Endurance-Aware Allocation of Data Variables on NVM-Based Scratchpad Memory in Real-Time Embedded Systems

Zhu Wang, Zonghua Gu, Min Yao, Zili Shao

***Abstract*—Non-Volatile Memory (NVM) has many benefits compared to the traditional SRAM, such as improved reliability and reduced power consumption, but it has long write latency and limited write endurance. Scratchpad Memory (SPM) is software-managed small on-chip memory for improving system performance and predicability. We consider SPM based on STT-RAM, a type of NVM with high performance and good endurance. We present algorithms for allocating data variables to SPM and distribute write activity evenly in the SPM address space, in order to achieve wear-leveling and prolong the lifetime of NVM. We present two optimization algorithms for minimizing system CPU utilization subject to NVM lifetime constraints: one is an optimal algorithm based on ILP, the other is an efficient heuristic algorithm that can obtain close-to-optimal solutions.**

***Index Terms*—Scratchpad memory, non-volatile memory, real-time embedded.**

I. Introduction

Scratchpad memory (SPM) is a type of small, software-managed on-chip memory that serves a similar purpose as cache, as part of the memory hierarchy to improve program performance. But different from cache, SPM is software-managed by the compiler and/or programmer, on contrast to hardware-managed cache with the cache controller. SPM is accessed by direct addressing, hence it does not have tags and tag comparison logic as in cache, and is more power-efficient than cache. It also has better timing predicability than cache, since the compiler generally can achieve better control over allocation and accesses to SPM than the hardware cache controller. Due to its power efficiency and timing predicability, SPM is used as an alternative to cache in many MCU products, e.g., Infineon Tricore. SPM is also found in SoPC (System-on-a-Programmable Chip) products from several FPGA vendors, e.g., Altera Nios II and Xilinx MicroBlaze processors, which are widely used in embedded systems design, as they permit hardware/software co-design by customizing the hardware platform. Most microprocessors provide on-chip memory that is configurable as cache or as SPM, so the design can customize the system design based on application requirements. SPM size ranges from very small. e.g., Atmel AVR microcontrollers with SPM of size 0.5-1KB; to relatively large, e.g., Infineon Tricore processor with SPM size of 48KB.

Non-Volatile Memory (NVM) has benefits of low static power consumption, high storage density, and high resistance to soft errors such as shift from Single Event Upsets. A variety of NVM technologies have been developed for different levels of the memory hierarchy, e.g., Phase Change Memory (PCM) is candidate for main memory because it has higher density and lower standby power compared to DRAM; Flash memory is already widely-deployed as secondary storage; Spin-Transfer Torque RAM (STT-RAM) is typically used as a replacement of SRAM as on-chip memories (cache or SPM) because of its advantages of fast read speed, low leakage power and high density. STT-RAM can be made up to four times denser than traditional Static RAM (SRAM) [1], i.e., for the same silicon area, a memory chip made of STT-RAM can have up to four times larger capacity than that made of SRAM. The main drawbacks of NVM include high latency and power consumption for write operations, and limited write endurance before wear-out. *Wear-leveling* techniques are necessary to help even out the write operations over the NVM chip, in order to improve its write endurance and prolong its lifetime.

For real-time embedded systems, a program's *Worst-Case Execution Time (WCET)* is more important than average performance. A given program's WCET is determined by its *Worst-Case Execution Path (WCEP)*, defined as the longest end-to-end path in the control flow graph in terms of execution time. Since our target is real-time embedded systems, we address WCET instead of average-case execution time in this paper. In this paper, we consider SPM based on NVM technology, e.g., STT-RAM, and present algorithms for wear-leveling and improving the endurance of NVM-based SPM, and evaluate their impact on program WCET and system schedulability. In contrast to the conventional SPM allocation problem, we not only decide which data variables are allocated to SPM, but also achieve wear-leveling over the SPM address space by allocating variables to specific SPM addresses at compile-time and computing the aggregate write count at each SPM memory address.

This paper is structured as follows: We discuss related work in Section II; present the allocation algorithm details in Section III, including an optimal ILP formulation in Section III-B, and an efficient heuristic algorithm in Section III-C; present performance evaluation in Section IV, and conclusions in Section V.

This work was supported by the National Natural Science Foundation of China under Grant 61272127.

Z. Wang, Z. Gu (corresponding author), M. Yao are with the College of Computer Science, Zhejiang University, Hangzhou 310027, China (e-mail: wangzhu981712@zju.edu.cn; zgu@zju.edu.cn; yaom@zju.edu.cn).

Z. Shao is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong (e-mail: cszlshao@comp.polyu.edu.hk).

II. Background and Related Work

There has been a lot of work on compiler techniques for allocation of program instructions or data variables to SPM to improve performance (either average-case or worst-case) or reduce energy consumption. These algorithms can be classified as either static or dynamic. For static allocation [2], [3], [4], instructions or data variables are statically-allocated to the SPM for the entire duration of program execution. For dynamic allocation [5], [6], the program code is divided into regions, and a different allocation is computed for each region, in order to achieve better utilization of the SPM space. Wang et al. [7] presented algorithms for WCET-aware energy-efficient *static* data allocation on SPM, i.e., selectively allocating data variables to SPM to minimize a program's energy consumption due to data variable accesses, while respecting a given WCET upper bound. In this paper, we consider a form of *dynamic* allocation of *stack variables*, where allocation decisions can be changed at procedure call and return points.

Many techniques have been proposed to extend lifetime of NVM-based SPM, cache or main memory. Hu et al [8], [9] considered a hybrid SPM configuration consisting of SRAM and PCM-based NVM, and presented a dynamic data allocation algorithm for reducing write operations on NVM by preferentially allocating read-intensive data variables into NVM, and write-intensive data variables into SRAM. Monazzah et al. [10] presented algorithms for fault-tolerant data allocation on hybrid SPM that consists of NVM, SRAM protected with Error-Correcting Code, and ECC protected with with parity. Wang et al. [11] considered a multitasking system with hybrid main memory consisting of PCM and DRAM, and addressed the problem of partitioning and allocating data variables to minimize average power consumption while guaranteeing schedulability. Qiu et al. [12], [13], [14] presented several techniques for improving data allocation efficiency and system performance of hybrid memory or SPM, including dynamic programming and genetic algorithms. Most related work aims to reduce the *number of writes* on NVM, but since data allocation addresses on SPM are not specified, it is possible to have *unbalanced writes* across the SPM address space, i.e., certain SPM addresses may be frequently-written and become hotspots and suffer early wearout. Hu et al [15] considered hybrid main memory consisting of DRAM and PCM-based NVM. They first used the data allocation algorithm in [8] to allocate data variables to DRAM or NVM, then keep track of the number of writes to each address in PCM and evenly distribute to all PCM addresses. Compared to [15], our work addresses SPM instead of main memory, program WCET instead of average performance, and multitasking system instead of a single program.

Although the heterogeneous, hybrid memory configuration of SRAM+NVM (for either SPM, cache or main memory) has advantages over NVM-only memory configuration in terms of improved endurance and lifetime, it adds complexity to hardware design and manufacturing, hence it is desirable to build NVM-only homogeneous memory configuration to reduce the integration, verification, and testing costs. While the conventional wisdom is that NVM-only configuration is applicable at lower levels of memory hierarchy, e.g., Last-Level Cache, Smullen et al. [16] several authors have proposed architectural-level techniques to improve write-performance of STT-RAM by relaxing non-volatility, making it suitable for use at the L1 cache level where data access speed is critical, e.g., Li et al. [17] presented a dual-mode STT-RAM-based L1 cache design. Even for hybrid SPM consisting of both SRAM and NVM, our algorithm in this paper for NVM-based SPM is still applicable, e.g., we can use the optimal data allocation algorithm in [8], [9] to allocate data variables to SRAM and NVM, then apply the algorithms in this paper to achieve wear-leveling on the NVM. Although either PCM or STT-RAM can be used as the NVM technology in SPM, we consider STT-RAM as the memory technology in performance evaluation experiments, possibly with architectural techniques for performance enhancement as described in [16], [17]. In the rest of the paper, *whenever we use the term* NVM*, we implicitly refer to* STT-RAM.

Ghattas et al. [18] and Kang et al. [19] considered interactions between data allocation to SPM and real-time scheduling algorithms in a multitasking system, and presented *Data Allocation with Real-Time Scheduling (DARTS)*, using *Preemption Threshold Scheduling (PTS)* [20] to reduce the degree of task preemption in a multi-tasking environment, and increase disjoint lifetimes of stack variables at task level. Our work is based on DARTS [18], [19], and addresses allocation of data variables to SPM in a multitasking environment, but considering NVM-based SPM, with tradeoffs between CPU utilization and system lifetime due to NVM endurance constraints.

Review of PTS: PTS allows a task to disable preemption from higher priority tasks up to a specified threshold priority; only tasks with priorities higher than the given task's threshold are allowed to preempt it. PTS is a generalization of preemptive and non-preemptive scheduling: if each task's PT is set to be the highest priority among all task priorities, then it is equivalent to non-preemptive scheduling; if each task's PT is set to be equal to its own priority, then it is equivalent to preemptive scheduling. The designer can control whether two tasks can preempt each other by setting appropriate preemption threshold values. Two mutually non-preemptive tasks can share a common stack space by overlaying the allocation of their stack variables in SPM; whereas if one task can preempt another, allocations of their stack variables must be non-overlapping in SPM, since the preempted task's stack variables must be saved when it is preempted, and restored when it resumes execution. Therefore, by using PTS to increase non-preemptive task pairs, we can reduce total system stack space requirement compared to fully-preemptive scheduling, which is important for low-cost embedded systems.

Figure 1 shows an example task-set with four tasks in decreasing priority order $\tau_1, \tau_2, \tau_3, \tau_4$. Figure 1(a) shows fixed-priority fully preemptive scheduling; Figure 1(b) shows Preemption Threshold Scheduling, where each dotted arrow's endpoint indicates the corresponding task's PT. Figure 1(c) shows the *preemption graph* for the PT assignment in Figure 1(b), where each task is represented by a vertex, and an edge from τ_i to τ_j means τ_i can preempt τ_j. There is

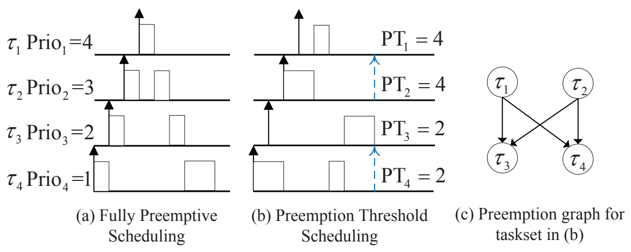


Fig. 1: An example of task-set with PTS.

no edge from τ_1 to τ_2 (or from τ_3 to τ_4) due to the PT assignment, meaning that τ_1 and τ_2 (or τ_3 to τ_4) are mutually non-preemptive, so they can share a common stack space.

III. Allocation Algorithms on NVM-based SPM

A. Problem Motivation

Either instruction or data can be allocated to SPM to improve performance and reduce power consumption. Consider a multi-threaded, single-process program. The program's variables can be classified into *static* (*global*) variables; *stack*(*automatic*) variables; and *heap* variables. In the rest of the paper, we use the terms *global* variable and *stack* variable, instead of *static* variable and *automatic* variable, to keep the terminology consistent. We consider small, resource-constrained embedded systems, where dynamic memory allocation of heap variables is generally undesirable for predicability issues, hence we only consider *global variables* and *stack variables* in this paper. As common in such small embedded systems, we assume there is a single OS process and multiple threads, hence all variables share the same address space. The global variables persist throughout the program's lifetime, but the stack variables are transient, i.e., they are created and destroyed dynamically at runtime. For allocation into SPM, global variables should be allocated permanently for the entire program duration; while stack variables can be allocated in an *overlayed* manner; when certain stack variables are de-allocated (destroyed) at end of each procedure call, the SPM space should be reused to allocate other stack variables.

Normally, the procedure stack grows in units of stack frames, one per procedure, where a stack frame is a block of contiguous memory locations containing all the variables and parameters of the procedure. The stack grows when a procedure is called, and shrinks when the procedure call is finished. It is possible to distribute stack variables onto multiple memory units, e.g., for a procedure with two local variables x and y, x can be allocated to SPM, and y can be allocated to main memory. The implementation details can be handled by the compiler, and are transparent to the programmer. This can be helpful for more flexible and efficient management of the limited SPM space. Ghattas et al [18] presented Data Allocation with Real-Time Scheduling (DARTS), and Kang et al [19] presented assembly program rewriting schemes to implement the stack frame splitting on an actual embedded processor, for practical realization of the techniques in [18]. Extra stack frame pointers are used to split the traditional single procedure stack frame into multiple parts that can be allocated separately. If a procedure's stack frame is split into two parts, then two sub-frame pointers are used to address them. Three levels of allocation granularity for stack variables are supported: (Note that for global variables, variable-level allocation is always used.)

- *Task-level* allocation: the entire stack frame of each task is the unit of allocation. All the stack variables in the same task are allocated as a whole, and share the same frame pointer. If two tasks are mutually non-preemptive, e.g., due to preemption threshold assignments, then they have disjoint lifetimes, and their data allocation can overlay in the SPM address space; if one task can preempt the other, then they have overlapping lifetimes, hence they are *mutually-exclusive*, i.e., they cannot overlay in the SPM address space.
- *Procedure-level* allocation: the stack frame of a procedure is the unit of allocation. All the stack variables in the same procedure are allocated as a whole, and share the same frame pointer. If two procedures are in the same task, but there is no path between them in the call graph, or they belong to two different tasks that are mutually non-preemptive, then they can overlay in the SPM address space. If two procedures are in the same task, and there exists a direct or indirect path between them in the call graph; or they belong to different tasks that can preempt each other, then they are *mutually-exclusive*, i.e., they cannot overlay in the SPM address space.
- *Variable-level* allocation: each stack variable is the unit of allocation. Two stack variables are *mutually-exclusive*, i.e., they cannot overlay in the SPM space, if they belong to the same procedure; or they belong to two different procedures within the same task, and there is a path between them in the call graph; or they belong to two different tasks that can preempt each other.

There are obvious tradeoffs between allocation granularity and runtime overhead: the finest granularity level of variable-level allocation is the most flexible, but incurs the highest runtime overhead; the coarsest granularity level of task-level allocation is the most restrictive, but incurs the lowest runtime overhead. For example, with variable-level allocation, the stack pointer needs to be modified to point to a different memory space (either SPM or main memory) whenever a new variable is accessed; with procedure-level allocation, the stack pointer may need to be modified upon procedure call and return; with task-level allocation, the stack pointer may need to be modified upon task context switches. In general, procedure-level allocation represents a reasonable tradeoff between allocation granularity and runtime overhead [18], [19], and we will mainly focus on procedure-level allocation for allocation of stack variables in the descriptions of ILP and heuristic algorithms, but the algorithms can be readily extended to apply to task-level and variable-level allocation. We use the term *data object* to refer to either a variable (global or stack), or a stack frame, or the set of stack frames of a task, depending on the allocation granularity chosen.

We use three examples in Figs 2 to 4 to illustrate the concept of wear-leveling in SPM. Consider a single task with two procedures $foo()$ and $bar()$; $foo()$ contains two stack variables x and y; $bar()$ contains two stack variables m and n. We denote the write count of each stack variable by a number in parentheses besides that variable. Suppose the designer has specified an upper bound on the write count of 60 on any specific address in the SPM. We can see the problem of uneven write counts in the conventional SPM stack allocation method, i.e., the default method in C compilers, which sometimes causes the upper bound of 60 to be exceeded. By using procedure-level allocation, we are able to achieve wear-leveling by spreading out the write operations more evenly across the SPM. We need to consider the *runtime instances* of the stack frame for each runtime invocation of a procedure. For example, in Fig. 4, the procedure $bar()$ is called twice, once by $main()$, once by procedure $foo()$, each call creating a distinct copy of $bar()$'s stack frame, which need to be considered as multiple stack frames during SPM allocation.

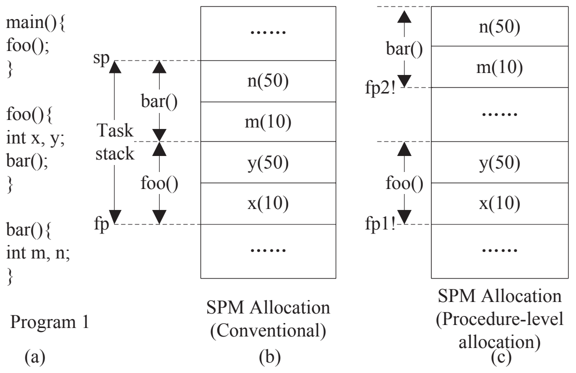


Fig. 2: $main()$ calls $foo()$; $foo()$ calls $bar()$. Since $foo()$ and $bar()$ are connected in the program call graph, they have overlapping lifetimes, so their stack frames cannot overlay with each other. With conventional SPM allocation (b), the task's entire stack frame is allocated in a contiguous region, addressed by the frame pointer fp; with procedure-level allocation (c), each procedure's stack frame can be placed at non-contiguous regions, addressed by the sub-frame pointers $fp1$ for $foo()$, and $fp2$ for $bar()$. (The allocation in (c) achieves no benefits in terms of wear-leveling, and only serves to demonstrate that a task's stack frame can be split into multiple parts and allocated separately.) The maximum write count on any specific SPM address for both approaches is 50, within the upper bound of 60.

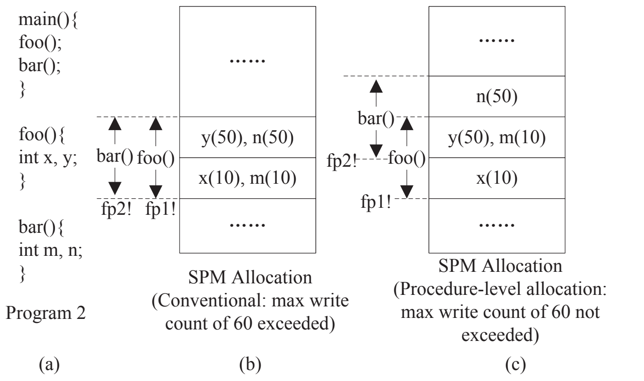


Fig. 3: $main()$ calls $foo()$, then calls $bar()$. Since $foo()$ and $bar()$ are not connected in the program call graph, they have disjoint lifetimes, and their stack frames can overlay with each other. With conventional SPM allocation (b), the two procedure stack frames are allocated at the same address of the SPM, and the maximum write count on any SPM address is 100, sum of write counts of two write-intensive variables y and n. This exceeds the upper bound of 60. With procedure-level allocation (c), we can place the stack frame of $bar()$ at a higher address in the SPM, in order to allocate y and n at different addresses, thus making the maximum write count 60.

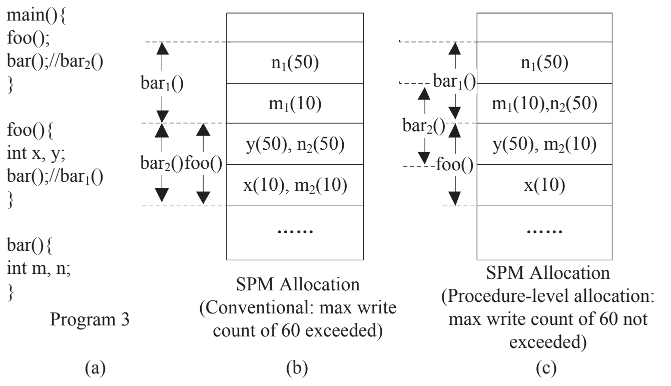


Fig. 4: $main()$ calls $foo()$; $foo()$ calls $bar()$; $main()$ then calls $bar()$ again. We denote the first invocation of $bar()$ as $bar_1()$, and the second invocation of $bar()$ as $bar_2()$. The stack frames of $foo()$ and $bar_1()$ cannot overlay with each other; those of $foo()$ and $bar_2()$ can overlay. The conventional allocation (b) overlays $foo()$ and $bar_2()$, causing the maximum write count to be 100; the procedure-level allocation (c) achieves the maximum write count of 60.

B. ILP Formulation

We consider a multitasking system consisting of N^T tasks $(\tau_1, \ldots, \tau_{N^T})$; each task τ_i contains N_i^P procedures; each procedure P_{ij} in task τ_i contains N_{ij}^V stack variables; each *runtime invocation* of the procedure P_{ij} creates a stack frame x_{ij}, with size equal to sum of all the stack variables in P_{ij}; in addition to stack variables, task τ_i contains N_i^G global variables $x_{ig}^G, g \in [1, N_i^G]$. L represents the expected lifetime of NVM-based SPM as specified by the designer, measured in terms of months or years. W_{th} represents the upper bound on aggregate write count on each NVM address before wearout. WCET of each task C_i consists of two parts: CPU execution time $Cexe_i$ and memory access time $Cmem_i$. Since SPM allocation decisions only affect memory access time, not CPU execution time, we only consider $Cmem_i$ in this paper. Table I summarizes the notations.

We mainly present the ILP model for procedure-level SPM allocation. The ILP encodings for task-level and variable-level allocation are similar to procedure-level allocation, with the following differences: for task-level allocation, the entire stack of each task, including all the procedures within it, are allocated as a whole to either SPM or main memory. For variable-level allocation, each stack variable x_{ijk} is allocated individually.

a) Optimization Objective: Since we consider hard real-time systems, we define out optimization objective as mini-

TABLE I: Notations

S_{spm}	Size of the SPM
N^T	Total number of tasks in the taskset
τ_i	The i^{th} task, $i \in [1, N^T]$
T_i	Period of task τ_i
C_i	WCET of task τ_i
$Prio_i$	Priority of task τ_i
PT_i	Preemption Threshold of task τ_i
$Cmem_i$	Total memory access latency on the program WCEP of task τ_i
$Cexe_i$	Total CPU execution time on the program WCEP of task τ_i
N_i^G	Total number of global variables of the task τ_i
x_{ig}^G	The g_{th} global variable of the task τ_i
N_i^P	Total number of procedures in task τ_i
P_{ij}	The j^{th} procedure in task τ_i
x_{ij}	Stack frame created with each runtime invocation of procedure P_{ij}
N_{ij}^V	Number of stack variables in procedure P_{ij} in task τ_i
x_{ijk}	The k^{th} stack (local) variable in procedure P_{ij} in task τ_i
$I(x)$	A 0-1 variable denoting whether data object x is allocated to SPM
$b_{pos}(x)$	A 0-1 variable denoting whether data object x occupies the SPM memory address pos.
$S(x)$	Size of data variable x in Bytes
$O(x)$	Start address of data variable x relative to the beginning of the SPM address range
$Cmm(x)$	Total memory access latency of data variable x on the program WCEP if it is allocated to main memory
$Cspm(x)$	Total memory access latency of data variable x on the program WCEP if it is allocated to SPM
$N^w(x)$	Average write count of data variable x for each program execution
U	Total CPU utilization of the taskset
L	Lifetime constraint of NVM given by the designer (in days or years)
W_{th}	Upper bound on total write count at each SPM memory address
U_G	Set of unallocated global variables
U_F	Set of unallocated procedure stack frames
$Reduc(x_{ig}^G)$	Metric for sorting the global variables to be placed in SPM
$Reduc(x_{ij})$	Metric for sorting the procedure stack frames to be placed in SPM

mizing the total CPU utilization:

$$\text{Minimize } \sum_{i=1}^{N^T} C_i/T_i \tag{1}$$

where C_i is the WCET of task τ_i, and T_i is its period. If the original taskset is unschedulable, then reducing CPU utilization can help make it schedulable; even if the original taskset is schedulable, reducing CPU utilization still has many benefits, since it enables use of power management techniques such as Dynamic Voltage-Frequency Scaling or Dynamic Power Management to reduce power consumption, or use of checkpointing/reexecution to enhance reliability against soft errors, or adding additional workload during system maintenance and upgrade. In fact, the problem can be viewed as a multi-objective optimization problem: one objective is to minimize CPU utilization, the other objective is to maximize system lifetime due to limited write endurance of NVM. To convert it into a single-objective optimization problem, either one can be treated as constraint, and the other as optimization objective. Our current problem formulation treats minimizing CPU utilization as the optimization objective, with system lifetime as constraint (expressed indirectly as the maximum write count on each memory address).

The task WCET C_i consists of CPU execution time plus memory access latency on the WCEP:

$$C_i = Cexe_i + Cmem_i \tag{2}$$

Since data allocation to SPM only affects the memory access latency on the WCEP $Cmem_i$, not the CPU execution time $Cexe_i$, we only consider $Cmem_i$ in this paper. To determine $Cmem_i$, we use a set of 0-1 variables $I(x_{ig}^G)$ ($I(x_{ij})$) denote whether data object x_{ig}^G (x_{ij}) is allocated to SPM. A set of integer variables $O(x_{ij})$ denote offset of the stack frame x_{ij} relative to the starting address of the SPM space (only meaningful when x_{ij} is allocated to SPM). Since stack frames can overlay with each other in the SPM space, multiple different stack frames may occupy overlapping memory addresses in SPM, hence we need to keep track of the aggregate write count on each memory address by summing up write counts of all stack frames with address range covering that memory address. Since global variables cannot overlay with each other, it is not necessary to keep track of their specific memory addresses in SPM for calculation of write count on each memory address. Therefore, the set of offset variables $O(x_{ij})$ are only defined for stack frames, not for global variables. We re-emphasize the main difference of our work from related work on SPM allocation and wear-leveling is to encode the specific SPM address for each data variable (or stack frame) allocated to SPM, in addition to the decision of whether each data variable (or stack frame) is allocated to SPM.

$$I(x) = \begin{cases} 1, & \text{if data object } x \text{ is allocated to SPM} \\ 0, & \text{if data object } x \text{ is allocated to main memory} \end{cases} \tag{3}$$

$Cmem_i$ is computed with Equation 4:

$$Cmem_i = \sum_{i=1}^{N^T}\sum_{g=1}^{N_i^G}(I(x_{ig}^G)Cspm(x_{ig}^G) + (1 - I(x_{ig}^G))Cmm(x_{ig}^G)) + \sum_{i=1}^{N^T}\sum_{j=1}^{N_i^P}(I(x_{ij})\sum_{k=1}^{N_{ij}^V}Cspm(x_{ijk}) + (1 - I(x_{ij}))\sum_{k=1}^{N_{ij}^V}Cmm(x_{ijk})) \tag{4}$$

The first term in Equation (4) denotes total memory access latency of global variables (allocated to either SPM or main memory) on the WCEP; the second term denotes total memory access latency of stack variables (allocated to either SPM or main memory) on the WCEP. Note that the program WCEP may change after data allocation to SPM, but this is implicitly accounted for by the iterative nature of our method.

b) SPM Size Constraints: We divide the SPM address space into two partitions as shown in Fig. 5: the left partition (lower memory addresses) contains global variables, with size equal to sum of sizes of all global variables allocated to SPM; The right partition (higher addresses) contains stack variables, with size that is possibly smaller than total size of stack variables allocated to SPM due to possible runtime overlays, i.e, stack variables may be switched in and out of SPM during program execution. The boundary between the two partitions is determined by our optimization algorithms.

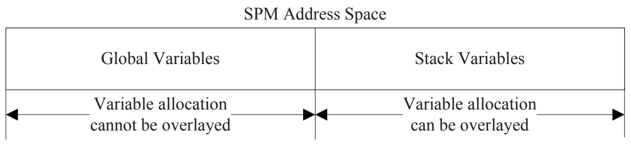

SPM Address Space
Global Variables
Stack Variables
Variable allocation cannot be overlayed
Variable allocation can be overlayed

Fig. 5: SPM space layout.

Total size of all global variables allocated to SPM cannot exceed the total SPM size:

$$\sum_{i=1}^{N^T}\sum_{g=1}^{N_i^G} I(x_{ig}^G)S(x_{ig}^G) \leq S_{spm} \tag{5}$$

We use $O(x_{ij})$ to denote starting address of stack variable x_{ij}, and it must lie to the right of the partition for global variables; $O(x_{ij})+S(x_{ij})$ denotes end address of stack variable x_{ij}, and it must not exceed the total size of SPM:

$$\begin{gathered}\forall i \in [1, N^T], \forall j \in [1, N_i^P], \\ O(x_{ij}) \geq \sum_{i=1}^{N^T}\sum_{g=1}^{N_i^G} I(x_{ig}^G)S(x_{ig}^G) \\ \&\&O(x_{ij}) + S(x_{ij}) \leq S_{spm}\end{gathered} \tag{6}$$

c) No-Overlay Constraints: Consider any pair of procedures with overlapping lifetimes. There may be two cases: the two procedures belong to the same task, and there exists a path linking the two procedures in the call graph; or the two procedures belong to two different tasks that are not mutually non-preemptive, i.e., one of the tasks can preempt the other. For such pair of procedures, their stack frames should not overlay with each other in the SPM address space. We use a large constant M in Equation 7 to encode the fact that the no-overlay constraint is valid only when both procedure stacks x_{ij} and $x_{i'j'}$ are allocated to the SPM ($I(x_{ij}) = I(x_{i'j'}) = 1$):

$$\begin{gathered}\forall i, i' \in [1, N^T], \forall j \in [1, N_i^P], \forall j' \in [1, N_{i'}^P], \\ x_{ij} \neq x_{i'j'} \&\& I(x_{ij}) = I(x_{i'j'}) = 1 \implies \\ (O(x_{ij}) + S(x_{ij}) \leq O(x_{i'j'}) || (O(x_{i'j'}) + S(x_{i'j'}) \leq O(x_{ij}))\end{gathered} \tag{7}$$

where $\implies$ is the *implies* operator supported by CPLEX as so-called *indicator constraints*. Note that the preconditions on whether the stack frames of two procedures can overlap with each other are determined through graph traversal of each program's control flow graphs and the system preemption graph based on assignments of priority and preemption threshold values (e.g., Fig. 1(c)), hence are not explicit in Equation 7.

d) Endurance Constraints: We obtain the total *average write count* on each memory address in the SPM during one task execution per period, by runtime profiling with random inputs. This number is multiplied with the total number of task executions within the required lifetime constraint (L/T_i) to obtain the total write count in the lifetime, then compared to the upper bound W_{th} to determine whether the endurance constraint is satisfied. Different from the calculation of program WCET and CPU utilization using the WCEP, which may be a cold path that is rarely executed, we must consider the average case execution path(s) that are frequently executed at runtime, since they determine the average write count on each SPM memory address, which in turn determines the endurance and lifetime.

Since no overlay is allowed for the global variables, each SPM memory address corresponds to at most one global variable, and the write count on that address can be simply computed by the write count of the variable allocated to that address, or 0 if no variable is allocated to that address. Since we consider the periodic task model, the total number of invocations of the task is equal to $\frac{L}{T_i}$, hence the endurance constraints related to global variables are expressed as:

$$\forall i \in [1, N^T], \forall g \in [1, N_i^G], \frac{L}{T_i} I(x_{ig}^G)N^w(x_{ig}^G) \leq W_{th} \tag{8}$$

If the data variable is an array, then the constraint applies to each element of the array.

For stack variables, since different stack variables may have overlay among them, we need to sum up all the write counts of different variables allocated to the same SPM memory address, in order to obtain the aggregate write count on this address. For example, consider a 32-bit integer stack variable x_{ijk} in procedure P_{ij} with size 4 ($S(x_{ijk}) = 4$), and it is allocated to SPM with start address 230 ($O(x_{ijk}) = 230$), then it occupies address range [230,234). Consider another 32-bit integer variable $x_{i'j'k'}$ in a different procedure $P_{i'j'}$, and procedures P_{ij} and $P_{i'j'}$ are not mutually-exclusive, then it is possible for the addresses range of variable x_{ijk} to be overlayed with that of variable $x_{i'j'k'}$. Suppose $x_{i'j'k'}$ occupies address range [232, 236), then we need to add up the write counts of both variables when computing write counts for addresses 232 and 233, since these addresses are written to when either x_{ijk} or $x_{i'j'k'}$ is written to; on the other hand, addresses 230, 231 are written to only when x_{ijk} is written to; addresses 234, 235 are written to only when $x_{i'j'k'}$ is written to.

In order to compute the aggregate write count on each SPM address, we define a set of 0-1 variable $b_{pos}(x_{ijk}), pos \in [0, S_{spm})$ encoding the range of positions (SPM addresses) occupied by variable x_{ijk}: $b_{pos}(x_{ijk})$ is set to 1 if x_{ijk} is assigned to SPM ($I(x_{ij}) = 1$), and the SPM position (address) pos is occupied by stack variable x_{ijk}, i.e., pos falls within range between start address and end address of variable x_{ijk} on the SPM. Otherwise, $b_{pos}(x_{ijk})$ is set to 0:

$$\begin{gathered}\forall i \in [1, N^T], \forall j \in [1, N_i^P], \forall k \in [1, N_{ij}^V], \forall pos \in [0, S_{spm}), \\ b_{pos}(x_{ijk}) = (I(x_{ij})\&\&pos \in [O(x_{ijk}), O(x_{ijk}) + S(x_{ijk})))\end{gathered} \tag{9}$$

Since we consider procedure-level allocation, the stack frame of each runtime invocation of a procedure is allocated as a single unit. Variable $O(x_{ij})$ denotes the start address of stack frame x_{ij}. Variable $O(x_{ijk})$ denotes the start address of variable x_{ijk} in stack frame x_{ij}, and can be expressed as $O(x_{ij})$ plus a known and fixed offset of variable x_{ijk} relative to $O(x_{ij})$.

To express the predicate $pos \in [O(x_{ijk}), O(x_{ijk})+S(x_{ijk}))$ in Equation 9, we introduce axillary predicates $Ge(w,v)$ ($G(w,v)$) to denote if a variable w is *greater than or equal to (greater than)* another variable v,

$$Ge(w,v) = \begin{cases} 1, & w \geq v \\ 0, & w < v \end{cases} \tag{10}$$

$$G(w,v) = \begin{cases} 1, & w > v \\ 0, & w \leq v \end{cases} \tag{11}$$

So $pos \in [O(x_{ijk}), O(x_{ijk}) + S(x_{ijk}))$ is equivalent to $Ge(pos, O(x_{ijk}))\&\&G(O(x_{ijk}) + S(x_{ijk}), pos)$.

For the operator $\geq$, we define an axillary 0-1 variable $p = Ge(w,v)$,

$$\begin{array}{l} -M \cdot (1-p) \leq w - v \\ w - v + 1 \leq M \cdot p \end{array} \tag{12}$$

where M is a large number.

For the operator $>$, we define an axillary 0-1 variable $p = G(w,v)$,

$$\begin{array}{l} -M \cdot (1-p) + 1 \leq w - v \\ w - v \leq M \cdot p \end{array} \tag{13}$$

The endurance constraints related to stack variables are expressed as:

$$\forall pos \in [0, S_{spm}), \sum_{i=1}^{N^T} \frac{L}{T_i} \sum_{j=1}^{N_i^P} \sum_{k=1}^{N_{ij}^V} b_{pos}(x_{ijk}) N^w(x_{ijk}) \leq W_{th} \tag{14}$$

Equation 14 states that for each SPM memory address $pos \in [0, S_{spm})$, the total write count of all stack variables that is allocated to it during the specified lifetime L must not exceed the upper bound W_{th}, the maximum number of writes to each address before the memory cell wears out.

Equations 1 to 14 form an ILP model that can be solved with an ILP solver.

1) An Illustrating Example: For illustration purposes, consider the toy example in Fig. 6. The SPM has size 3. The upper bound on total write count at each SPM memory $W_{th} = 8000$. The taskset consists of a single task τ_1 with no global variables. The task contains four procedures $\{P_{11}, P_{12}, P_{13}, P_{14}\}$ called by *main()*, with no path linking any two procedures in the call graph, hence the corresponding four stack frames $\{x_{11}, x_{12}, x_{13}, x_{14}\}$ can be placed arbitrarily on the stack without any overlay constraints. Each stack frame contains 2 variables with size 1, so the total set of variables is $\{x_{111}, x_{112}, \ldots, x_{141}, x_{142}\}$. Each variable's write count on the WCEP during each execution of the task x_{ij} is shown in Fig. 6a. Each variable's read count is not shown in the figure, assumed to be the same as its write count. Assume the task is a very simple program with simple control flow, and has WCEP the same as its average-case execution path. The latency of each SPM read operation is 1; latency of each SPM write operation is 10; Latency of each main memory read operation and write operation are both 50. The lifetime constraint of NVM given by the designer $L = 1 * 10^6$ (with arbitrary time unit). The task has period $T_1 = 1000$ (with the same time unit). WCET C_i consisting of CPU execution time plus memory access latency on the WCEP $C_i = Cexe_i + Cmem_i$ (with the same time unit). Assuming $Cexe_i = 0$ for simplicity.

a) Optimization Objective: Instantiating Equations 1 to 4:

$$\text{Minimize } (Cexe_1 + Cmem_1)/T_1 \tag{15}$$

where

$$\begin{aligned} Cmem_1 = \\ I(x_{11}) * (10*2 + 1*2 + 10*2 + 1*2) \\ +(1 - I(x_{11})) * (50*2 + 50*2 + 50*2 + 50*2) \\ +I(x_{21}) * (10*3 + 1*3 + 10*2 + 1*2) \\ +(1 - I(x_{21})) * (50*3 + 50*3 + 50*2 + 50*2) \\ +I(x_{31}) * (10*4 + 1*4 + 10*3 + 1*3) \\ +(1 - I(x_{31})) * (50*4 + 50*4 + 50*3 + 50*3) \\ +I(x_{41}) * (10*4 + 1*4 + 10*5 + 1*5) \\ +(1 - I(x_{41})) * (50*4 + 50*4 + 50*5 + 50*5) \end{aligned} \tag{16}$$

For example, Frame x_{11} contains two stack variables x_{111} and x_{112}, each read and written 2 times in each period. Hence during each execution of the task, the total memory access latency of variable x_{111} is 10*2+1*2 if allocated to SPM; $50*2+50*2$ if allocated to main memory; same for variable x_{112}. Hence the total memory access latency of Frame x_{11} is $I(x_{11})*(10*2+1*2+10*2+1*2)+(1-I(x_{11}))*(50*2+50*2+50*2+50*2)$. The optimization objective (Equations 15 and 16) has the effect of preferably allocating more variables to SPM to reduce $Cmem_i$. Without this objective, the ILP solver can legally find trivial solutions with infinite lifetime by allocating all variables to main memory.

Simplifying Equation 16, we have:

$$\begin{aligned} Cmem_1 = \\ I(x_{11}) * 44 + (1 - I(x_{11})) * 400 \\ +I(x_{21}) * 55 + (1 - I(x_{21})) * 500 \\ +I(x_{31}) * 77 + (1 - I(x_{31})) * 700 \\ +I(x_{41}) * 99 + (1 - I(x_{41})) * 900 \end{aligned} \tag{17}$$

b) SPM Size Constraints: Since the task contains no global variables, Equations 5 and 6 are omitted.

Instantiating Equation 6:

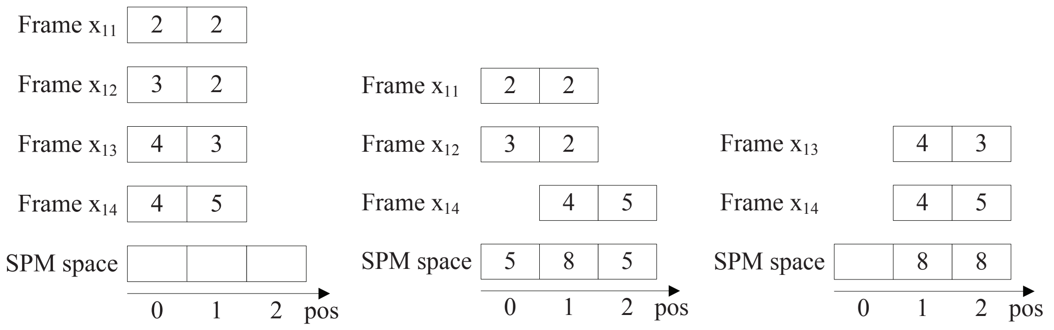
$$O(x_{11}) + 2 \leq 3, O(x_{12}) + 2 \leq 3, O(x_{13}) + 2 \leq 3, O(x_{14}) + 2 \leq 3 \tag{18}$$

c) No-Overlay Constraints: Since we assume there are no overlay constraints, Equation 7 is omitted.

d) Endurance Constraints: Since the task contains no global variables, Equation 8 is omitted.

Instantiating Equation 9:

$$\begin{array}{r} \forall i \in [1,1], \forall j \in [1,4], \forall k \in [1,2], \forall pos \in [0,3) \\ b_{pos}(x_{ijk}) = (I(x_{ij} = 1)\&\&pos \in [O(x_{ijk}), O(x_{ijk}) + 1)) \end{array} \tag{19}$$



(a) Before allocation (b) Solution with the ILP method (c) Solution with the heuristic algorithm

Fig. 6: A toy example of allocating four stack frames of a task to a SPM with size 3.

Instantiating Equation 14:

$$\forall pos \in [0,3)$$
$$\sum_{\forall i\in[1,1],j\in[1,4],k\in[1,2]} \frac{1*10^6}{1000} b_{pos}(x_{ijk}) N^w(x_{ijk}) \le 8000 \tag{20}$$

The upper bound on the write count *per period* is $8000/\frac{1*10^6}{1000} = 8$.

Equations 15 to 20 form an ILP model that can be solved by an ILP solver, with the resulting solution:

$$\begin{aligned} I(x_{11}) = 1, I(x_{12}) = 1, I(x_{13}) = 0, I(x_{14}) = 1 \\ O(x_{11}) = 0, O(x_{12}) = 0, O(x_{14}) = 1 \end{aligned} \tag{21}$$

Substituting into Equation 17, we have $Cmem_i = 44+55+700+99 = 898$, so the resulting CPU utilization (Equation 15) is (0+898)/1000=0.898. The solution is graphically shown in Fig. 6b.

C. Heuristic Algorithm

As the ILP method works by solving NP-complete problems optimally with exhaustive search, it may not be scalable to large systems. In this section, we present an efficient heuristic algorithm as an alternative to the ILP method.

$$\begin{aligned} Reduc(x_{ig}^G) = (Cmm(x_{ig}^G) - Cspm(x_{ig}^G))/(T_i * S_i) \\ Reduc(x_{ij}) = (Cmm(x_{ij}) - Cspm(x_{ij}))/(T_i * S_i)) \end{aligned} \tag{22}$$

$Cmm(x_{ig}^G)$ (or $Cspm(x_{ig}^G)$) denotes the total memory access latency of global data variable x_{ig}^G on the program WCEP if it is allocated to main memory (or SPM); $Cmm(x_{ij})$ (or $Cspm(x_{ij})$) denotes the total memory access latency of stack frame x_{ij} on the program WCEP if it is allocated to main memory (or SPM). $Reduc(x_{ig}^G)$ (or $Reduc(x_{ij})$) denotes the reduction of *per-byte* and *per-period* memory latency when the global variable x_{ig}^G (or the stack frame x_{ij} of procedure P_{ij}) is allocated to SPM. Larger values of $Reduc(x_{ig}^G)$ (or $Reduc(x_{ij})$) imply larger benefits of allocation to SPM in terms of reducing program WCET, hence we sort the global variables and stack frames based on decreasing order of these metrics, and choose the variable/stack frame with the largest value to allocate to SPM. As shown in Fig. 5, the SPM address space is divided into a global variables partition and a stack frame partition. The heuristic algorithm allocates global

Algorithm 1: Heuristic algorithm.

```
void GlobalVarAllocate($x_{ig}^G$){
Check if variable $x_{ig}^G$ satisfies the endurance constraint (Equation 8);
Check if the remaining empty space between the global variables partition and the stack variables partition is large enough to fit $x_{ig}^G$;
if Above two conditions are satisfied then
    Allocate $x_{ig}^G$ to SPM, and append it to the right side of global variables partition;
    Update partition boundary of the global variables partition;
else
    Allocate $x_{ig}^G$ to main memory;
}
void StackFrameAllocate($x_{ij}$){
Going from high to low address, find a contiguous address range with size $S(x_{ij})$ that satisfies the following two conditions:
1. Write count of each stack variable in procedure $P_{ij}$ does not exceed the remaining allowed write count of each memory address it occupies;
2. There does not exist another stack frame that is mutually-exclusive with $x_{ij}$ with overlapping address range;
if Such an address range is found then
    Allocate stack frame $x_{ij}$ to the address range, and update the write count on each SPM address covered by $x_{ij}$;
    Update partition boundary of the stack variables partition;
else
    Allocate stack frame $x_{ij}$ to main memory;
}
void SPMAllocate($U_G, U_F$){
Initialize remaining write count of each memory address in the SPM space to be $W_{th}$;
Compute $Reduc(x_{ig}^G)$ for each global variable $x_{ig}^G$, and $Reduc(x_{ij})$ for each procedure stack frame $x_{ij}$;
Sort global variables in $U_G$ in descending order of $Reduc(x_{ig}^G)$; sort stack frames in $U_F$ in descending order of $Reduc(x_{ij})$;
while $U_G \neq \emptyset$ or $U_F \neq \emptyset$ do
    Choose the data object (global variable or stack frame) in $U_G$ and $U_F$ with largest value of $Reduc()$;
    if the chosen object is a global variable $x_{ig}^G$ then
        GlobalVarAllocate($x_{ig}^G$); Remove $x_{ig}^G$ from $U_G$;
    else
        StackFrameAllocate($x_{ij}$); Remove $x_{ij}$ from $U_F$;
}
```

variables to SPM from left to right (low address to high address), and stack variables from right to left (high address to low address), hence the global variables partition grows from left to right, and the stack variables partition grows from right to left, with an empty space between the two partitions that continuously shrinks during running of the algorithm. The steps of the algorithm is shown as follows:

- Lines 1-9: function for allocating global variable x_{ig}^G. Since global variables cannot overlay with each other, a global variable can be allocated to SPM if the write count of the variable does not exceed the upper bound W_{th} (the endurance constraint (Equation 8), and it can fit in the empty space between the global variables partition and the stack variables partition.
- Lines 10-19: function for allocating stack frame x_{ij} . The search process is conducted from right (high addresses) to left (low addresses), since the left region of the SPM contains global variables. Check each address covered by x_{ij} to see if the address is covered by some other procedure stack frame which is mutually-exclusive to procedure P_{ij}, and make sure that the write count of this address does not exceed the upper bound. (Unlike the global variables, two stack frames that are not mutually exclusive can overlay each other in the SPM address space.)
- Lines 20-30: main function for allocating global variables and procedure stack frames.
- Lines 21-23: initialization of relevant parameters.
- Lines 24-29: Iteratively choose one data object from either U_G or U_F based on the metric Reduc(), and try to allocate it to SPM. The data object is removed from the set U_G or U_F regardless of whether the allocation fails or succeeds, since once a data object cannot be allocated to SPM, it can no longer be allocated to SPM during latter operations in the algorithm.

Fig. 7 illustrates the heuristic algorithm. Fig. 7a shows the situation before allocation of x_{ig}^G and x_{ij}. Since global variables cannot overlay with each other, a global variable x_{ig}^G can only be append it to the right side of global variables partition, as shown in. Since stack frames can overlay with each other if both endurance and No-Overlay constraints are satisfied, it is possible for a stack frame x_{ij} to be allocated either within the current stack variables partition (Fig. 7c), or in the empty space (Fig. 7d).

1) An Illustrating example: Consider the example in Fig. 6 again. Instantiating Equation 22:

$$\begin{aligned} Reduc(x_{11}) = ((50*2+50*2+50*2+50*2)- \\ (10*2+1*2+10*2+1*2))/(1000*2) = 0.178 \end{aligned} \tag{23}$$

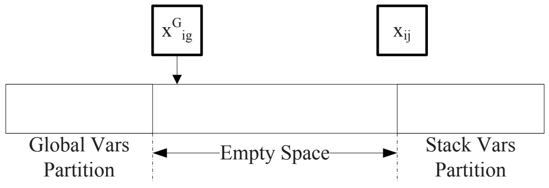
Similarly, we can obtain $Reduc(x_{21}) = 0.2225$, $Reduc(x_{31}) = 0.3115$, $Reduc(x_{41}) = 0.4005$. Since allocating stack frame x_{41} to SPM results in the largest reduction of per-byte and per-period memory latency, we choose x_{41} as the first stack frame to be allocated. Going from high address to low address, we can find the contiguous address range $[1, 2]$ with size $S(x_{41}) = 2$ that satisfies the following two conditions: 1. Write count of each stack variable in procedure P_{41} (4 and 5, respectively) does not exceed the remaining allowed (per-period) write count of each memory address it occupies (8); 2. There does not exist another stack frame that is mutually-exclusive with x_{41} with overlapping address range. So we allocate x_{41} to SPM in the address range [1,2]. Next, we consider stack frame x_{31} with the largest $Reduc(x_{31}) = 0.3115$ among the remaining unallocated stack frames, and find that the same contiguous address range $[1, 2]$ satisfies the two conditions, since the per-period write count of each stack variable in procedure P_{31} (4 and 3, respectively) does not exceed the remaining allowed (per-period) write count of each memory address it occupies (8-4=4, 8-5=3, respectively). So we allocate x_{31} to SPM in the address range [1,2]. Next, we consider the remaining unallocated variables x_{11} and x_{21}, but are unable to find any contiguous address range to place either one of them, since each address in the SPM address range [1,2] has reached its allowed maximum write count of 8. Eventually the heuristic algorithm terminates with the solution shown in Fig. 6c, with the following parameters:

$$\begin{aligned} I(x_{11}) = 0, I(x_{12}) = 0, I(x_{13}) = 1, I(x_{14}) = 1 \\ O(x_{13}) = 1, O(x_{14}) = 1 \end{aligned} \tag{24}$$

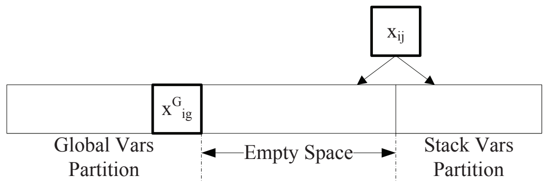
Substituting into Equation 17, we have $Cmem_1 = 400 + 500 + 77 + 99 = 1076$, so the resulting CPU utilization (Equation 15) is (0+1076)/1000=1.076. (The taskset is not schedulable in this case, although the endurance constraint is satisfied.) The solution is graphically shown in Fig. 6c, which is an inferior solution compared to the solution obtained by ILP (Fig. 6b) in terms of the optimization objective of CPU utilization.

IV. Performance Evaluation

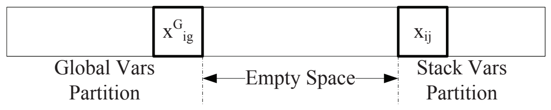
Table II shows target platform parameters (The memory system parameters are obtained from [10]). We consider resource-constrained embedded systems based on low-end 8 or 16-bit microcontrollers, with limited on-chip memory storage that can be used as cache or SPM. Since STT-RAM can be made much denser than SRAM, we compare target systems with 2 KB STT-RAM based SPM versus 1 KB SRAM-based SPM. We assume the upper bound on write count at each STT-RAM memory address is 10^{12} [10]. We take some benchmark programs from the Malardalen WCET benchmarks [21]. Since we focus on small real-time embedded systems with hard real-time constraints, we use the Malardalen WCET benchmarks, which are typically small programs with complex control flow to stress-test WCET analysis tools, in contrast to standard architectural benchmarks such as SPEC, which are computation-intensive programs to stress-test high-performance computers. We selected six programs from the Malardalen WCET benchmarks, including $minver$, $adpcm$, $ludcmp$, $ndes$, fir and edn. The selection criteria are that each program must contain a significant number of data variables. The programs are compiled with the SimpleScalar toolchain with ”-O2” optimization option. We use the Chronos WCET tool [22] to obtain each program's WCET, along with the corresponding WCEP, and obtain the number of read/write memory accesses of each variable x from the memory access trace on the WCEP, then obtain $Cmm(x)$ and $Cspm(x)$, total memory access latency of data variable x on the program WCEP if it is allocated to main memory or SPM. We also run each benchmark program multiple times with random inputs



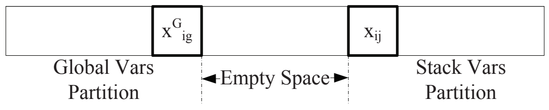
(a) Before allocation of x^G_{ig} and x_{ij}



(b) x^G_{ig} is allocated to SPM, reducing the empty space by its size



(c) x_{ij} is allocated to SPM, overlayed with existing stack variables, hence does not affect the empty space size



(d) x_{ij} is allocated to SPM, not overlayed with existing stack variables, reducing the empty space by its size

Fig. 7: Illustration of the heuristic algorithm, before and after allocation of global variable x^G_{ig} and stack frame x_{ij}.

to collect memory access traces, and obtain *average* number of *write* memory accesses $N^w(x)$ of each variable x from the traces, used in the endurance constraints. We construct a taskset with six tasks, each one corresponding to one of the six programs. Table III shows taskset parameters. Each task is assigned a nominal priority $Prio_i$ and a preemption threshold PT_i (larger number denotes higher priority). We assign the maximum value of 6 to each task's preemption threshold, making them all mutually non-preemptive, hence the stack frames of all tasks can share the same SPM space. The period of each task is set to be 10 times its WCET with all its variables allocated to main memory.

The experiment platform is: Intel Core i7 CPU with CPU speed 2.67GHz, 4GB main memory running 64-bit Ubuntu Linux Release 11.10. The IBM ILOG CPLEX Optimizer V12.6.1 is used as the ILP solver.

TABLE II: Target platform parameters

Component	Description
CPU	Frequency: 1.0 GHz
NVM	Read latency: 1 cycle, write latency: 10 cycles. Size: 2 KB
SRAM	Read latency: 1 cycle, write latency: 1 cycle. Size: 1 KB
Main memory	Access latency: 50 cycles

TABLE III: Taskset parameters. τ_i denotes the task; $Prio_i$ denotes its priority; PT_i denotes its preemption threshold; T_i denotes its period.

τ_i	$Prio_i$	PT_i	T_i	Global Vars (B)	Stack Vars (B)
minver	6	6	1	460	88
adpcm	5	6	3	2000	100
ludcmp	4	6	6	1144	92
ndes	3	6	11	1516	172
fir	2	6	13	944	416
edn	1	6	25	1224	124

We consider six different lifetime constraints: 1, 2, 4, 8, 16 and 32 years. We apply the ILP method, and the heuristic algorithm for procedure-level variable allocation, denoted *ILP* for ILP method, and *Heuristic* for the heuristic algorithm. Fig. 8 plots total system CPU utilization vs. system lifetime constraint. We can see that CPU utilization for 2KB NVM-based SPM is much less than that for 1KB SRAM-based SPM,

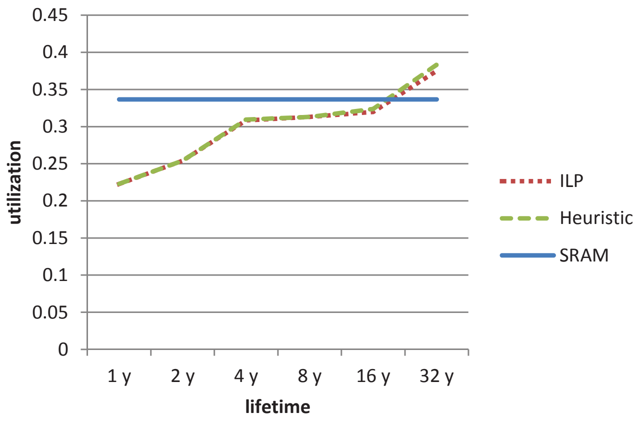
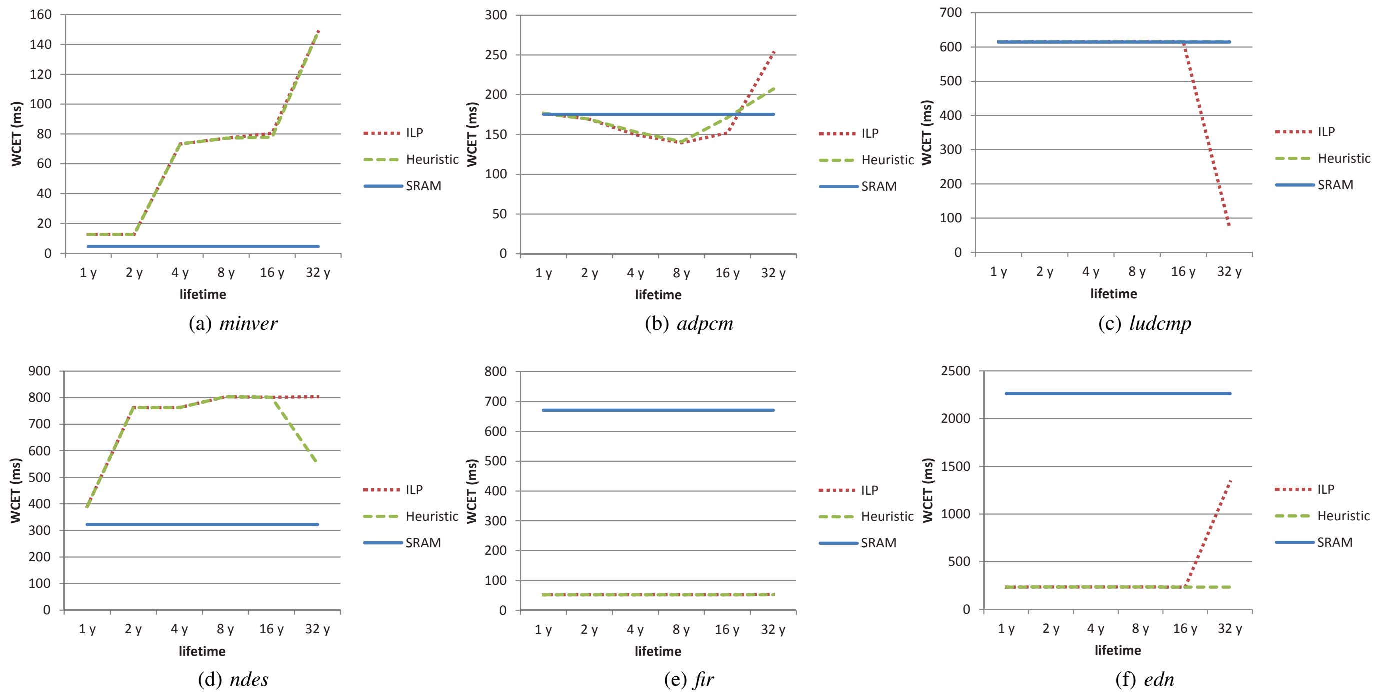


Fig. 8: Total system CPU utilization vs. system lifetime constraint.

especially when the system lifetime is short. In general, a program's WCET increases when lifetime becomes longer, since more variables need to be moved from SPM to main memory to reduce the write pressure on NVM-based SPM. As shown in Fig. 8, the total CPU utilization increases with each program's WCET, ranging from 0.22 for 1-year lifetime, to 0.38 for 32-year lifetime. The CPU utilization is not very large, e.g., it increases by 14% when the lifetime increases from 1 year to 2 years; by 38% when the lifetime increases from 1 year to 4 years. We can also see that the heuristic algorithm can obtain results very close to the optimal results by the ILP method. We applied the data allocation algorithm DART in [18], [19] at procedure-level allocation granularity without considering NVM endurance issues, and found that NVM lifetime is 369 days, with CPU utilization of 22.3%.

Fig. 9 plots each individual program's WCET vs. system lifetime constraint. Since the optimization objective is to minimize total CPU utilization, not WCET of each individual program, it is reasonable to see each program's WCET value increase or decrease when system lifetime constraint is changed. In addition, even though ILP and heuristic obtain similar results for the total CPU utilization, they may produce different WCET values for different individual programs, as



(a) *minver* (b) *adpcm* (c) *ludcmp* (d) *ndes* (e) *fir* (f) *edn*

Fig. 9: Each program's WCET vs. system lifetime constraint.

shown in Fig. 9.

For some programs, e.g., *fir* (Fig. 9(e)) and *edn* ((Fig. 9(f)), most variables are read-intensive, hence many variables can be allocated to NVM-based SPM with size 2KB, causing their WCET values to be much smaller than WCET values for the SRAM-based SPM with size 1KB. They stay small even for long system lifetime constraints, e.g., WCET of fir stays the same for all system life constraints, while WCET of *edn* shows an increase only when lifetime constraint exceeds 16 years.

WCET of *adpcm* (Fig. 9 (b)) *decreases* when the lifetime constraint is increased from 1 year to 8 years, but *increases* after the lifetime constraint is increased from 8 years to 32 years. It is a result of tradeoffs when making data allocation decisions for *adpcm* and *minver*. As shown in Table III, the task period of *minver* (1ms) is much smaller than that of *adpcm* (3ms). This causes the data variables of *minver* to be preferentially allocated to SPM for short lifetime constraints, since their allocation has larger effect in reducing CPU utilization than allocation of variables of *adpcm*. As system lifetime constraint is increased from 1 to 8 years, some data variables of *minver* are forced to move from SPM to main memory to reduce the number of writes on SPM, causing a large increase in its WCET (Fig. 9 (a)); at the same time, some variables of *adpcm* can be moved from main memory to SPM, since the task running *adpcm* has larger period than the task running *minver*, causing a decrease in its WCET (Fig. 9 (b)). But as system lifetime constraint is increased further beyond 8 years, variables of *adpcm* are also forced to move from SPM to main memory, causing an increase in its WCET. For the case discussed above, ILP and heuristic algorithm yield consistent results. There are also some cases when ILP and heuristic algorithm produce inconsistent results, e.g., WCET of *ludcmp* (Fig. 9 (c)) shows a large decrease when system lifetime is increased from 16 to 32 years with ILP, due to moving some variables from main memory to SPM, but not with heuristic algorithm.

Table IV shows size (in Bytes) of each program's variables (global and stack) allocated to SPM with different lifetime constraints, obtained with the ILP method. Each table cell is formatted as *"size of global variables allocated to SPM, size of stack variables allocated to SPM"*. The sum of the two fields may exceed the total SPM size of 2KB. With system lifetime constraint of 1 year, the global variables in SPM have total size of 1928 Bytes, hence the partition for stack variables has size 2048-1928=120 Bytes. The total size of stack variables allocated to SPM is 552 Bytes, which are overlayed over the partition of size 120 Bytes during program execution. As system lifetime constraint is increased to 32 years, the total size of stack variables allocated to SPM is decreased to 184 Bytes, with the partition size for stack variables equal to 2048-1960=98 Bytes. In general, total size of stack variables allocated to SPM decreases when system lifetime constraint is increased, since stack variables are temporary, and likely to be write-intensive; since runtime overlays cause increased writes to each address in the partition of stack variables on SPM, the frequency of overlays will also be decreased (roughly measured as total stack variable size divided by partition size of stack variables).

Fig. 10 shows running times of the ILP solver (CPLEX) with different lifetime constraints for our case study. The running time reaches its peak value of 383 seconds when lifetime constraint is set as 16 years, and shorter or longer lifetime constraints both causes running time to decrease. When lifetime constraint is shorter, the *per-period* upper

TABLE IV: Size (in Bytes) of all variables allocated to NVM with different lifetime constraints.

	1 year	2 years	4 years	8 years	16 years	32 years
minver	452,88	452,88	380,88	360,88	360,8	0,0
adpcm	196,100	208,100	280,100	316,76	276,76	12,4
ludcmp	0,92	0,92	0,92	0,92	0,92	968,88
ndes	132,136	132,104	132,104	132,88	132,124	132,88
fir	544,12	544,12	544,12	544,12	544,12	544,0
edn	608,124	608,92	608,92	608,92	608,124	304,4
total	1928,552	1952,488	1944,488	1960,444	1920,436	1960,184

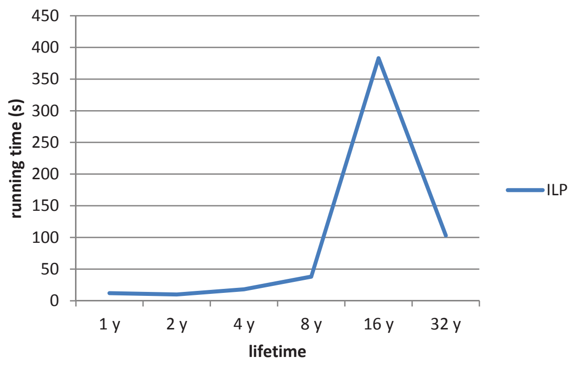


Fig. 10: Running time of the ILP solver.

bound on the total write count at each SPM address is higher, according to Equations 8 and 14, causing the ILP problem to be *under-constrained*, and it is easy to find solutions where most variables are allocated to SPM. On the other hand, when lifetime constraint is longer, the ILP problem is *over-constrained*, and it is easy to find solutions where most variables are allocated to main memory. The ILP problem becomes the most difficult to solve when the problem is neither under-constrained nor over-constrained, as is the case with lifetime constraint of 16 years. The heuristic algorithm consistently runs very fast (less than 1 second), hence its running times are not plotted.

V. Conclusions

In this paper, we consider NVM-based SPM for real-time embedded systems, and present algorithms to allocate the data variables to SPM which can distribute the number of writes evenly in the SPM address space, in order to achieve wear-leveling and prolong the lifetime of NVM.

References

[1] Y. Xie, Ed., *Emerging Memory Technologies: Design, Architecture, and Applications*. Springer, 2014.

[2] P. R. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems," *ACM Trans. Design Autom. Electr. Syst.*, vol. 5, no. 3, pp. 682–704, 2000.

[3] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 1, no. 1, pp. 6–26, 2002.

[4] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Wcet centric data allocation to scratchpad memory," in *Proc. RTSS*. IEEE, 2005, pp. 233–242.

[5] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Trans. Embedded Comput. Syst.*, vol. 5, no. 2, pp. 472–511, 2006.

[6] Q. Wan, H. Wu, and J. Xue, "Wcet-aware data selection and allocation for scratchpad memory," in *Proc. LCTES*, 2012, pp. 41–50.

[7] Z. Wang, Z. Gu, and Z. Shao, "Wcet-aware energy-efficient data allocation on scratchpad memory for real-time embedded systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2015.

[8] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E.-M. Sha, "Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[9] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Data allocation optimization for hybrid scratch pad memory with sram and nonvolatile memory," *IEEE Trans. VLSI Syst.*, vol. 21, no. 6, pp. 1094–1102, 2013.

[10] A. M. H. Monazzah, H. Farbeh, S. G. Miremadi, M. Fazeli, and H. Asadi, "Ftspm: A fault-tolerant scratchpad memory," in *DSN*. IEEE, 2013, pp. 1–10.

[11] Z. Wang, Z. Gu, and Z. Shao, "Optimizated allocation of data variables to pcm/dram-based hybrid main memory for real-time embedded systems," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 61–64, 2014.

[12] M. Qiu and E. H.-M. Sha, "Cost minimization while satisfying hard/soft timing constraints for heterogeneous embedded systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 2, p. 25, 2009.

[13] M. Qiu, Z. Chen, J. Niu, G. Quan, X. Qin, and L. Yang, "Data allocation for hybrid memory with genetic algorithm," *IEEE Transactions on Emerging Topics in Computing*, 2015.

[14] M. Qiu, M. Zhong, J. Li, K. Gai, and Z. Zong, "Phase-change memory optimization for green cloud with genetic algorithm," *IEEE Transactions on Computers*, 2015.

[15] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H.-M. Sha, "Software enabled wear-leveling for hybrid pcm main memory on embedded systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. IEEE, 2013, pp. 599–602.

[16] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 50–61.

[17] Y. Li, Y. Zhang, H. Li, Y. Chen, and A. K. Jones, "C1c: A configurable, compiler-guided stt-ram l1 cache," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, p. 52, 2013.

[18] R. Ghattas, G. S. Parsons, and A. G. Dean, "Optimal unified data allocation and task scheduling for real-time multi-tasking systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2007, pp. 168–182.

[19] S. Kang and A. G. Dean, "Darts: Techniques and tools for predictably fast memory using integrated data allocation and real-time task scheduling," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, M. Caccamo, Ed. IEEE Computer Society, 2010, pp. 333–342.

[20] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*. IEEE, 1999, pp. 328–335.

[21] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future." *WCET*, vol. 15, pp. 136–146, 2010.

[22] X. Li, L. Yun, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 56–67, 2007.

Zhu Wang received his Bachelor's degree in Software Engineering from Wuhan University in 2009. He is currently a Ph.D. student in the College of Computer Science, Zhejiang University. His research area is real-time embedded systems.

Zonghua Gu received the Ph.D. degree in Computer Science and Engineering from the University of Michigan at Ann Arbor in 2004. He is currently an associate professor in the College of Computer Science, Zhejiang University. His research area is real-time and embedded systems.

Min Yao received his Ph. D. degree in Biomedical Engineering and Instrument from Zhejiang University, China, in 1995. He is currently a professor at the college of Computer Science and Technology, Zhejiang University. His research interests include computational intelligence, pattern recognition, knowledge discovery and knowledge service.

Zili Shao received the B.E. degree in electronic mechanics from the University of Electronic Science and Technology of China, Sichuan, China, in 1995, and the M.S. and the Ph.D. degrees from the Department of Computer Science, University of Texas at Dallas, Dallas, TX, USA, in 2003 and 2005, respectively.

He has been an Associate Professor with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, since 2010. His current research interests include embedded software and systems, real-time systems, and related industrial applications. He is an associate Editor for IEEE Transactions on Computers, ACM Transactions on Cyber-Physical Systems, IEEE Embedded Systems Letters, and Journal of Systems Architecture: Embedded Software Design (subject area: Memory Systems). He serves/served the technical program committees of many top conferences in the real-time embedded system field (such as DAC, RTSS, ICCAD, IPDPS, RTAS, DATE, CODES+ISSS, EMSOFT, ISLPED, ASP-DAC, ICCD and LCTES).