/10.1145/2629422

[21] David R. Karger and Clifford Stein. 1996. A New Approach to the Minimum Cut Problem. *J. ACM* 43, 4 (July 1996), 601–640. DOI: http://dx.doi.org/10.1145/234533.234534

[22] Simon Kellner. 2010. Flexible Online Energy Accounting in TinyOS. In *Real-World Wireless Sensor Networks: 4th International Workshop, REALWSN 2010, Colombo, Sri Lanka, December 16-17, 2010. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 62–73. DOI: http://dx.doi.org/10.1007/978-3-642-17520-6_6

[23] JeongGil Ko, Qiang Wang, Thomas Schmid, Wanja Hofer, Prabal Dutta, and Andreas Terzis. 2010. Egs: A Cortex M3-Based Mote Platform. In *Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on*. 1–3. DOI: http://dx.doi.org/10.1109/SECON.2010.5508223

[24] Rainer Leupers and Daniel Kotte. 2001. Variable partitioning for dual memory bank DSPs. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, Vol. 2. 1121–1124 vol.2. DOI: http://dx.doi.org/10.1109/ICASSP.2001.941118

[25] Lian Li, Lin Gao, and Jingling Xue. 2005. Memory coloring: A compiler approach for scratchpad memory management. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*. 329–338. DOI: http://dx.doi.org/10.1109/PACT.2005.27

[26] Qing'an Li, Yanxiang He, Yong Chen, Wei Wu, and Wenwen Xu. 2010. A heuristic algorithm for optimizing page selection instructions. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, Vol. 2. 143–148. DOI: http://dx.doi.org/10.1109/ICSTE.2010.5608834

[27] Lotus Development Corporation, Intel Corporation, and Microsoft Corporation. 1987. Expanded Memory Specification (Version 4.0). 300275-005. (Oct. 1987). http://www.phatcode.net/res/218/files/limems40.txt

[28] Lotus Development Corporation, Intel Corporation, and Microsoft Corporationand AST Research, Inc. 1988. eXtended Memory Specification (XMS), ver 2.0. (July 1988). http://www.phatcode.net/res/219/files/xms20.txt

[29] George S. Lueker. 1978. A data structure for orthogonal range queries. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*. IEEE, 28–34. DOI: http://dx.doi.org/10.1109/SFCS.1978.1

[30] Yuan Mengting, Chun J. Xue, Chen Yong, Li Qing'an, and Yingchao Zhao. 2013. Minimizing code size via page selection optimization on partitioned memory architectures. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*. 1–10. DOI: http://dx.doi.org/10.1109/CASES.2013.6662516

[31] Microchip Technology, Inc. 2002. PIC16F7X: 28/40-pin, 8-bit CMOS FLASH Microcontrollers. Data Sheet, DS30325B. (2002).

[32] Bhargavi Nisarga. 2007. Extended Memory Access Using IAR v3.42A and CCE v2. Texas Instruments, Inc., Application Report, SLAA376. (Nov. 2007).

[33] Preeti R. Panda, F. Catthoor, Nikil D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. 2001. Data and Memory Optimization Techniques for Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.* 6, 2 (April 2001), 149–206. DOI: http://dx.doi.org/10.1145/375977.375978

[34] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 73–82. DOI: http://dx.doi.org/10.1145/1345206.1345220

[35] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. 1996. Exploiting Dual Data-memory Banks in Digital Signal Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. ACM, New York, NY, USA, 234–243. DOI: http://dx.doi.org/10.1145/237090.237193

[36] Bernhard Scholz, Bernd Burgstaller, and Jingling Xue. 2006. Minimizing Bank Selection Instructions for Partitioned Memory Architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*. New York, NY, USA, 201–211. DOI: http://dx.doi.org/10.1145/1176760.1176786

[37] Paul Sokolovsky. 2013. 8051 Code Banking in ContikiOS. https://github.com/contiki-os/contiki/wiki/8051-Code-Banking. (May 2013).

[38] Mechthild Stoer and Frank Wagner. 1997. A Simple Min-cut Algorithm. *J. ACM* 44, 4 (July 1997), 585–591. DOI: http://dx.doi.org/10.1145/263867.263872

[39] Texas Instruments, Inc. 2013. MSP430x2xx Family: User's Guide. SLAU144J. (July 2013).

[40] Texas Instruments, Inc. 2014. CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee Applications, CC2540/41 System-on-Chip Solution for 2.4-GHz Bluetooth low energy Applications: User's Guide. Data Sheet, no. SWRU191F. (April 2014).

[41] Sumesh Udayakumaran and Rajeev Barua. 2003. Compiler-decided Dynamic Memory Allocation for Scratch-pad Based Embedded Systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03)*. ACM, New York, NY, USA, 276–286. DOI: http://dx.doi.org/10.1145/951710.951747

[42] Jean-Philippe Vasseur and Adam Dunkels. 2010. *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[43] Raeto C. West. 1985. *Programming the Commodore 64: The Definitive Guide*. COMPUTE! Publications, Inc., Greensboro, NC, USA. 609 pages.

[44] David P. Williamson and David B. Shmoys. 2011. *The Design of Approximation Algorithms*. Cambridge University Press. 518 pages.

[45] Xiaotong Zhuang, Santosh Pande, and John S. Greenland Jr. 2002. A framework for parallelizing load/stores on embedded processors. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*. 68–79. DOI: http://dx.doi.org/10.1109/PACT.2002.1106005

Received December 2015; revised August 2016; accepted February 2017