Data Scratchpad Prefetching for Real-time Systems

Muhammad R. Soliman and Rodolfo Pellizzoni
University of Waterloo, Canada. {mrefaat, rpellizz}@uwaterloo.ca

***Abstract*— In recent years, the real-time community has produced a variety of approaches targeted at managing on-chip memory (scratchpads and caches) in a predictable way. However, to obtain safe Worst-Case Execution Time (WCET) bounds, such techniques generally assume that the processor is stalled while waiting to reload the content of on-chip memory; hence, they are less effective at hiding main memory latency compared to speculation-based techniques, such as hardware prefetching, that are largely used in general-purpose systems. In this work, we introduce a novel compiler-directed prefetching scheme for scratchpad memory that effectively hides the latency of main memory accesses by overlapping data transfers with the program execution. We implement and test an automated program compilation and optimization flow within the LLVM framework, and we show how to obtain improved WCET bounds through static analysis.**

I. Intro

The performance of computer programs can be significantly affected by main memory latency, which has largely remained similar in recent years. As a consequence, cache prefetching has been extensively researched in the architecture community [1]. Prefetching techniques incorporate hardware and/or software to hide cache miss latency by attempting to load cache lines from main memory before they are accessed by the program. The essence of these techniques is speculation of the data locality and the cache behavior, which makes them unsuitable to provide Worst-Case Execution Time (WCET) guarantees for real-time programs.

In the context of real-time systems, in recent times there has been significant attention to the management of on-chip memory. In particular, a large number of allocation schemes for scratchpad memories have been proposed in the literature; compared to caches, ScratchPad Memory (SPM) requires an explicit management of transfers from/to main memory. We note that cache memories can also be managed in a predictable manner similar to SPM, for example employing cache locking [2]. These techniques allow the derivation of tighter WCET bounds by statically determining which memory accesses target on-chip and which main memory. However, they do not solve the fundamental memory latency problem, because they generally assume that the core is stalled while the content of on-chip memory is reloaded.

To address such issue, in this paper we present a novel compiler-directed prefetching scheme that optimizes the allocation of program data in on-chip memory with the objective to minimize the WCET. Our method relies on a Direct Memory Access (DMA) controller to move data between on-chip and main memory. Compared to related work, we do not stall the program while transferring data; instead, we rely on static program analysis to determine when data is used in the program, and we prefetch it into on-chip memory ahead of its use so that the time required for the DMA transfer can be *overlapped* with the program execution. As we show in our evaluation, for certain benchmarks our solution allows to drastically reduce the stall time due to memory latency. More in details, we provide the following contributions:

- We describe an allocation mechanism for SPM that manages DMA transfers with minimum added overhead to the program. To statically determine which accesses target the SPM, we introduce a program representation and allocation constraints based on refined code regions.
- We develop an allocation algorithm for data in scratchpad memory that takes into account the overlap between DMA transfers and program execution.
- We show how to model the proposed mechanism in the context of static WCET analysis using a standard data-flow approach for processor analysis.
- We fully implement all required code analysis, optimization and transformation steps within the LLVM compiler framework [3], and test it on a collection of benchmarks. Outside of loop bound annotations, our prototype is able to automatically compile and optimize the program without any programmer intervention.

The rest of the paper is organized as follows. We recap related work in Section II. We then introduce a motivating example in Section III. We detail the region-based program representation in Section IV, and our proposed allocation mechanism in Section V. Section VI discusses the allocation algorithm, and Section VII introduces the WCET abstraction for our prefetch mechanism. Finally, we present the compiler implementation in Section VIII and experimental results in Section IX, and provide concluding remarks in Section X.

II. Related Work

ScratchPad Memory (SPM) management has been widely explored in the literature, both for code and data allocation. We focus on data SPM as it drew more attention in the literature due to the challenges connected to data usage analysis and optimization. Many approaches target improving the average case performance [4], [5], [6], [7], [8], [9]. Other mechanisms optimize the allocation for the WCET in real-time systems [10], [11], [12], [13]. In general, management techniques are divided between static or dynamic. Static methods partition the data memory between the SPM and the main memory and assign fixed allocation of the SPM at compile-time [6], [10]. On the other hand, dynamic methods adapt to the changing working data set of the program by moving objects between the SPM and main memory during run-time [4], [5], [13], [9], [11], [7]. Since our proposed scheme allows us to more efficiently hide the cost of data transfers, we focus on a dynamic approach.

The closest related work in the scope of dynamic methods for data SPM are [7], [8], [14], which utilize prefetching through DMA. In [7], the authors use a SPM data pipelining (SPDP) technique that utilizes Direct Memory Access (DMA) to achieve data parallelization for multiple iterations of a loop based on iteration access patterns of arrays. The work in [8] proposes a general prefetching scheme for on-chip memory. It exploits the usage of DMA priorities and pipelining to prefetch arrays with high reuse to minimize

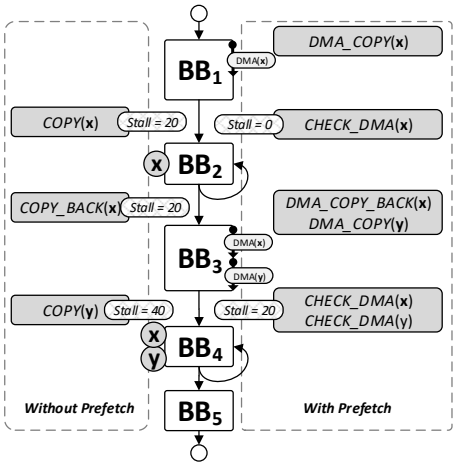


Fig. 1: Motivating Example

the energy and maximize average performance. In [14], the authors add a DMA engine to the processor to control the DMA transfers using a job queue, similarly to the mechanism proposed in our work. They also provide high level functions to manage the DMA. However, no optimized allocation scheme is discussed. Furthermore, all three discussed works target the average case rather than the worst case.

In the context of real-time systems, the closest line of work is the PRedictable Execution Model (PREM) [15], [16], [17]. Under PREM, the data and code of a task is prefetched into on-chip memory before execution, preferably using DMA. A variety of co-scheduling schemes (see for example [18], [19]) have been proposed to avoid stalling the processor by scheduling the DMA operations for one task with the execution of another task on the same core. However, we argue that such approaches suffer from three main limitations, that we seek to lift in this work. 1) Statically loading all data and code before the beginning of the program severely limits the flexibility and precision of the allocation. 2) DMA transfers cannot be overlapped with the execution of the same task, only other tasks. This makes the proposed approaches less suitable for many-core systems, where it might be preferable to execute a single task/thread on each core. 3) With the exception of [20], the proposed approaches assume manual code modification, which we find unrealistic in practice. An automated compiler toolchain is described in [20], but since it relies on profiling, it cannot guarantee WCET bounds.

III. Motivating Example

In this section, we present an example that shows the benefit of data prefetching in scratchpad-based systems. Given a set of data *objects* used by a program, the general scratchpad allocation problem is to determine which subset of objects should be allocated in SPM to minimize the WCET of the program. Since the latency of accessing an object in the SPM is less than in main memory, we can compute the *benefit* in terms of WCET reduction for each object allocated in the SPM. We model the program's execution with a Control Flow Graph (CFG) where nodes represent basic blocks, i.e., straight-line pieces of code. In particular, Figure 1 shows the CFG of a program where object x is read/written in basic blocks BB_2 and BB_4 and object y is read in BB_4. Note that BB_2 and BB_4 are loops, since they include back-edges (i.e., the program execution can jump back to the beginning of the block); hence, x and y can be accessed many times. Assume that the SPM can only fit x or y. A static SPM allocation approach will choose to allocate either x or y for the whole program execution. A dynamic SPM allocation approach will try to maximize the benefit by possibly evicting one of the two objects to fit the other during the program execution.

Let the benefit of accessing x from the SPM instead of the main memory be 100 cycles for BB_2 and 10 cycles for BB_4. Similarly, the benefit for accessing y from the SPM in BB_4 is 70 cycles. Let the cost to transfer x from main memory to the SPM or vice-versa be 20 cycles, and the cost for y 40 cycles. Then, for static allocation, the total benefit of allocating x is $100+10=110$ cycles and the cost is 2*20 cycles (fetch x from memory to SPM at the beginning of the program and write it back from SPM to main memory at the end). Similarly, the benefit for allocating y is 70 cycles and the cost is 20 cycles (fetch only as y is not modified, so there is not need to write it back to main memory). The optimal allocation would choose x as it has a net benefit of 70 cycles versus 50 cycles for y.

In previous approaches that adopt dynamic allocation, the program execution has to be interrupted to transfer objects either using a software loop or a DMA unit. We represent this case in the *without prefetch* box in Figure 1. In the example, x is fetched before BB_2 and written back after BB_2 to empty the SPM for y. Then, y is fetched before BB_4. Since x is allocated in the SPM for BB_2 and y is allocated for BB_4, this results in a total benefit of $100+70=170$. The program will stall before BB_2 to fetch x, after BB_2 to write-back x, and before BB_4 to fetch y. The total cost is $20+20+40=80$ cycles as the execution has to be stalled for each fetch/write-back transfer. The net benefit is $170-80=90$ cycles, which is 20 cycles better than the static allocation.

However, if memory transfers can be parallelized with the execution time of the program, we next show that we can exploit the SPM more efficiently. We illustrate the prefetching sequence in the *with prefetch* box in Figure 1. Let us assume that the amount of execution time that can be overlapped with DMA transfers is 30 and 40 cycles for BB_1 and BB_3, respectively. We start prefetching x before BB_1 by configuring the DMA to copy x from main memory to SPM. Then, we poll the DMA before BB_2 where x is first used to ensure that the transfer has finished. Since transferring x requires less cycles than the maximum overlap for BB_1 (20 versus 30), the prefetch operation for x finishes in parallel with the execution of BB_1; hence, there is no need to stall the program before x can be accessed from the SPM in BB_2. Before BB_3, we first write-back x so that we have enough space in the SPM to then prefetch y. We propose to schedule both transfers back-to-back, *e.g.* using a scatter-gather DMA, in parallel with the execution of BB_3. Since the amount of overlap for BB_3 is 40, the write-back for x completes after 20 cycles, leaving 20 additional cycles of overlap for the prefetch of y. Hence, by the time BB_4 is reached, the CPU stalls for $40-20=20$ cycles to complete prefetching y before using it in BB_4. For the described prefetching approach, the benefit is the same as the dynamic allocation. However, the cost is lower as the CPU only stalls for 20 cycles. The net benefit is $170-20=150$ cycles, compared to 90 cycles without prefetching.

IV. Region-Based Program Representation

The motivating example shows that the cost of copying objects between main memory and SPM can be reduced by overlapping DMA transfers with program execution. However, to achieve a positive benefit, we also need to predict whether any given memory access targets the SPM rather than main memory. In general, programs contain branches

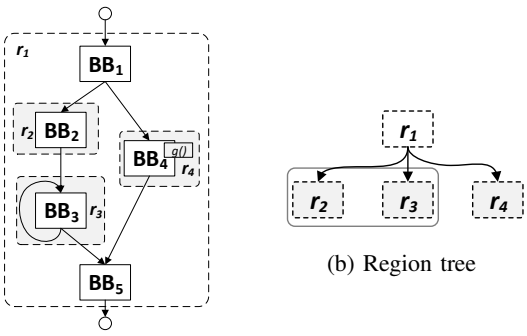
and function calls, making such determination possibly dependent on the execution path. To produce tight WCET bounds, a fundamental goal of our approach is to *statically determine which memory accesses are in the SPM regardless of the flow through the program*. To achieve this objective, in this section we consider a program representation based on code *regions* [21] and we add constraints on how objects can be allocated in the SPM based on regions.

We consider a program composed of multiple functions. Let $G_f = (N, E)$ be the CFG for function f, where N is the set of nodes representing basic blocks and E is the set of edges. A *Single Entry Single Exit (SESE) region* is a sub-graph of the CFG that is connected to the remaining nodes of the CFG with only two edges, an entry edge and an exit edge. A region is called *canonical* if there is no set of regions that can be combined to construct it. Any two canonical regions are either disjoint or completely nested. The canonical regions of a program can be organized in a *region tree* such that the *parent* of a region is the closest containing region, and children of a region are all the regions immediately contained within it. Two regions are *sequentially composed* if the exit of one region is the entry of the following region. Note that a basic block with multiple entry/exit edges does not construct a region by itself.

Figure 2a shows an example CFG and its canonical regions. The corresponding region tree is shown in Figure 2b. In this example, region r_1 is the parent of regions r_2, r_3 and r_4. Regions r_2 and r_3 are sequentially composed; this is represented by a solid-line box in the figure.

In the rest of the paper, we use the term *allocation* to refer to the act of reserving a space for an object in the SPM at a given address during the execution of the program code. In our solution, we restrict the allocation of objects on a per-region basis: space for an object is reserved upon entering a region, and the object is then evicted from the SPM upon exiting the same region. This guarantees that the object is available in the SPM independently of which path the program takes through the region; as an example, if we allocate object x in r_1, then we statically know that any reference to x in BB_5 will access the SPM independently of whether the program flows through $BB_2 - BB_3$ or BB_4, or of how many iterations of the loop in BB_3 are taken.

Unfortunately, the proposed region-based allocation has two limitations: 1) we cannot allocate an object in BB_1 only, because BB_1 is not a region; 2) in the example, BB_4 performs a call to another function $g()$. Since the entirety of BB_4 is a region, we cannot decide to allocate an object only for the call to $g()$, or only for the rest of the code of BB_4. To address these limitations, we propose to construct a refined region tree that allows a finer granularity of allocation.



(a) Program CFG

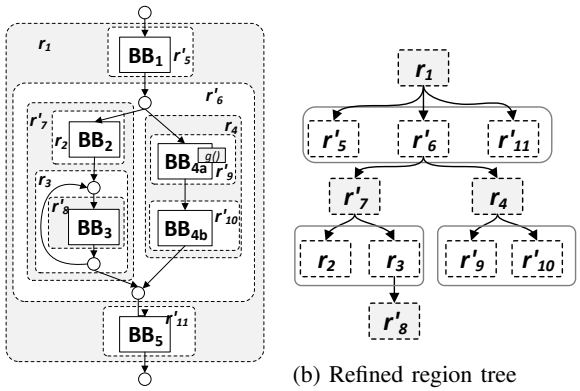
(b) Region tree

Fig. 2: Program CFG G_f and region tree

To obtain the refined regions, we first construct a modified graph $\bar{G}_f = (\bar{N}, \bar{E})$ from G_f, where $\bar{N}$ is the set of basic block nodes, call nodes and merge/split nodes and $\bar{E}$ is the set of edges such that:

- Each call to a function in G_f is split into a separate *call node*.
- A *merge/split node* is inserted before/after a basic block / call node with multiple entry/exit edges.

Note that after the transformation, every node in $\bar{G}$ that is not a merge/split node has a single entry and a single exit; hence, it is a region. We denote a region that consists of a sequence of sequentially composed regions as a *sequential region*. A sequential region is not canonical as it is constructed by combining other regions. Finally, we construct the refined region tree by considering both canonical regions and maximal sequential regions, i.e., any sequential region that encompasses a maximal sequence of sequentially composed regions. It is proved in [22] that adding maximal sequential regions to the tree still results in a unique region tree.



(a) Refined program CFG

(b) Refined region tree

Fig. 3: Refined program CFG $\bar{G}_f$ and region tree

Figure 3 shows the refined CFG and region tree for the example in Figure 2. We added merge points before BB_3 and BB_5, and split points after BB_1 and BB_3. Assuming that function $g()$ is called at the beginning of BB_4, we split BB_4 to a call node BB_{4a} that contains the function call and a basic block BB_{4b} for the rest of the instructions in BB_4. In the refined region tree in Figure 3b, regions r_1, r'_7 and r_4 are sequential regions. The regions r_1 to r_4 are the same as in the original region tree, while regions r'_5 to r'_{11} are added as a result of the refinement process. We refer to r'_3 as a *call region* as it contains the call node BB_{4a}. Finally, we use the term *trivial region* to denote any leaf of the refined region tree ($r'_5, r'_{11}, r_2, r'_8, r'_9$ and r'_{10} in the example); note that by definition, each trivial region must comprise either a single basic block or a single call node, i.e., trivial regions represent code segments in the program. Since allocations are based on regions, for simplicity we will omit individual nodes when representing CFGs and instead draw regions.

V. Allocation Mechanism

We now present our proposed allocation mechanism in details. In the rest of the paper, we assume the following:

- We focus solely on the allocation of data SPM, as it is generally more challenging. We assume a separate instruction SPM that is large enough to fit the code.
- The allocation is object-based, meaning that we do not allow allocation of parts of an object. Transformations

like tiling and pipelining could further improve the allocation, especially for small sizes of SPM. We keep this possible expansion to future work.
- We assume that the target program does not use recursion or function pointers and that local objects have fixed or bounded sizes. We argue that these assumptions conform with standard conventions for real-time applications.
- We assume that all loops in the program are bounded. The bounds can be derived using compiler analysis, annotations or profiling.
- We use pointer analysis to determine the references of the load/store instructions. A *points-to set* is composed for each pointer reference. The size of the points-to set depends on the precision of the pointer analysis. An allocation-site abstraction is used for dynamically allocated objects to represent objects, *i.e.*, to consider a single abstract object to stand in for each run-time object allocated by the same instruction [23]. To be able to allocate a dynamically allocated object, an upper bound on the allocation size should be provided at compile-time.
- For simplicity, we consider a system comprising a single core running one program. However, the proposed method could be extended to a multicore system supporting a predictable arbitration for main memory as long as each core is provided with private or partitioned SPM.

As discussed in the motivating example, to efficiently manage the dynamic allocation of multiple objects we require a DMA unit capable of queuing multiple operations. In general, many commercial DMA controllers with scatter-gather functionality support such requirement, albeit the complexity of managing the DMA controller in software and checking whether individual operations have been completed could increase with the number of transfers. As a proof of concept, we based our implementation on a dedicated unit, which we call the *SPM controller*; we reserve implementation on a COTS platform as future work [1].

Our proposed mechanism works by inserting *allocation commands* in the code of the program, which are then executed by the SPM controller. The process of allocating an object starts with reserving space in the SPM and prefetching the object from main memory if necessary (**ALLOC** command). Then, once the prefetch operation is complete, the SPM address is read and passed to the memory references that access the object (**GETADDR** command). Finally, the object is evicted from the SPM and written back to main memory if necessary (**DEALLOC** command). As discussed in Section IV, we restrict object allocation based on regions; hence, the **ALLOC** command is always inserted at the beginning of a region, and the corresponding **DEALLOC** command at the end of the same region. In the rest of the section, we first detail the operation of the SPM controller, followed by the semantic of the allocation commands. Finally, we provide a comprehensive allocation example.

A. SPM controller

Figure 4 shows the proposed SPM controller and its connections to an SPM-based system. There is a separate instruction SPM (I-SPM) that is assumed to fit the code of the program. The data SPM (D-SPM) is managed by the SPM controller. Since the processor must be able to access the SPM directly, the SPM is assigned an address range distinct from main memory. The SPM controller is also a memory mapped unit, since the CPU sends allocation commands to the SPM controller by reading/writing to its address range. The system incorporates a DMA unit for memory transfers. The D-SPM is assumed to be dual-ports, which means that access to the SPM by the CPU and transferring data between SPM and main memory using DMA can occur simultaneously. The proposed allocation method and WCET analysis can be applied for single-port SPM, but this will offer less opportunity to overlap the memory transfers. The DMA is connected to a shared bus with the main memory. This bus can be used by either the CPU or the DMA. To efficiently support the parallization of memory transfers with the execution time, the DMA is designed to work in transparent mode: it transfers an object only when the CPU is not using the main memory. Whenever the CPU requests the memory bus, the DMA yields to the request and stalls any ongoing transfer until the memory bus is released.

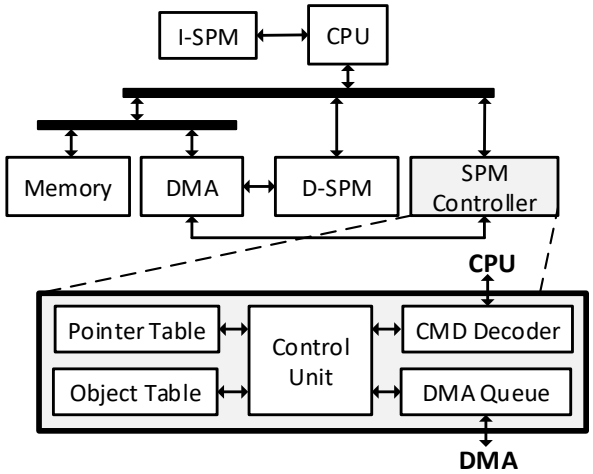


Fig. 4: SPM-based System

The SPM controller consists of *command decoder*, *object table*, *pointer table*, *DMA queue* and *control unit* as shown in Figure 4. As discussed, allocation commands are encoded as load/store instructions to the SPM controller. So, the command decoder reads the address and the data of the memory operation and decodes them into one of the allocation commands; the control unit then executes the command using the object table, the pointer table and the DMA queue.

The object table tracks the status of allocated objects in the SPM. The entry of the object table contains the main memory address (*MM_ADDR*), the size of the object (*SIZE*), the SPM address (*SPM_ADDR*) and allocation flags. The flags reflects the status of the object:

A	(A)llocated in the SPM
PF_OP	(P)re(F)etching (OP)eration has been scheduled
WB_OP	(W)rite-(B)ack (OP)eration has been scheduled
WB	(W)rite-(B)ack when de-allocated if used
U	(U)sed in the SPM
USERS	number of current users of the object

USERS field records the number of allocations that have issued an **ALLOC** command for the object and still using it, *i.e.*, the corresponding **DEALLOC** has not been reached. It is incremented by **ALLOC** and decremented by **DEALLOC**. We show an example for the usage of this flag in Section V-C. The DMA is configured with source address, size and

[1]For example, the Freescale MPC5777M SoC used in previous work [16] includes both SPM memory and a dedicated I/O processor that could be used to implement the described management functionalities.

a destination address extracted from an entry in the object table. DMA operations are scheduled in the DMA queue that allows scheduling multiple DMA transfers and executing them in FIFO order.

The pointer table is used to disambiguate pointers during run time. An entry in the pointer table consists of the main memory address (*MM_ADDR*) of a pointer, flag (*ALIASED*) and a reference to an entry in the object table (*OBJ_TBL_IDX*). If *MM_ADDR* aliases with the main memory address range of an object in the object table, the flag *ALIASED* is set and the index of the aliased object is stored in *OBJ_TBL_IDX*.

B. Allocation Commands

Commands (**ALLOC/ DEALLOC/ GETADDR**) manage the allocation of objects/pointers in the SPM. Table commands (**SETMM/ SETSIZE/ SETPTR**) are used to set the object and pointer tables in the SPM controller.

Two commands, **SETMM** and **SETSIZE**, set the main memory address and the size of an object in the entry *OBJ_TBL_IDX* in the object table:

SETMM *OBJ_TBL_IDX, MEM_ADDR*

SETSIZE *OBJ_TBL_IDX, SIZE*

These commands are used to initialize the object table, add the information of dynamically-allocated objects or change the set of objects tracked by the table during run-time.

A pointer resolution command **SETPTR** is required for pointers:

SETPTR *PTR_TBL_IDX, MEM_ADDR*

This command configures the entry *PTR_TBL_IDX* in the pointer table with memory address *MEM_ADDR*. This memory address is compared with the valid entries of the object table to find the entry of the pointee object. If the pointee is found, *ALIASED* flag is set and the table index of the object is stored in *OBJ_TBL_IDX*. All the allocation commands on pointers check the pointer entry for the aliasing object to use in allocation. Alias checking can be implemented in one cycle using a one-shot comparator or over multiple cycles comparing one entry at a time. If the number of entries in the object table is large, an alias set that specifies which objects that can alias with the pointer can be used to reduce the number of comparisons. For the sake of simplicity of analysis, we assume in this paper a one-cycle implementation.

Next, we present the allocation commands. An allocation command can be issued for an object or a pointer. However, an object entry *OBJ_TBL_IDX* is required for both cases. We use *TBL_IDX* to refer to an entry in the object or pointer table based on a flag *PTR*. If $PTR = 0$, $OBJ\_TBL\_IDX = TBL\_IDX$. If $PTR = 1$, $PTR\_TBL\_IDX = TBL\_IDX$ and *OBJ_TBL_IDX* is obtained from the pointer entry in the pointer table.

ALLOC*XX TBL_IDX, MEM_ADDR, PTR*

DEALLOC *TBL_IDX, PTR*

GETADDR *TBL_IDX, PTR*

ALLOC*XX* command reserves the space in the SPM at *MEM_ADDR* and schedules a DMA transfer if necessary. There are four versions of **ALLOC***XX* command according to the flags (*XX*): **ALLOC**, **ALLOC***P*, **ALLOC***W*, **ALLOC***PW*. The *P* flag directs the controller to prefetch the object from the main memory. The SPM controller will schedule a prefetch transfer for the object and set *PF_OP* flag in the object entry. If *P* flag is not used, the object is allocated directly and *A* flag is set. Otherwise, *A* flag is set once the prefetch transfer completes. The *W* flag sets the *WB* flag in the object entry to direct the controller to copy back the object to the main memory when de-allocated. *P* flag is used in two cases: 1) if, during the execution of the region where the object is allocated, the current value of the object is read or 2) the object is partially written, *e.g.* writing some elements of an array. *W* flag is used if the object is modified, so that the main memory is updated with the new values after de-allocating the object. Note that for local objects defined in a function, there is no need to prefetch the object before its first use in the function or write-back the object after its last use in the function.

DEALLOC command de-allocates the object/pointer. If the *WB* and *U* flags are set in the object entry, the controller will schedule a write-back transfer, set *WB_OP* flag and reset *A* flag. Otherwise, the object will be de-allocated by simply resetting *A* flag.

GETADDR command returns the current address of the object/pointer. For a pointer, if *ALIASED* flag in the pointer entry is not set, MM_ADDR is returned, otherwise the address is checked for the object in *OBJ_ADDR_TBL*. If *PF_OP* or *WB_OP* flag is set for the object, the controller stalls until the DMA completes transferring the object. If no transfer is scheduled or after the transfer finishes, the controller returns *SPM_ADDR* if *A* flag is set and *MM_ADDR* otherwise. **GETADDR** command is only added before the first use of the object after an allocation/de-allocation. The address returned by the command is then applied for all the next uses until another allocation/de-allocation occurs. This process is compiler-automated and does not require per-access address translation from main memory to SPM addresses, as in related work [11]; hence, we do not add extra overhead to the critical path of the processor. For pointers, **SETPTR** and **GETADDR** commands are required if the pointer can alias with the content of the SPM even if the pointer itself is not allocated.

If a prefetch transfer has been scheduled and a **DEALLOC** command is issued for the object to be prefetched, the transfer is canceled as the object is not needed anymore. Also, if a write-back transfer has been scheduled for an object and it was followed by **ALLOC***XX* for the same object, the transfer is canceled if the object is allocated to the same SPM address, otherwise the transfer is not canceled. This is particularly important for allocations within loops, when the object can be allocated to the same address over multiple iterations.

C. Example

Figures 5, 6 depict an example for the allocation process. There are two objects x and y corresponding to entries 3 and 5 in the object table. Also, a pointer p is an argument to function f and uses entry 0 in the pointer table. Figure 5 shows the CFG of two functions where r_1-r_{10} represent regions. x is read/written in r_3, and the pointee of p is read in r_9. Note that function f, comprising regions r_7 to r_{10}, is called from two different call regions, r_4 with $p = \&x$ and r_5 with $p = \&y$. In the example, we assume that x is allocated at address a_1 in the SPM in sequentially composed regions r_2, r_3 and r_4. The argument of function f is allocated at a different address a_2 inside the function.

We use program points ❶ to ❾ to follow the allocation process. Entries 3 and 5 of the object table, Entry 0 of the pointer table and the DMA queue are traced in Figure 6 for

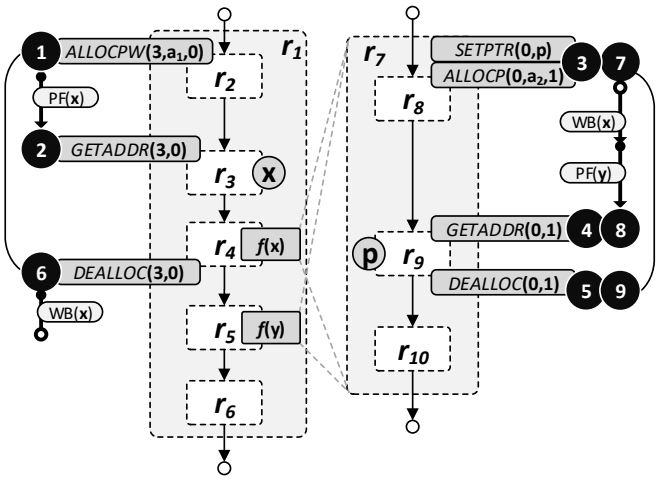


Fig. 5: Allocation Example

these program points. At ❶, x is allocated to address a_1 with P and W flags. In the object table, PF_OP is set to indicate x is being prefetched, WB is set to indicate a write-back when de-allocated, and $USERS$ is incremented. A prefetch transfer $PF(x)$ is scheduled in the DMA queue. At ❷, **GETADDR** checks entry 3 for the address; as $PF(x)$ did not finish at this point, the CPU is stalled. When the prefetch finishes, x is allocated, PF_OP is reset, A is set; and the CPU continues execution. Also, U is set to mark x as used in the SPM. In r_4, function f is called. At ❸, **SETPTR**$(0,p)$ sets the address for p in entry 0 of the pointer table and apply alias checking with the object table. As p aliases with x at this point,flag $ALIASED$ is set and OBJ_TBL_IDX refers to entry 3 in the object table. An allocation of p to a_2 is issued; however, its pointee x is already in the SPM at address a_1. So, no new allocation at a_2 is performed, and $USERS$ is incremented in entry 3 to indicate that two **ALLOC** commands (users) have been executed for x. **GETADDR** at ❹ returns a_1. When p is deallocated at ❺, $USERS$ is decremented in entry 3 of the object table. However, x is not evicted as there is another user for it. When x is deallocated at ❻, x is evicted as this is the last user of x in the SPM. As WB and U are set, a write-back is scheduled for x. f is called again in r_5. The same process to set entry 0 in the pointer table at point ❼ is done with the address of y and OBJ_TBL_IDX refers to entry 5 in the object table. Then, p is also allocated to a_2 with P flag. So, a prefetch is scheduled for y. Before p is used in r_9, **GETADDR** is executed. At this point the write-back transfer of x is done and the prefetch for y is partially completed. The CPU stalls till y is completely prefetched, then U flag is set in entry 5, address a_2 is returned, and the execution continues at ❽. Finally, p is deallocated at ❾ which evicts y. No write-back is needed as WB flag is not set.

An essential observation is that the state of the SPM and the sequence of DMA operations in function f depend on which region calls f: if f is called from r_4, then x is already available in SPM at address a_1, and the allocation of p to a_2 is not used. If instead f is called from r_5, p is allocated to a_2 and the object y must be prefetched from main memory. Therefore, let σ be the *context* under which a region executes, i.e., the sequence of call regions starting from the main function; note that since the main function of the program is not called by any other function, the only valid context for regions in the main is $\sigma = \emptyset$. We denote the execution of a region r_n in a context σ as r_n^σ, which we call a *region-context* pair. Then, allocation decisions, which involve adding allocation commands in the code, must be based on regions, but the state of the SPM and DMA operations, which are needed for WCET estimation, depend on region-context pairs. Intuitively, this is equivalent to considering multiple copies of each region r_n, one for each context in which r_n can execute.

VI. Allocation Problem

We now discuss how to determine a set of allocations for the entire program with the objective to minimize the program's WCET. For the remaining of the section, we let S_{SPM} to denote the size of the SPM. $V = \{v_1, \ldots, v_j, \ldots\}$ is the set of allocatable objects, where $S(v_j)$ denotes the size of object v_j. We let $R = \{r_1, \ldots, r_n, \ldots\}$ be the set of program regions across all functions. Without loss of generality, we assume that region indexes are topologically ordered, so that each parent region has smaller index than its children, each call region has smaller index than the regions in the called function, and sequentially composed regions have sequential indexes; this is also the order used in Figure 5. Note that such topological order must exist since the refined region tree for each function is unique, and furthermore the call graph has no loops due to the absence of recursion. To define the relation between region-context pairs we introduce a parent function $\wp(r_n^\sigma)$ for a region-context r_n^σ in function f as follows: if r_n is the root region of the refined region tree for f, then $\wp(r_n^\sigma) = r_m^{\sigma'}$, where $r_m^{\sigma'}$ is the region-context that calls f in context σ. Otherwise, $\wp(r_n^\sigma) = r_m^\sigma$, where r_m is the parent region of r_n. As an example based on Figure 5, assume that r_4 executes in context σ. Then when r_7 is called from r_4, r_7 executes in context $\sigma \cup r_4$. We further have $\wp(r_7^{\sigma \cup r_4}) = r_4^\sigma$, while for example $\wp(r_8^{\sigma \cup r_4}) = r_7^{\sigma \cup r_4}$. Finally, to generalize the problem for the usage of pointers, let $P = \{p_1, \ldots, p_k, \ldots\}$ be the set of pointers in the program. As the pointee of the pointer can change based on the program flow, we define $\chi_{r_n^\sigma}(p_k)$ as the points-to set for pointer p_k in region-context r_n^σ. For simplicity, we refer to the allocation of the pointee of pointer as the allocation of the pointer. We define $S_{r_n^\sigma}(p_k)$ as the size of p_k in region-context r_n^σ which is computed as:

$$S_{r_n^\sigma}(p_k) = \max_{v \in \chi_{r_n^\sigma}(p_k)} S(v)$$

Note that the pointee of p_k can be a global, local objects from the set of objects in V or a dynamically allocated object. In case of dynamic allocation, v refers to a dynamic allocation site in the program.

We begin by formalizing the conditions under which a set of allocations are feasible as a satisfiability problem. This is similar to a multiple knapsack problem where regions are knapsacks (available space in SPM), except that we add additional constraints to model the relation between regions. Remember that to allocate an object v_j (pointer p_k) in a region r_n, we have to assign an address in the SPM to the object (pointer). Hence, an allocation solution is represented by an assignment to the following decision variables over all regions $r_n \in R$ and all objects $v_j \in V$ (pointers $p_k \in P$):

$$alloc_{r_n}^{v_j} = \begin{cases} 1, & \text{if } v_j \text{ is allocated in } r_n \\ 0, & \text{otherwise} \end{cases}$$

$$assign_{r_n}^{v_j} = \text{address assigned to } v_j \text{ in } r_n$$

$$alloc_{r_n}^{p_k} = \begin{cases} 1, & \text{if } p_k \text{ is allocated in } r_n \\ 0, & \text{otherwise} \end{cases}$$

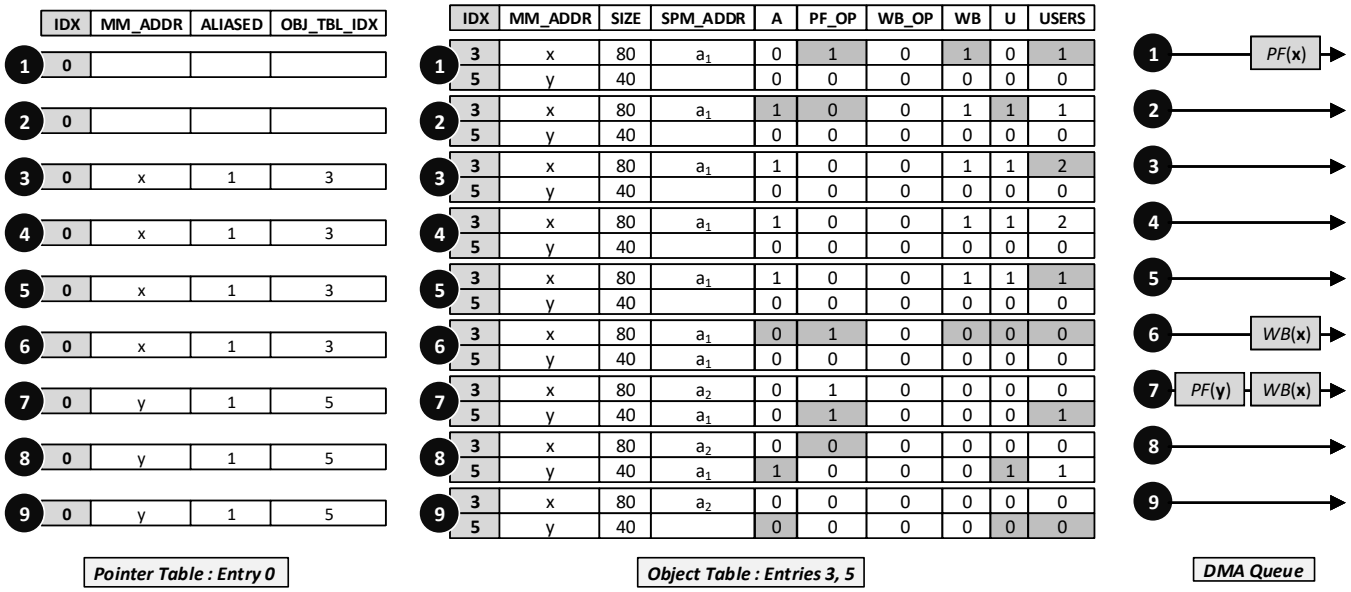


Fig. 6: SPM Controller State for Allocation Example in Figure 5

$assign_{r_n}^{p_k}$ = address assigned to p_k in r_n

An allocation solution is feasible if the allocated objects fit in the SPM at any possible program point. As discussed in Section V-C, the state of the SPM depends on the context under which a region is executed. Hence, we introduce new helper variables to define the availability of an object v_j (pointer p_k) in a region-context r_n^σ:

$$avail_{r_n^\sigma}^{v_j} = \begin{cases} 1, & \text{if } v_j \text{ is available in SPM for execution of } r_n^\sigma \\ 0, & \text{otherwise} \end{cases}$$

$address_{r_n^\sigma}^{v_j}$ = address of v_j in the SPM during execution of r_n^σ

$$avail_{r_n^\sigma}^{p_k} = \begin{cases} 1, & \text{if } p_k \text{ is available in SPM for execution of } r_n^\sigma \\ 0, & \text{otherwise} \end{cases}$$

$address_{r_n^\sigma}^{p_k}$ = address of p_k in the SPM during execution of r_n^σ

We can determine the value of the helper variables based on the allocation. We first discuss the basic constraints assuming that the points-to information is not available. In this case, allocation of a pointer is handled as an allocation of an object. After that, we discuss how the constraints can be modified to consider aliasing between objects and pointers.

1) Basic Constraints: We present a set of necessary and sufficient constraints for an allocation problem in which points-to information is not available.

$$\forall v_j, r_n^\sigma : alloc_{r_n}^{v_j} \vee avail_{\wp(r_n^\sigma)}^{v_j} \Leftrightarrow avail_{r_n^\sigma}^{v_j}. \quad (1)$$

$$\forall p_k, r_n^\sigma : alloc_{r_n}^{p_k} \vee avail_{\wp(r_n^\sigma)}^{p_k} \Leftrightarrow avail_{r_n^\sigma}^{p_k}. \quad (2)$$

Equation 1 (2) simply states that v_j (p_k) is available in the SPM during the execution of r_n^σ if either v_j (p_k) is allocated in r_n, or if v_j (p_k) was already available in the SPM during the execution of the parent region-context pair.

$$\forall v_j, r_n^\sigma : avail_{\wp(r_n^\sigma)}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} = address_{\wp(r_n^\sigma)}^{v_j}. \quad (3)$$

$$\forall p_k, r_n^\sigma : avail_{\wp(r_n^\sigma)}^{p_k} \Rightarrow address_{r_n^\sigma}^{p_k} = address_{\wp(r_n^\sigma)}^{p_k}. \quad (4)$$

$$\forall v_j, r_n^\sigma : \neg avail_{\wp(r_n^\sigma)}^{v_j} \wedge alloc_{r_n}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} = assign_{r_n}^{v_j}. \quad (5)$$

$$\forall p_k, r_n^\sigma : \neg avail_{\wp(r_n^\sigma)}^{p_k} \wedge alloc_{r_n}^{p_k} \Rightarrow address_{r_n^\sigma}^{p_k} = assign_{r_n}^{p_k}. \quad (6)$$

Equations 3, 5 (4, 6) specify the address in the SPM for objects (pointers). If the object (pointer) was already available in the parent region-context, then the address is the same. Otherwise, if the object (pointer) is allocated in r_n, then the address is the one assigned by the allocation.

Finally, given the object (pointer) availability and address for each region-context pair, we can express the feasibility conditions for the allocation problem.

$$\forall v_j, r_n^\sigma : avail_{r_n^\sigma}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} + S(v_j) \leq S_{SPM}. \quad (7)$$

$$\forall p_k, r_n^\sigma : avail_{r_n^\sigma}^{p_k} \Rightarrow address_{r_n^\sigma}^{p_k} + S_{r_n^\sigma}(p_k) \leq S_{SPM}. \quad (8)$$

$$\begin{aligned} \forall v_j, v_k, r_n^\sigma, j \neq k : (avail_{r_n^\sigma}^{v_j} \wedge avail_{r_n^\sigma}^{v_k}) \Rightarrow \\ (address_{r_n^\sigma}^{v_j} + S(v_j) \leq address_{r_n^\sigma}^{v_k}) \vee \\ (address_{r_n^\sigma}^{v_k} + S(v_k) \leq address_{r_n^\sigma}^{v_j}) \end{aligned} \quad (9)$$

$$\begin{aligned} \forall p_j, p_k, r_n^\sigma, j \neq k : (avail_{r_n^\sigma}^{p_j} \wedge avail_{r_n^\sigma}^{p_k}) \Rightarrow \\ (address_{r_n^\sigma}^{p_j} + S_{r_n^\sigma}(p_j) \leq address_{r_n^\sigma}^{p_k}) \vee \\ (address_{r_n^\sigma}^{p_k} + S_{r_n^\sigma}(p_k) \leq address_{r_n^\sigma}^{p_j}) \end{aligned} \quad (10)$$

$$\begin{aligned} \forall v_j, p_k, r_n^\sigma : (avail_{r_n^\sigma}^{v_j} \wedge avail_{r_n^\sigma}^{p_k}) \Rightarrow \\ (address_{r_n^\sigma}^{v_j} + S(v_j) \leq address_{r_n^\sigma}^{p_k}) \vee \\ (address_{r_n^\sigma}^{p_k} + S_{r_n^\sigma}(v_k) \leq address_{r_n^\sigma}^{v_j}) \end{aligned} \quad (11)$$

Equation 7 (8) states that if v_j (p_k) is in the SPM during the execution of r_n^σ, then it must fit within the SPM size. Equations 9 to 11 state that if two objects/pointers are in the SPM during the execution of r_n^σ, then their addresses must not overlap. Note that the size for a pointer is dependent on the region-context pair. Giving a points-to set $\chi_{r_n^\sigma}(p_k)$, the size required for allocating p_k in r_n^σ is the maximum size of all objects in the points-to set.

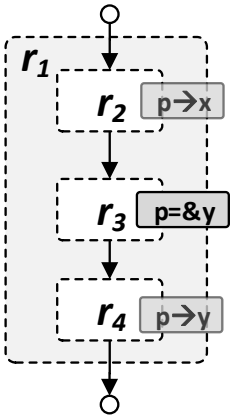


Fig. 7: Example of Pointer Definition

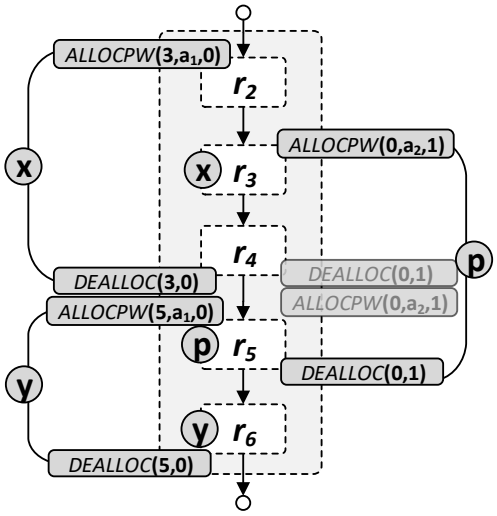


Fig. 8: Allocation Overlap Example

$$\forall p_k, r_n^\sigma : def_{r_n^\sigma}^{p_k} \Rightarrow \neg alloc_{r_n}^{p_k} \tag{12}$$

The constraint in Equation 12 states that the allocation of pointer p_k is not allowed in r_n ($alloc_{r_n}^{p_k} = 0$) if p_k is *defined* in r_n, *i.e.*, p_k can change its reference in the region. This constraint is required for correctness of execution and analysis. This case is depicted in Figure 7 where pointer p is defined in region r_3 to point to y rather than x. Hence, p can be allocated in r_2 and r_4 while $alloc_{r_3}^{p} = 0, alloc_{r_1}^{p} = 0$. The allocation of p in r_3 or r_1 will result in pointer invalidation as any reference to p after its definition in r_3 should point to y not x.

As long as Equations 7 to 11 are satisfied for a given solution in all region-context pairs, all objects fit in the SPM; hence, the allocation problem can be feasibly implemented. To do so, we next discuss how to determine the list of commands (**ALLOC/DEALLOC/GETADDR**) that must be added to each region. For a region r_n that is not sequentially composed, an **ALLOC** is inserted at the beginning of the region and a **DEALLOC** at the end of the region.

In the case of sequential regions, to reduce the number of DMA operations, we note the following: if the same object v_j is allocated in two sequentially composed regions r_p and r_q with the same assigned address, then there is no need to **DEALLOC** v_j at the end of r_p and **ALLOC** it again at the beginning of r_q. Hence, we consider the maximal sequence of sequentially composed regions $r_p, \ldots, r_q$ such that for every region r_n in the sequence: $alloc_{r_n}^{v_j} = 1$ and the address $assign_{r_n}^{v_j}$ assigned to v_j is the same. We then add the **ALLOC** command at the beginning of r_p and the **DEALLOC** command at the end of r_q. The P and W flags of the **ALLOC** command are set as discussed in Section V-B based on the usage throughout the whole sequence. The same procedure applies for a pointer p_k allocated in a sequence of sequentially composed regions.

Also, we note the following for object v_j and pointer p_k such that $v_j \in \chi_{r_n^\sigma}$: if object v_j and pointer p_k are allocated in overlapped sequence of sequentially composed regions, DMA operations on the pointer are inserted to re-locate the pointee to avoid pointer invalidation. The example shown in Figure 8 shows the case where object x is allocated to address a_1 in r_2, r_3 and r_4, pointer p is allocated to address a_2 in r_3, r_4 and r_5, and object y is allocated to a_1 in r_5 and r_6. If p can point to x, it will be already in address a_1 in the SPM when p is allocated in r_3 and x will not be copied to address a_2. However, the copy of x at a_1 should be written-back after r_4 to allocate y to a_1. Using p in r_5 with the assumption that x is in the SPM will result in a conflict. So, a re-location must be guaranteed after r_4, so that the copy of x is moved to a_2 before y is fetched to a_1. Note that the relocation commands will cancel each other if p is not pointing to x.

Example: refer to the example in Figure 5, where p is allocated in two regions in sequence (r_8 and r_9). **ALLOC** is inserted before r_8 and **DEALLOC** is inserted after r_9. P flag is set in **ALLOC** even though x is not used in r_8, but it is read in r_9. Similarly, W is not set as x is not modified in neither r_8 nor r_9.

Finally, to compute the WCET for the program, we need to determine whether an **ALLOC/DEALLOC** command triggers a DMA operation; this again depends on the context σ in which a given region r_n is executed, as demonstrated by the example in Section V-C. As in Equation 3, we know that the **ALLOC** will be canceled if v_j was already available in the parent region-context; hence, for a region r_n that performs an **ALLOC** on v_j and a context σ, the **ALLOC** generates a DMA prefetch on v_j only if both the P flag in the **ALLOC** is set and $avail_{\wp(r_n^\sigma)}^{v_j} = 0$ (similarly for DEALLOC, a DMA operation is generated if the W flag is set and $avail_{\wp(r_n^\sigma)}^{v_j} = 0$).

2) Aliasing Constraints: The feasibility problem can be relaxed using the points-to information of each pointer. Points-to information are derived from a must-may alias analysis. We consider the must-alias points-to sets with object v_j and pointer p_k such that $\chi_{r_n^\sigma}(p_k) = \{v_j\}$; which is a common case with passing by reference in functions. In this case, the allocation of either v_j or p_k in region-context r_n^σ means that both v_j and p_k are available in the SPM in this region-context. The constraints can be extended if there are multiple pointers that only point to v_j in a region-context.

$$alloc_{r_n}^{v_j} \vee alloc_{r_n}^{p_k} \vee avail_{\wp(r_n^\sigma)}^{v_j} \vee avail_{\wp(r_n^\sigma)}^{v_j} \Leftrightarrow avail_{r_n^\sigma}^{v_j}. \tag{13}$$

$$alloc_{r_n}^{v_j} \vee alloc_{r_n}^{p_k} \vee avail_{\wp(r_n^\sigma)}^{v_j} \vee avail_{\wp(r_n^\sigma)}^{p_k} \Leftrightarrow avail_{r_n^\sigma}^{p_k}. \tag{14}$$

Equations 13, 14 replace Equation 1 , 2 and state that v_j and p_k are available in the SPM during the execution of r_n^σ if v_j or p_k is allocated in r_n, or if v_j or p_k was already available in the SPM during the execution of the parent region-context pair. Note that in Equation 14, v_j can be available in the parent region-context $\wp(r_n^\sigma)$ while p_k is not available in it

if p_k changes its reference in the children of $\wp(r_n^\sigma)$, *i.e.*, $\chi_{r_n^\sigma}(p_k) \neq \{v_j\}$. In that case, Equation 14 is not applicable to $\wp(r_n^\sigma)$ and $alloc^{p_k}_{\wp(r_n^\sigma)} = 0$ according to Equation 12.

$$avail^{v_j}_{\wp(r_n^\sigma)} \Rightarrow address^{p_k}_{r_n^\sigma} = address^{v_j}_{\wp(r_n^\sigma)}. \tag{15}$$

$$\begin{aligned} &\neg avail^{v_j}_{\wp(r_n^\sigma)} \wedge alloc^{v_j}_{r_n} \wedge \neg alloc^{p_k}_{r_n} \Rightarrow \\ &address^{v_j}_{r_n^\sigma} = address^{p_k}_{r_n^\sigma} = assign^{v_j}_{r_n}. \end{aligned} \tag{16}$$

$$\begin{aligned} &\neg avail^{v_j}_{\wp(r_n^\sigma)} \wedge alloc^{p_k}_{r_n} \wedge \neg alloc^{v_j}_{r_n} \Rightarrow \\ &address^{p_k}_{r_n^\sigma} = address^{v_j}_{r_n^\sigma} = assign^{p_k}_{r_n}. \end{aligned} \tag{17}$$

$$\begin{aligned} &\neg avail^{v_j}_{\wp(r_n^\sigma)} \wedge alloc^{v_j}_{r_n} \wedge alloc^{p_k}_{r_n} \Rightarrow \\ &(address^{v_j}_{r_n^\sigma}, address^{v_j}_{r_n^\sigma}) = \gamma(assign^{v_j}_{r_n}, assign^{p_k}_{r_n}). \end{aligned} \tag{18}$$

Equation 3 still applies for v_j as the availability in the parent dominates any allocation in the region. However, the address p_k inherits the address of the v_j if it is available in its parent as in Equation 15. If v_j is not available in the parent, there are two cases:

Equations 16,17 state that if only v_j or p_k is allocated, the address of v_j and p_k is the assigned address of the allocated one.

Equation 18 state that if both v_j and p_k are allocated, the assigned address for each of them to be determined with an arbitrary function γ that depends on how the allocation is implemented. In this work, we use $\gamma(assign^{v_j}_{r_n}, assign^{p_k}_{r_n}) = assign^{v_j}_{r_n}$ as we consider relocation of the pointer as we illustrated before in the example shown in Figure 8.

Example: refer to the example in Figure 5, where p is allocated with assigned address a_2 in r_8. For context $\sigma \cup r_5$, we have $avail^{p}_{r_7^{\sigma\cup r_5}} = avail^{y}_{r_7^{\sigma\cup r_5}} = 0$, since $\chi_{r_7^{\sigma\cup r_5}}(p) = \{y\}$ and y is not available in r_5^σ, the parent of $r_7^{\sigma\cup r_5}$. Hence, we also have $address^{p}_{r_8^{\sigma\cup r_5}} = address^{y}_{r_8^{\sigma\cup r_5}} = a_2$. However, for context $\sigma \cup r_4$ we obtain $avail^{p}_{r_7^{\sigma\cup r_4}} = avail^{x}_{r_7^{\sigma\cup r_4}} = 1$, since $\chi_{r_7^{\sigma\cup r_4}}(p) = \{x\}$ and x is available in r_4^σ. So, we get $address^{p}_{r_7^{\sigma\cup r_4}} = address^{x}_{r_7^{\sigma\cup r_4}} = a_1$.

The constraints for SPM size are the same as in Equations 7, 8. Equations 10, 11 for address overlap are not applied for must alias cases. That is, if $\chi_{r_n^\sigma}(p_{k1}) = \chi_{r_n^\sigma}(p_{k2}) = \{v_j\}$, then their address ranges match $address^{p_{k1}}_{r_n^\sigma} = address^{p_{k2}}_{r_n^\sigma} = address^{v_j}_{r_n^\sigma}$. So, Equations 11, 10 is not applied between p_{k1}, p_{k2}, v_j for region-context r_n^σ.

We illustrated the possible aliasing constraints for one pointer and one object. Another set of constraints can be derived for aliasing pointers or pointers with multiple pointees in their points-to set. We do not detail these constraints as they exploit a may-alias which means the constraints do not represent necessity.

A. WCET Optimization

For a given allocation solution $\{alloc^{v_j}_{r_n}, assign^{v_j}_{r_n}, alloc^{p_k}_{r_n}, assign^{p_k}_{r_n} | \forall v_j, p_k, r_n\}$, the described procedure determines the set of objects available in the SPM and the set of DMA operations for each region-context r_n^σ. Assuming that bounds on the time required for SPM and main memory accesses are known, this allows us to determine the benefit (WCET reduction) for every trivial region in context σ, as well as the length of DMA operations. For a dynamic allocation approach without prefetch, the length of DMA operations could simply be summed to the execution time of the corresponding region, since DMA operations stall the core.

However, for our proposed approach with prefetching, the cost of DMA operations depend on the overlap: since DMA works in transparent mode, for a trivial region the maximum amount of overlap is equal to the execution time of its code minus the time that the CPU accesses main memory directly. Furthermore, since the length of DMA operations is generally longer than the execution of a trivial region, the total overlap depends on the program flow. Therefore, we compute the amount of overlap as part of an integrated WCET analysis, which we present in Section VII. We solve the allocation problem by adopting a heuristic approach that first searches for feasible allocation solutions, and then run the WCET analysis on feasible solutions to determine their fitness; we discuss it next in Section VI-B.

Finally, we note that the proposed region-based allocation scheme is a generalization of the approaches used in related work on dynamic allocation. In [10], the authors applied a structured analysis to choose a set of variables for static allocation. They analyzed innermost loop as Directed Acyclic Graph (DAG) for worst case path and then collapsed the loop into a basic block to analyze the outer loop. The region tree representation captures this structure as loops, conditional statements and functions as regions. The dynamic allocation in [5] is based on program points around loops, if statements and functions which can be matched with an entry/exit of a region. In [13], Deverge *et al*. proposed a general graph representation that allows different granularities of allocation. The authors formulated the dynamic allocation problem based on the flow constraints which can also be applied to the region representation. All such approaches use heuristics to determine the overall program allocation. Hence, to allow a fair evaluation focused on the benefits of data prefetching, in Section IX we compare our proposed scheme against a standard dynamic allocation approach with no overlap using the same region-based program representation and search heuristic.

B. Allocation Heuristic

The allocation heuristic adopts a genetic algorithm to search for near-optimal solutions to the allocation problem.

- **Chromosome Model:** The chromosome is a binary string where each bit represents one of the $alloc^{v_j}_{r_n}$ decision variables. Note that we do not represent the $assign^{v_j}_{r_n}$ decision variables in the chromosome; instead, we use a fast address assignment algorithm as part of the fitness function to find a feasible address assignment for a chromosome.
- **Fitness Function:** The fitness fit of a chromosome represents the improvement in the WCET of the program with this allocation if it is feasible. The fitness function first applies the address assignment algorithm to the chromosome. If the allocation is not feasible, the chromosome has $fit = 0$. Otherwise, we execute the WCET analysis after the program is transformed to insert the allocation commands; the fitness of the allocation is then assigned as $fit = WCET_{MM} - WCET_{alloc}$ where $WCET_{MM}$ is the WCET with all the objects in main memory and $WCET_{alloc}$ is the WCET for the analyzed solution.
- **Initialization:** The initial population $P(0)$ is generated randomly with feasible solutions, *i.e.*, $fit > 0$.

Algorithm 1 Address Assignment

Input: region information, $\{alloc_{r_n}^{v_j}, alloc_{r_n}^{p_k} | \forall v_j, p_k, r_n\}$

1: **for all** region r_n by increasing index starting with r_1 **do**
2: $end\_addr_{r_n} \leftarrow$ ASSIGN_ADDRESSES(r_n)

3: **function** ASSIGN_ADDRESSES(r_n)
4: $end\_addr_{r_n} = \max_\sigma\{end\_addr_{\wp(r_n^\sigma)}\}$
5: **if** r_{n-1} is not sequentially composed with r_n **then**
6: **for all** v_j such that $alloc_{r_n}^{v_j}$ **do**
7: $assign_{r_n}^{v_j} \leftarrow end\_addr_{r_n}$
8: $end\_addr_{r_n} \leftarrow end\_addr_{r_n} + S(v_j)$
9: **for all** p_k such that $alloc_{r_n}^{p_k}$ **do**
10: $assign_{r_n}^{p_k} \leftarrow end\_addr_{r_n}$
11: $end\_addr_{r_n} \leftarrow end\_addr_{r_n} + \max_\sigma(S_\sigma(p_k))$
12: **else**
13: **for all** v_j such that $alloc_{r_n}^{v_j} \wedge alloc_{r_{n-1}}^{v_j}$ **do**
14: $assign_{r_n}^{v_j} \leftarrow assign_{r_{n-1}}^{v_j}$
15: **for all** p_k such that $alloc_{r_n}^{p_k} \wedge alloc_{r_{n-1}}^{p_k}$ **do**
16: $assign_{r_n}^{p_k} \leftarrow assign_{r_{n-1}}^{p_k}$
17: **for all** v_j such that $alloc_{r_n}^{v_j} \wedge \neg alloc_{r_{n-1}}^{v_j}$ **do**
18: Compute $assign_{r_n}^{v_j}$ using best fit based on already assigned addresses
19: **for all** p_k such that $alloc_{r_n}^{p_k} \wedge \neg alloc_{r_{n-1}}^{p_k}$ **do**
20: Compute $assign_{r_n}^{p_k}$ using best fit based on already assigned addresses
21: $max_{r_n}^{v} \leftarrow \max_{v_j \text{ s.t. } alloc_{r_n}^{v_j}}\{assign_{r_n}^{v_j} + S(v_j)\}$
22: $max_{r_n}^{p} \leftarrow \max_{p_k \text{ s.t. } alloc_{r_n}^{p_k}}\{assign_{r_n}^{p_k} + \max_\sigma(S_\sigma(p_k))\}$
23: $end\_addr_{r_n} \leftarrow \max\{max_{r_n}^{v}, max_{r_n}^{p}\}$

- **Evolution Operations:** The evolution process incorporates random random selection, one-point crossover and random bit mutation to generate $P'(t+1)$. The elite chromosomes with highest fitness from $P(t)$ and $P'(t+1)$ are chosen to form the next population $P(t+1)$.
- **Termination:** The algorithm is terminated after k generation or if the best chromosome does not change for n generations.

The address assignment algorithm is depicted in Algorithm 1. Given a chromosome, the region tree is traversed in topological order assigning addresses to the allocated objects and pointers in each region. The topological order visits all the nodes with the same parent before visiting the children. For the root of a function, all the parents (call regions) of the function are visited before the root of the function. Also, for a sequence of sequentially composed regions, the order of the sequence is maintained. After the objects in a region are assigned to SPM addresses, an end address to the last allocated address is maintained. For each region r_n, the previous end address is the maximum of all parent regions (note that if r_n is not the root of its function, it has a single parent region). For a region that is not sequentially composed or the first region in a sequence of regions, addresses are iteratively assigned to the allocated objects starting from the previous end address. For a region in a sequence, an allocated object maintains the same address as the previous region if the object is allocated in both. Otherwise, a best fit algorithm is used to assign the remaining addresses. The end address for each region is then computed as the maximum end address for any allocated object. Note that the algorithm trivially ensures that objects/pointers allocated in a region cannot overlap with any object or pointer that is available in a parent; hence, Equations 9, 11 are always satisfied. However, the algorithm is not optimal, since it does not consider that an allocation might not be required in any context where the object is already available in the SPM or the aliasing between objects and pointers. Finally, the allocation is considered feasible only if the end address never exceeds the SPM size; this guarantees that Equations 7, 8 are also satisfied.

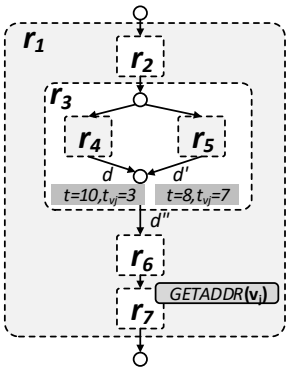


Fig. 9: WCET Example: Merging states from different paths

VII. WCET Analysis

We discuss how to model the behavior of our prefetch mechanism in the context of static timing analysis so that a safe bound to the WCET of the program running uninterrupted can be computed. We assume a given allocation solution computed based on Section VI. We rely on the standard approach of Data Flow Analysis (DFA) [24], where the detailed state of the hardware is generalized into an *abstract state* based on the theory of abstract interpretation [25], [26]. To avoid maintaining a different state for each path through the program, the analysis relies on computing fixed points by "merging" states when paths joins (i.e., branch join and loops entry/exit). In detail, given two abstract states d and d', we need to compute a *join operator* $\vee$ such that the resulting state $d'' = d \vee d'$ is more general than either d or d'. We model time as natural numbers, i.e., processor clock cycles.

We begin by providing an intuitive discussion of the challenges of handling our prefetching mechanism, followed by our intended solutions. In what follows, we use function $(x)^+$ as a shorthand or $\max(0,x)$ and $\mathscr{P}(A)$ to denote the powerset of set A. As discussed in Section IV, let $\bar{G}_f = (\bar{N}, \bar{E})$ to denote the refined CFG for function f. To keep track of the program execution, it is useful to formally define the concept of program state:

Definition 1 (State): The program execution is defined as the transformation of a *program state*. We let Σ be the set of all possible program states; we use $\mathbf{s} \in \Sigma$ to denote an individual state and $S \subseteq \Sigma$ to denote a set of states. The state at any given point in the execution of the program represents the amount of *elapsed time* $\mathbf{s}.t$ since the beginning of the program, and the content of all hardware registers and memories.

Definition 2 (Transfer Function): For every edge $e : BB_i \rightarrow BB_j$ in $\bar{G}_f$ and context σ for f, we define a transfer function $\mathscr{T}_{e,\sigma} : \Sigma \rightarrow \Sigma$ such that: if $\mathbf{s}$ is the program state at the beginning of the execution of BB_i and the program execution

flows from BB_i to BB_j, then $s' = \mathscr{T}_{e,\sigma}(\mathbf{s})$ is the program state at the beginning of the execution of BB_j. Function $\mathscr{T}'_{e,\sigma} : \mathscr{P}(\Sigma) \to \mathscr{P}(\Sigma)$ denotes the obvious set-extension of function $\mathscr{T}_{e,\sigma}$, i.e. $\mathscr{T}'_{e,\sigma}(S) = \cup_{\mathbf{s}\in S}\mathscr{T}_{e,\sigma}(\mathbf{s})$.

Note that based on Definition 2 [2], if the execution cannot flow from node BB_i to BB_j for any state in S, then $\mathscr{T}'_{e,\sigma}(S) = \emptyset$. Given a set of initial program states S_{entry} with $t = 0$ [3], a WCET analysis could then simply proceed as follows: enumerate all paths through every function in the program; for each path through the program, iteratively apply function $\mathscr{T}'_{e,\sigma}$ starting from state set S_{entry} for all edges comprising the path, obtaining a set of final states S_{exit}. The WCET can then be obtained as the maximum time elapsed for any state in any final state set S_{exit}. Since based on our assumptions the number of paths through the whole program is finite, this approach is computable, but it is generally computationally intractable for all but the simplest programs, as the number of paths is exponential in the number of branches/loops in the program.

To obtain a tractable analysis, WCET techniques typically attempt to prune paths that cannot lead to the WCET by making *local* decisions: ideally, we could examine each branch point in the CFG one at a time, determine which branch leads to the WCET, and exclude from the analysis all other branches, thus implicitly identifying the Worst Case Execution Path (WCEP) through the program. In practice, this might not be possible, because the worst case path through a branch might depend on the path taken through another branch preceding or following the one under analysis, either due to the program semantic (i.e., some paths might be invalid) or due to architectural considerations (i.e., an hardware operation started during a basic block might influence the timing of a successive basic block).

Since our objective is to show how to integrate our proposed prefetching scheme in existing WCET frameworks, in the rest of the section we focus on architectural analysis. To explain how we model the behavior of DMA operations, consider as an example the execution of the CFG and associated region tree in Figure 9 in a context σ. Assume that the analysis for the path through r_4^σ has computed a program state $\mathbf{s}$ with an upper bound to the execution time of the program up to this point equal to $t = 10$, and an upper bound to the remaining time to complete a DMA fetch operation for an object v_j equal to $t_{v_j} = 3$. For the path through r_5^σ, we instead have a state $\mathbf{s}'$ with $t = 8, t_{v_j} = 7$, i.e., the execution takes longer along the path through r_4^σ than through r_5^σ, but results in a shorter remaining DMA time. Assume now that a **GETADDR** command on object v_j is executed at the beginning of region/context r_7^σ. The amount of time that the command will block is then equal to t_{v_j} minus the amount of overlap that the DMA operation has with r_6^σ, or zero if the operation completes during r_6^σ. Assume a simple case where the execution through r_6^σ requires Δ units of time and performs no access to main memory, so that the DMA operation can overlap up to Δ. The program can then resume from **GETADDR** at time $t + \Delta + \max(t_{v_j} - \Delta, 0)$. Hence, note that for $\Delta = 7$, the worst case path is through r_4^σ, resulting in a time of 17 units against 15 for the path through r_5^σ. However, for $\Delta = 3$, the worst case path is through r_5^σ, with a time of 15 time units against 13 for the path through r_4^σ. In summary, we cannot determine which path through a branch leads to the worst case unless we analyze the regions following the branch in the CFG (r_6^σ and r_7^σ in the example). This shows that the WCEP determination is a *global* decision.

A typical solution to the global decision problem is to employ a *Meet Over Path* (MOP) solution: if we do not know which state to use for b_4, we can *abstract* the execution of the program by considering a new join state $\mathbf{s}''$ that is worse than either $\mathbf{s}$ or $\mathbf{s}'$. Such state does not need to represent any real execution of the system (i.e., it is *abstract*), as long as we can prove that the WCET obtained based on $\mathbf{s}''$ is no smaller than the ones determined based on $\mathbf{s}$ and $\mathbf{s}'$. In this case, a trivial solution would be to computing a join state $\mathbf{s}''$ with $t = \max(10,8) = 10$ and $t_{v_j} = \max(3,7) = 7$. However, this would lead us to over-approximate the time for the **GETADDR**, resulting in 17 time units for $\Delta = 3$, rather than the computed bound of 15 time units. Therefore, we seek to derive a tighter abstraction.

Intuitively, this can be achieved by abstracting the states $\mathbf{s}$ and $\mathbf{s}'$ for the execution through r_4^σ and r_5^σ into abstract states d and d'. An abstract state d is composed of two information: the elapsed program execution time $d.t$, and a set of *timers* $\{t_{v_j}\}$. For an object v_j, $d.t_{v_j}$ represents the worst case time required to complete either a prefetch or write-back operation in the allocation queue; since the allocation queue is served in FIFO order, this represents the time to transfer that specific object, plus the time required for all operations ahead of it in the queue. For the example in Figure 9, let d be the state through r_4^σ and d' be the state through r_5^σ. Since there is only one DMA operation in the queue, we have $d.t = 10, d.t_{v_j} = 3$ and $d'.t = 8, d'.t_{v_j} = 7$, i.e., the abstract states are equivalent to the corresponding program states. The join state $d'' = d \vee d'$ is then computed as follows:

$$d''.t = t_{\max} = \max(d.t, d'.t), \tag{19}$$

and for every timer t_{v_j}:

$$d''.t_{v_j} = \max\left(d.t_{v_j} - (t_{\max} - d.t), d'.t_{v_j} - (t_{\max} - d'.t)\right). \tag{20}$$

Based on Equations 19, 20, we compute a join state for the example $d''.t = \max(10,8) = 10, d''.t_{v_j} = (3 - (10-10), 7 - (10-8)) = 5$. Note that this abstraction is tighter compared to the values $t = 10, t_{v_j} = 7$ obtained by the trivial over-approximation; in particular, it is easy to see that for the provided example, the time for the **GETADDR** command computed based on d'' is *exactly* equal to the worst case between d and d' for any value of Δ, albeit for more complex cases involving multiple DMA operations it is still a (tighter) over-approximation. However, the abstraction does not correspond to any "real" program state, since the values of t and t_{v_j} are different than the program state at r_7^σ for either execution paths. The key intuition is that adding Δ units of time to the execution time of the program is always worse than adding Δ units of time to the length of timers, since a **GETADDR** might block the program for a time at most equal to the length of the corresponding timer. Hence, if the execution time along two paths differs by a value Δ, we are guaranteed to obtain an upper bound if we consider the longest execution time but subtract Δ units of time from the timers along the shortest path, as performed in Equation 20.

Note that in general, a single DMA operation could overlap with many regions, and the amount of overlap can be further modified by the path through each region and

[2] Also note that $\mathscr{T}_{e,\sigma}$ defines a *deterministic* machine: assuming we know the state $\mathbf{s}$ at the beginning of the basic block, we can compute the exact state $\mathbf{s}'$ along e. If the machine is non-deterministic, the definition can be modified to return a set of states rather than a single state while maintaining the same theoretical framework, see [26].

[3] Note that in general, a set of program states must be considered, rather than a single state, because the initial state of the hardware, including the program inputs, is not known.

allocation commands for both the same and other objects. Due to the presence of the max term in Equation 20, modeling the WCET problem as an ILP (a technique also known as implicit path enumeration [24]) would require adding a large number of auxiliary variables. Therefore, we propose to instead compute the WCET by performing the MOP procedure using a structure-based approach [24] that relies on the region tree, as summarized in Algorithm 2.

Algorithm 2 WCET Analysis

Input: initial program state d with $d.t = 0$, region information, allocation solution

```
1: d ← ANALYZE_REGION(r_1, ∅, d)
2: return d.t + max_{v_j}{d.t_{v_j}}

3: function ANALYZE_REGION(r, σ, d)
4:     if r is trivial region then
5:         d ← STATE_TRANSFER(r, σ, d)
6:         if r calls a region r_n then
7:             d ← ANALYZE_REGION(r_n, σ ∪ r, d)
8:     else
9:         for all paths p_i in r do
10:            d_i ← d
11:            for all subregions r_n along p_i do
12:                d_i ← ANALYZE_REGION(r_n, σ, d_i)
13:        d ← JOIN(r, σ, {d_i})
14:    return d
```

Starting from an initial abstract program state d and region r_1, the root of the main function, the algorithm recursively calls function *ANALYZE_REGION* to update state d based on the execution of region r in context σ. If r is a trivial region, then function *STATE_TRANSFER* is used to update d based on the region's code, including any allocation command. Note that we need to pass the context σ to the function, since as explained in Section VI, the availability and address of objects in the scratchpad depends on the context for the region. If the region is a call region, we also need to recursively invoke *ANALYZE_REGION* on the called region after updating the context. If region r is not trivial, then we need to recursively analyze all sub-regions along every path in r; this results in an updated state d_i for each path p_i. The states are then joined by function *JOIN*. If region r has no backedge (i.e., it is not a loop), then the function simply applies the join operator over all states d_i. If the region is a loop, then function *JOIN* performs a fixed-point iteration over the abstract state based on loop iteration bounds (since such fixed point iteration is a well-understood technique in DFA [25], [26], we do not discuss it further). At the end of the analysis, we return the total elapsed time plus the maximum timer length, to indicate the need to complete any remaining write back operation.

In the next section, we first provide required preliminaries on the underlying mathematical principles of DFA using the MOP approach. We then formally introduce our abstraction and prove it correct in Section VII-B. Note that while Algorithm 2 enumerates region, in practice the only regions that contain code and must thus be analyzed are trivial regions, containing one basic block each. Hence, for simplicity and to be consistent with previous analyses, we discuss the MOP procedure over basic blocks using the refined CFG $\bar{G}_f$. Finally, note that while we focused on modeling the behavior of DMA operations, the abstract state can also model both architectural states, such as the state of the processor pipeline [26], as well as the value of program variables, which can be used to exclude invalid paths (flow analysis) and compute loop bounds [27].

A. Preliminaries

The theory of abstract interpretation [25] provides a formal way to describe a mathematical model for the state of the program. In this section, we base our discussion on the formulation of DFA with abstract interpretation for WCET analysis proposed in [26].

Definition 3 (Bounds for Partially Ordered Set): Consider a set A with partial order $\leq_A$. We say that an element $a \in A$ is an upper bound (lower bound) for a subset Y of A iff $\forall y \in Y : y \leq_A a$ (respectively, $a \leq_A y$). We further say that a is the unique least upper bound (greatest lower bound) for Y, and write $a = \vee_A Y$ (respectively, $a = \wedge_A Y$) iff for all other upper bounds b of Y it holds $a \leq_A b$ (respectively, $b \leq_A a$).

For simplicity, for a set $Y = \{a, b\}$, we shall write $a \vee_A b$ ($a \wedge_A b$) as a shorthand for $\vee_A Y$ ($\wedge_A Y$).

Definition 4 (Complete Lattice): A partially ordered set $(A, \leq_A)$ is said to be a complete lattice if any subset Y of A admits both a least upper bound and a greatest lower bound.

Observation 5 (Concrete State Set): The set $\mathscr{P}(\Sigma)$ together with the subset partial relation $\subseteq$ is a complete lattice, where $S \vee S' = S \cup S'$ and $S \wedge S' = S \cap S'$.

The complete lattice $\left(\mathscr{P}(\Sigma), \subseteq\right)$ is used to model the "real" (concrete) state of the system; in this sense, the partial order $\subseteq$ represents a relation of generality, in the sense that if $S \subseteq S'$, we can say that S' is more general (since it contains more program states), or equivalently less precise, compared to S.

Definition 6 (Monotone Function): Let $(A, \leq_A)$ and $(B, \leq_B)$ be partially ordered sets. A function $f : A \to B$ is said to be monotone iff: $\forall a, a' \in A : a \leq_A a' \Rightarrow f(a) \leq_B f(b)$.

Definition 7 (Abstraction): We say that a complete lattice $(D, \leq_D)$ is an abstraction for the concrete state $\left(\mathscr{P}(\Sigma), \subseteq\right)$ iff there exists a monotone function $\gamma : D \to \mathscr{P}(\Sigma)$ such that:

$$\forall S \in \mathscr{P}(\Sigma) : \exists d \in D : S \subseteq \gamma(d). \tag{21}$$

γ is also called the *concretization* function of the abstraction. Since the concretization function is monotone, for every $d \leq_D d'$, it must hold: $\gamma(d) \subseteq \gamma(d')$. In other words, the partial order $\leq_D$ on D must express a relation of generality similar to the one for the concrete state. Furthermore, Equation 21 ensures that for every concrete state S, there exists an abstract state d that "contains" S.

Based on the described framework, the MOP DFA is then carried out as follows: we first obtain an initial abstract state d_{entry} such that $S_{entry} \subseteq \gamma(d_{entry})$. We then traverse the DFG using an abstract transfer function $\hat{\mathscr{T}}_{e,\sigma} : D \to D$, which represents the abstraction of the transfer function $\mathscr{T}_{e,\sigma}$ to the abstract state D. Whenever we need to join paths for two abstract states d, d', we compute a new join state $d'' = d \vee_D d'$. After obtaining a final abstract state d_{exit} for the program, we then determine the WCET as the largest elapsed time in $\gamma(d_{exit})$. There are two fundamental advantages to this approach: 1) as discussed in the example in Section VII, we can represent states that cannot occur in the concrete execution of the system. 2) Since the abstract state set D is a model of the system, we can ignore program and architectural details that are too complex to handle in the analysis, albeit at the cost of decreased analysis precision. Overall, the goal is to obtain an abstract transfer function

$\hat{\mathscr{T}}_{e,\sigma}$ that can be computed in a reasonable amount of time, rather than evaluating $\mathscr{T}_{e,\sigma}$ on all program states contained in a concrete state set S, which is generally computationally intractable. The following theorem states the fundamental sufficient condition on the abstract transfer function that we use in this work.

Theorem 8 (MOP II Correctness; Theorem 3.3.5 in [26]): Let D be an abstraction for $\mathscr{P}(\Sigma)$. If for every edge e, the transfer function $\hat{\mathscr{T}}_{e,\sigma}$ satisfies the following property:

$$S \subseteq \gamma(d) \Rightarrow \mathscr{T}'_{e,\sigma}(S) \subseteq \gamma(\hat{\mathscr{T}}_{e,\sigma}(d)), \tag{22}$$

then the MOP analysis over D using an initial state d_{entry} : $S_{entry} \subseteq \gamma(d_{entry})$ is a correct analysis for the program, meaning that $S_{exit} \subseteq \gamma(d_{exit})$.

Intuitively, Equation 22 means that applying the abstract transfer function $\hat{\mathscr{T}}_{e,\sigma}(d)$ results in a state that is more general compared to applying the concrete transfer function $\mathscr{T}'_{e,\sigma}$. In turn, this implies that if we start with an initial abstract state d_{entry} that is more general than the initial concrete state S_{entry}, we will obtain a final abstract state d_{exit} that is still more general (hence, a safe approximation) than the final concrete state S_{exit}.

B. Abstract State Model

We detail our abstraction for WCET analysis in this section. Since our goal is to show how to handle the scratchpad controller, for the sake of simplicity we will consider the simplest possible model for the rest of the hardware system, namely, an in-order CPU where the number of clock cycles required to process the instructions in each basic block does not depend on previous block (i.e., no pipelining effects between blocks), and memory accesses stall the CPU. Under this model, we let t_{comp} be the maximum computation time for a code block without considering the stall time due to load/store operations, t_{spm} be the maximum time for SPM accesses, and t_{mm} the maximum time for main memory accesses; the total execution time of the basic block can then be bounded as $t_{comp}+t_{spm}+t_{mm}$ plus the GETADDR blocking time. We will also not include any memory state (i.e., value assigned to variables) in the abstract state. However, please note that both memory state and other architectural states could be included in the abstract state following well-established WCET analysis techniques [26]. Finally, again for simplicity and to match our implementation, we will assume that all scratchpad commands can be executed in one clock cycle, i.e. we do not handle the command queue. However, if the alias check takes multiple clock cycles, the effects of the command queue could be handled by adding an additional timer to the abstract state, as it will become clearer in the rest of the discussion.

Based on Theorem 8, in the rest of the section we provide the following steps:

- define abstract state set D and its partial order $\leq_D$. This is done in Definitions 11 and 14;
- prove that $(D, \leq_D)$ is a complete lattice (Lemma 15);
- define concretization function γ (Definition 18), prove that it is monotone and it satisfies Equation 21 (Lemma 19);
- define $\hat{\mathscr{T}}_{e,\sigma}(d)$ (Definition 25);
- finally, prove that Equation 22 holds (Theorem 31).

This ensures that all assumptions in Theorem 8 hold, hence proving that the MOP analysis over the described abstraction is correct.

We begin by providing a definition for the program state $\mathbf{s}$ that will be used throughout the section. In what follows, let t^x_{dma} denote the time required for the DMA operation (prefetch or write-back) for an object x, while for a pointer x it denotes the maximum DMA operation time of any object pointed to by x.

Definition 9 (Trailing DMA time): We define the *trailing length* of any DMA operation in the allocation queue as follows:

- the trailing length of the operation at the front of the queue is the time remaining to complete the operation;
- the trailing length for any other operation on an object v is t^v_{dma} plus the trailing length of the operation immediately ahead in the queue.

Essentially, the trailing length for an operation represents the maximum DMA time required to complete it, considering that operations in the allocation queue are served in FIFO order.

Definition 10 (Abstract Timers): Let $\mathscr{V}$ be the set of all objects and $\mathscr{A}$ be the set of all addresses assigned to objects/pointers by the address assignment algorithm. We define the following set of *abstract timers*:

- For an object v, the abstract prefetch timer $\mathbb{T}^{pr}_v$ is a single value $\mathbb{T}^{pr}_v.t \in \mathbb{N}$.
- For an object v, the abstract write-back timer $\mathbb{T}^{wb}_v$ is a tuple $\{\mathbb{T}^{pr}_v.t, \mathbb{T}^{pr}_v.A\}$ with $\mathbb{T}^{wb}_v.t \in \mathbb{N}$ and $\mathbb{T}^{wb}_v.A \in \mathscr{P}(\mathscr{A})$.
- For a pointer p, the abstract prefetch timer $\mathbb{T}^{pr}_p$ is a tuple $\{\mathbb{T}^{pr}_p.t, \mathbb{T}^{pr}_p.V\}$ with $\mathbb{T}^{pr}_p.t \in \mathbb{N}$ and $\mathbb{T}^{pr}_p.V \in \mathscr{P}(\mathscr{V})$.
- For a pointer p, the abstract write-back timer $\mathbb{T}^{wb}_p$ is a tuple $\{\mathbb{T}^{wb}_p.t, \mathbb{T}^{wb}_p.A, \mathbb{T}^{wb}_p.V\}$ with $\mathbb{T}^{wb}_p.t \in \mathbb{N}$, $\mathbb{T}^{wb}_p.V \in \mathscr{P}(\mathscr{V})$ and $\mathbb{T}^{wb}_p.A \in \mathscr{P}(\mathscr{A})$.

For simplicity, we use the symbol $\mathbb{T}$ to denote any abstract timer, defining $\mathbb{T}.V = \{v\}$ for the timers of object v, and $\mathbb{T}.A = \emptyset$ for prefetch timers. We call the value t the timer's *trailing length*, A its *address set*, and V its *points-to set*. We write $\mathbb{T} = 0$ to mean $\mathbb{T}.t = 0, \mathbb{T}.A = \emptyset, \mathbb{T}.V = \emptyset$.

Definition 11 (DMA Abstraction): An abstract state d is a tuple $d = \{d.t, \ldots, d.\mathbb{T}^{pr}_{v_i}, d.\mathbb{T}^{wb}_{v_i}, \ldots, d.\mathbb{T}^{pr}_{p_k}, d.\mathbb{T}^{wb}_{p_k}, \ldots\}$, comprising one prefetch and one write-back timer for each object v_i and each pointer p_k. We call $d.t \in \mathbb{N}$ the *abstract elapsed time*. Let D be the set of all abstract states.

Intuitively, an abstract state is composed of an elapsed time, which is an upper bound to the time elapsed since the beginning of the program, and a prefetch and write-back timer for every object and every pointer. For all timers, the trailing length $\mathbb{T}.t$ models the trailing length of any prefetch or write-back operation for that object/pointers. In essence, our abstraction models the cumulative DMA time required for the operations of a given object/pointer, rather than the ordered list of DMA operations. Since the same pointer can point to different objects during its lifetime, pointer timers must also store the point-to list $\mathbb{T}.V$. Finally, write-back timers additionally store the address at which the object/pointer was allocated. As explained in Section V-B, this is required to cancel a write-back operation if the same object is allocated at the same address. To allow the MOP procedure, $\mathbb{T}.A$ must be defined as a set of addresses (i.e., an element of the powerset of $\mathscr{A}$) so that the union over different paths can be computed.

To simplify notation, we further define the following intuitive operations on timers.

Definition 12 (Operations on Timers): We define the following operations, where $\Delta \in \mathbb{N}$.

- $\mathbb{T}' = \mathbb{T} + \Delta$ returns the timer $\mathbb{T}'$ where $\mathbb{T}.t$ is incremented by Δ.

- $\mathbb{T}' = \mathbb{T} - \Delta$ returns the timer $\mathbb{T}'$ where $\mathbb{T}.t$ is decremented by Δ if $\Delta < \mathbb{T}.t$; otherwise, $\mathbb{T}' = 0$.
- $\mathbb{T}' = \mathbb{T} \setminus v$ returns the timer $\mathbb{T}'$ where $\mathbb{T}'.V = \mathbb{T}.V \setminus \{v\}$ if $\mathbb{T}.V \neq \{v\}$; otherwise, $\mathbb{T}' = 0$.
- $\mathbb{T}'' = \mathbb{T} \vee \mathbb{T}'$ where $\mathbb{T}''.t = \max(\mathbb{T}.t, \mathbb{T}'.t), \mathbb{T}''.A = \mathbb{T}.A \cup \mathbb{T}'.A, \mathbb{T}''.V = \mathbb{T}.V \cup \mathbb{T}'.V$.
- $\mathbb{T}'' = \mathbb{T} \wedge \mathbb{T}'$ where $\mathbb{T}''.t = \min(\mathbb{T}.t, \mathbb{T}'.t), \mathbb{T}''.A = \mathbb{T}.A \cap \mathbb{T}'.A, \mathbb{T}''.V = \mathbb{T}.V \cap \mathbb{T}'.V$.

Definition 13 (Partial Order for Abstract Timers): We define the partial order $\leq$ on abstract timers such that for any two timers $\mathbb{T}, \mathbb{T}'$ for the same object/pointer and operation (prefetch or writeback):

$$\mathbb{T} \leq \mathbb{T}' \Leftrightarrow \mathbb{T}.t \leq \mathbb{T}'.t \text{ and } \mathbb{T}.V \subseteq \mathbb{T}'.V \text{ and } \mathbb{T}.A \subseteq \mathbb{T}'.A. \quad (23)$$

Definition 14 (Partial Order on DMA Abstraction): We define the partial order $\leq_D$ on the abstraction D such that for any two abstract states d, d':

$$d \leq_D d' \Leftrightarrow d.t \leq d'.t \text{ and } \forall \text{ timer } \mathbb{T} : d.\mathbb{T} + d.t \leq d'.\mathbb{T} + d'.t. \quad (24)$$

Since $\subseteq$ is a partial order on any set, and $\leq$ is a total order on $\mathbb{N}$, it is trivial to see that $\leq_D$ is also a partial order. Intuitively, d' is larger than d if and only if it has both a larger elapsed time, and a larger value of elapsed time plus timer for every object and pointer; following the example in Section VII, this implies that d' is guaranteed to cause a larger delay on successive basic blocks compared to d. We next show that $(D, \leq_D)$ is a complete lattice.

Lemma 15: $(D, \leq_D)$ is a complete lattice, where for any two abstract states d, d' with $t_{\max} = \max(d.t, d'.t)$ and $t_{\min} = \min(d.t, d'.t)$:

- for $d'' = d \vee_D d'$ it holds $d''.t = t_{\max}$ and for any timer $\mathbb{T} : d''.\mathbb{T} = \big(d.\mathbb{T} - (t_{\max} - d.t)\big) \vee \big(d'.\mathbb{T} - (t_{\max} - d'.t)\big)$;
- for $d'' = d \wedge_D d'$ it holds $d''.t = t_{\min}$ and for any timer $\mathbb{T} : d''.\mathbb{T} = \big(d.\mathbb{T} + (d.t - t_{\min})\big) \wedge \big(d'.\mathbb{T} + (d'.t - t_{\min})\big)$.

Proof: We formally prove that $(D, \leq_D)$ is a lattice, i.e., for any two elements d and d', $d \vee_D d'$ is the least upper bound to d, d' and $d \wedge_D d'$ is the greatest lower bound to d, d'; the completeness of the lattice (i.e., the fact that we can find a least upper bound and greatest lower bound for any subset Y of D) then follows from the completeness of the sets $\mathbb{N}$, $\mathscr{P}(\mathscr{A})$, $\mathscr{P}(\mathscr{V})$ used to represent times and address/object sets.

Consider $d'' = d \vee_D d'$. Since $d''.t = \max(d.t, d'.t)$, $d''.t$ is the smallest value that satisfies the partial order constraints $d.t \leq d''.t$ and $d'.t \leq d''.t$. Similarly, since $d''.\mathbb{T}.A = d.\mathbb{T}.A \cup d'.\mathbb{T}.A$, it is the smallest set that satisfies $d.\mathbb{T}.A \subseteq d''.\mathbb{T}.A$ and $d'.\mathbb{T}.A \subseteq d''.\mathbb{T}.A$; the same argument applies to $\mathbb{T}.V$. Next, assume without loss of generality that $d.t \leq d'.t$. Based on Definition 12 we then obtain: $d''.\mathbb{T}.t + d''.t = \max\big(d.\mathbb{T}.t - (t_{\max} - d.t), d'.\mathbb{T}.t - (t_{\max} - d'.t)\big) + d''.t = \max(d.\mathbb{T}.t - d'.t + d.t, d'.\mathbb{T}.t) + d'.t = \max(d.\mathbb{T}.t + d.t, d'.\mathbb{T}.t + d'.t)$; hence, $d''.\mathbb{T}.t$ is the smallest value that satisfies the partial order constraint for timer trailing length (Equation 24), concluding the proof for the least upper bound.

We omit the proof for the greatest lower bound as it is specular to the least upper bound. ∎

Note that the operator $d \vee_D d'$ computes the same upper bound as in Equations 19, 20.

Definition 16 (Generation of DMA Operations): Given an abstract state d and a DMA operation for object v in the allocation queue for a program state $\mathbf{s}$, we say that a timer $d.\mathbb{T}$ can *generate* the operation if it is of the same type (prefetch or write-back) as the timer and its trailing length is less than or equal to $d.\mathbb{T}.t$; additionally, v must be contained in $d.\mathbb{T}.V$; finally, for a write-back timer, the SPM address of the operation must be contained in $d.\mathbb{T}.A$.

Observation 17: By definition, if a timer $\mathbb{T}$ can generate a DMA operation, then any timer $\mathbb{T}' : \mathbb{T} \leq \mathbb{T}'$ can also generate that operation.

Definition 18 (Concretization Function): Given any abstract state d, the concrete state $S = \gamma(d)$ is the set of all feasible program states $\mathbf{s}$ for which:

- the elapsed time t since the beginning of the program is less than or equal to $d.t$; let $\Delta = d.t - t$;
- for any DMA operation in the allocation queue with trailing length greater than Δ, there is at least one timer $d.\mathbb{T}$ such that $d.\mathbb{T} + \Delta$ can generate the operation.

Definitions 16, 18 are key to understand how the abstraction works. In essence, the key idea is that adding Δ units of time to the elapsed time is always worse than increasing the trailing lengths of timers by the same amount Δ. Hence, if the difference between elapsed times for the abstract and program state is Δ, the program state can contain any DMA operation with trailing length up to Δ; while for operations with larger trailing length $k > \Delta$, a timer of the correct type/address/points-to set is required with $k \leq d.\mathbb{T}.t + \Delta$.

Lemma 19: The DMA Abstraction D is a valid abstraction for $\mathscr{P}(\Sigma)$.

Proof: We first show that Equation 21 holds. Given a concrete state S, we construct the abstract state d such that $d.t$ is an upper bound to the elapsed time of any program state $\mathbf{s} \in S$, and for any object v: $d.\mathbb{T}_v^{pr}.t$ is an upper bound to the trailing length of any prefetch operation for v in $\mathbf{s}$; $d.\mathbb{T}_v^{wb}.t$ is an an upper bound to the trailing length and $d.\mathbb{T}_v^{wb}.A$ is the union of the SPM addresses of any write-back operation for v in $\mathbf{s}$. It then immediately follows that for any $\mathbf{s} \in S$, the elapsed time for $\mathbf{s}$ is less than or equal to $d.t$ and every DMA operation is generated by a timer in d; hence, based on Definition 18, we have $\mathbf{s} \in \gamma(d)$ and thus $S \subseteq \gamma(d)$.

It remains to show that γ is monotone. Consider two abstract states $d \leq_D d'$; we have to show that $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s} \in \gamma(d')$. Let t be the elapsed time of $\mathbf{s}$; then it must hold $t \leq d.t \leq d'.t$. Define $\Delta = d.t - t$; since $\Delta \geq 0$, based on Definition 14 it must hold for any timer: $d.\mathbb{T} + \Delta \leq d'.\mathbb{T} + \Delta + (d.t' - d.t)$. Hence, if an operation of $\mathbf{s}$ can be generated by timer $d.\mathbb{T} + \Delta$, it can also be generated by timer $d'.\mathbb{T} + \Delta + (d.t' - d.t)$. This concludes the proof. ∎

It now remains to define the abstract transfer function $\hat{\mathscr{T}}_{e,\sigma}$, and prove Equation 22. We start by defining a set of helper functions. For simplicity of notation, we will consider three-valued logic variables which can assume one of the following values: {True, False, Unknown}. In particular, for each ALLOC/DEALLOC command on an object/pointer x we define an *exec* flag with the following meaning: if *exec* = True, then the value of the USERS field for the object pointed to by x is guaranteed to be 0 before an ALLOC and 1 before a DEALLOC; this implies that the corresponding command is effectively executed. If instead *exec* = False, USERS is guaranteed to be greater than 0/1 for an ALLOC/DEALLOC; hence, the command does not cause any state change. Finally, if *exec* = Unknown, then no assumptions on the value of the USERS can be made. In our approach, the *exec* flags are statically computed by the allocation algorithm: for a given allocation, if there is no enclosing allocation (in an ancestor region) on the same object, then *exec* = True. If there is an enclosing allocation which is guaranteed to be on the same object, then *exec* = False. Otherwise, *exec* = Unknown; note this case is required to handle pointers where the value of USERS can only be determine at run-time.

Definition 20 (ALLOC function): The function $d' = ALLOC(d, x, a, BB, pr, exec)$, where x is an object or pointer, a an address, BB a basic block, pr a binary flag and *exec*

a three-valued flag, modifies the abstract state d into d' by performing the following steps:

1) if $exec =$ True and $d.\mathbb{T}_x^{wb}.A = \{a\}$ and x points to a single object v in BB, then $d'.\mathbb{T}_x^{wb} = d.\mathbb{T}_x^{wb} \setminus v$;
2) then if $pr = 1$ and $exec \neq$ False, $d'.\mathbb{T}_x^{pr}.t$ is set to the maximum trailing length of any timer plus t_{dma}^x and $d'.\mathbb{T}_x^{pr}.V$ is the union of $d.\mathbb{T}_x^{pr}.V$ and the points-to list of x in BB.

Definition 21 (DEALLOC function): The function $d' = DEALLOC(d,x,a,BB,wb)$, where wb is a binary flag, modifies the abstract state d into d' by performing the following steps:

1) if $exec =$ True and x points to a single object v in BB, then $d'.\mathbb{T}_x^{pr} = d.\mathbb{T}_x^{pr} \setminus v$;
2) then if $wb = 1$ and $exec \neq$ False, $d'.\mathbb{T}_x^{wb}.t$ is set to the maximum trailing length of any timer plus t_{dma}^x; $d'.\mathbb{T}_x^{wb}.A = d.\mathbb{T}_x^{wb}.A \cup \{a\}$; and $d'.\mathbb{T}_x^{wb}.V$ is the union of $d.\mathbb{T}_x^{wb}.V$ and the points-to list of x in BB.

Functions *ALLOC* and *DEALLOC* are applied every time an ALLOC or DEALLOC command is encountered in a basic block. Based on the discussion in Section V-C, the ALLOC command is guaranteed to cancel a write-back operation on the same object if the two allocations target the same address in the SPM. This is performed in the *ALLOC* function by checking that the address of the write-back timer coincides with the address of the ALLOC, and removing the pointed-to object from the points-to set of the write-back timer. Note that for an object timer, this is equivalent to resetting the timer to 0, since by definition every object points to itself only; however, for a pointer we can do so only if there is no ambiguity in the points-to list (i.e., the pointer points to a single object in b). Then, if $pr = 1$, meaning that a prefetch operation must be scheduled, the function intuitively "appends" a new operation of length t_{dma}^x to the end of the allocation queue by setting the prefetch timer to the maximum trailing length in the queue plus t_{dma}^x. The behavior of the *DEALLOC* function is equivalent. Finally, all steps are dependent on the value of *exec*: to conservatively capture the worst case, we add a timer if the command could be executed ($exec =$ True or Unknown), but we remove a timer only if we are certain that the command is executed ($exec =$ True).

Definition 22 (ELAPSE function): The function $d' = ELAPSE(d,\Delta,\Lambda)$, with $\Delta,\Lambda \in \mathbb{N}$, modifies the abstract state d into d' such that: $d'.t = d.t + \Delta$ and $\forall$ timer $\mathbb{T}$: $d'.\mathbb{T} = d.\mathbb{T} - \Lambda$.

Intuitively, the function *ELAPSE* is used to increment time: the elapsed time is increased by Δ and every abstract timer is decreased by an amount Λ. Note that $\Lambda \leq \Delta$, since the DMA unit is stalled while the CPU accesses main memory.

Definition 23 (GETADDR stall): Given an abstract state d, we say that a GETADDR command on object/pointer x in basic block BB *stalls* on a timer $d.\mathbb{T}$ iff the intersection of the points-to list of x in BB and $d.\mathbb{T}.V$ is not empty.

Definition 24 (Depending ALLOC/DEALLOC): We say that an ALLOC/DEALLOC command for object/pointer x in basic block BB *depends* on a GETADDR command for object/pointer y in the same basic block iff the intersection of the points-to lists of x and y in BB is not empty.

Intuitively, if a GETADDR stalls on a timer, then in the worst case we need to wait until that timer elapses before the GETADDR can proceed. Similarly, if an ALLOC/DEALLOC depends on GETADDR, then in the worst case the GETADDR will stall on any DMA operation added by the ALLOC/DEALLOC.

Definition 25 (Abstract Transfer Function): Consider a CFG edge $e : BB \rightarrow BB'$. Let the execution for BB along e be divided into a set of consecutive *intervals*, such that the set of intervals cover all executed instructions but any change to the state of the SPM controller (including stalling the core due to a blocking command) only happens between one interval and the next. Then abstract transfer function $\hat{\mathscr{T}}_{e,\sigma}(d)$ is computed by applying an iterative set of transformations of the abstract state d using functions *ELAPSE*, *ALLOC*, *DEALLOC* based on the order of intervals, ALLOC, DEALLOC and GETADDR commands in BB:

- For each interval, let t_{comp}, t_{mm} and t_{spm} be the maximum computation time, main memory and SPM time for the interval, assuming that all load/stores to any object v_i (pointer p_k) access main memory iff $spm_{r_j}^{v_i} = 0$ (respectively, $spm_{r_j}^{p_k} = 0$) for all regions that contain BB. Then transform the state into $ELAPSE(d, t_{comp}^{BB} + t_{mm}^{BB} + t_{spm}^{BB}, t_{comp}^{BB} + t_{spm}^{BB})$.
- For a GETADDR command on object/pointer x, transform the state into $ELAPSE(d,\Delta,\Delta)$, where Δ is the maximum trailing length of any timer on which the GETADDR stalls.
- For an ALLOC command on object/pointer x, transform the state into $ALLOC(d,x,a,BB,pr,exec)$, where a is the SPM address of the ALLOC and $pr = 1$ if the P flag is set.
- For a DEALLOC command on object/pointer x, transform the state into $DEALLOC(d,x,a,BB,wb,exec)$, where a is the SPM address of the DEALLOC and $wb = 1$ if the W flag is set in the ALLOC command corresponding to this DEALLOC.

Note that in Definition 25, the execution of the code within the basic block is modeled by advancing elapsed time by the maximum execution time $t_{comp} + t_{mm} + t_{spm}$ and decreasing all timers by $t_{comp} + t_{spm}$, which is the time that DMA operations can proceed in parallel with the CPU assuming a dual-ported SPM. If the SPM is single-ported, we amend the definition to instead decrease the timers by t_{comp} only.

We are now ready to prove our main Theorem 31, which shows that Equation 22 holds for the described abstraction, hence concluding our proof obligations. Due to its complexity, we first present the intuition behind the proof and introduce several supporting lemmas. We first prove that the equation holds assuming that the order of ALLOC/DEALLOC/GETADDR commands and other instructions in the basic block is known. Intuitively, we construct a chain of abstract and program states, starting at the beginning of the basic block until its end; each successive pairs of states $d^i, \mathbf{s}^i$ and $d^{i+1}, \mathbf{s}^{i+1}$ represent the state changes caused by the execution of an SPM commands, or time elapsed executing instructions. In particular, in Lemmas 27-30 we prove that at each step in the chain $\mathbf{s}^i \in \gamma(d^i) \Rightarrow \mathbf{s}^i \in \gamma(d^i)$; this ensures that the abstract state always remains more general than the concrete state, as required in Equation 22.

Lemma 26: Consider a DEALLOC command in basic block BB for object/pointer x, and let v be the object pointed to by x in BB. If for v it holds $USERS = 1$ before executing the DEALLOC, then the WB flag for v is equal to the W flag for the corresponding ALLOC command.

Proof: Since allocations for objects/pointers that might point to the same object must be fully nested, if $USERS = 1$ before the DEALLOC, then it must have hold $USERS = 0$ before the corresponding ALLOC; hence, the value of the WB flag after the ALLOC command is equal to the W flag. Furthermore, any nested allocation on the same object

v cannot modify the WB flag, given that after the original ALLOC it holds $USERS = 1$ for v. Hence, the value of the WB flag before the DEALLOC must still be equal to the W flag for the corresponding ALLOC. ■

Lemma 27: Let $\mathbf{s}$ be the program state after the instruction(s) for an ALLOC command has been decoded and processed, but before any change to the state of the SPM controller is made, and let $\mathbf{s}'$ be the state after the changes (if any). Furthermore, let d' be computed based on abstract state d according to Definition 20, where $x, a, BB, pr, exec$ are determined based on the ALLOC command. Then $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d')$.

Proof: By definition, no instruction is processed between $\mathbf{s}, \mathbf{s}'$, hence no time elapses and $\mathbf{s}.t = \mathbf{s}'.t$. Furthermore by Definition 20, we have $d.t = d'.t$; hence, $\Delta = d.t - \mathbf{s}.t = \Delta' = d.t - \mathbf{s}'.t$. Therefore, to show $\mathbf{s}' \in \gamma(d')$, we only need to prove that the timers in d' generate all DMA operations in $\mathbf{s}'$ with trailing length greater than $\Delta = \Delta'$. Hence, consider changes to the list of DMA operations between $\mathbf{s}$ and $\mathbf{s}'$ and to the values of timers between d and d'. If a DMA operation is removed, then all DMA operations in $\mathbf{s}'$ must also be in $\mathbf{s}$, except that operations in $\mathbf{s}'$ might have smaller trailing length (if the removed operation was ahead in the queue). Hence, they can still be generated by the abstract state. Similarly, if a timer $\mathbb{T}$ is changed such that $d.\mathbb{T} \leq d'.\mathbb{T}$, then all operations generated by $\mathbb{T}$ in $\mathbf{s}$ can also be generated in $\mathbf{s}'$ (Observation 17). In summary, we only need to prove that the inclusion $\mathbf{s}' \in \gamma(d')$ is maintained for the following two changes to the program and abstract state: a DMA operation is added, or a timer $\mathbb{T}$ is changed and $d.\mathbb{T} \not\leq d'.\mathbb{T}$; we call the second case a *timer removal*.

Timer removal: Note that for step 2 in Definition 20, it holds $d.\mathbb{T} \leq d'.\mathbb{T}$ by construction. Hence, we only consider step 1, where $d'.\mathbb{T}^{wb}_x = d.\mathbb{T}^{wb}_x \setminus v$ if $exec =$ True and $d.\mathbb{T}^{wb}_x.A = \{a\}$ and x points to a single object v in BB. To prove that the inclusion $\mathbf{s}' \in \gamma(d')$ is maintained, we show that any DMA operation on object v generated by $d.\mathbb{T}^{wb}_x$ in $\mathbf{s}$ must be removed in $\mathbf{s}'$. By assumption, any such operation must be a write-back at the same address a as the ALLOC, the ALLOC command is for the same object v as the operation, and the command is executed ($exec =$ True); hence, based on Section V the ALLOC command will indeed cancel the DMA operation.

Operation insertion: Assume that a prefetch operation for v is inserted in the allocation queue (potentially after canceling a write-back). Based on Section V, the following must then be true: the P flag is set, $USERS = 0$ for v before the ALLOC, and the points-to list of x in BB must include v. This implies $pr = 1$ and $exec \neq$ False, hence, $\mathbb{T}^{pr}_x$ is modified in step 2 of Definition 22. Now let k be the maximum trailing length of any DMA operation in $\mathbf{s}$ before the write-back removal (if any), and K be the maximum trailing length of any timer in d. Based on Definition 18, it must hold: $k \leq K + \Delta$. Similarly, let $\bar{k}, \bar{K}$ be the maximum trailing lengths after the write-back removal: based on the previous timer removal case, if a timer is reset in the abstract state, then the corresponding operation is removed from the program state, thus it also holds $\bar{k} \leq \bar{K} + \Delta$. The trailing length of the appended prefetch operation for v in $\mathbf{s}'$ is then $\bar{k} + t^v_{dma}$, while based on Definition 20 for d' we set the timer $d'.\mathbb{T}^{pr}_x.t = \bar{K} + t^x_{dma}$, where $d'.\mathbb{T}^{pr}_x.V$ is union of $d'.\mathbb{T}^{pr}_x.V$ and the points-to set for x. Since x can point to v, then $t^x_{dma} \geq t^v_{dma}$, implying $\bar{k} + t^v_{dma} \leq \bar{K} + \Delta + t^x_{dma} = d'.\mathbb{T}^{pr}_x.t + \Delta'$. Hence, the added prefetch operation in $\mathbf{s}'$ is generated by $d'.\mathbb{T}^{pr}_x$, concluding the proof. ■

Lemma 28: Let $\mathbf{s}$ be the program state after the instruction(s) for a DEALLOC command has been decoded and processed, but before any change to the state of the SPM controller is made, and let $\mathbf{s}'$ be the state after the changes (if any). Furthermore, let d' be computed based on abstract state d according to Definition 21, where $x, a, BB, wb, exec$ are determined based on the DEALLOC command. Then $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d')$.

Proof: Similarly to the proof of Lemma 27, we have $\Delta = d.t - \mathbf{s}.t = \Delta' = d.t - \mathbf{s}'.t$ and we only need to prove that the inclusion $\mathbf{s}' \in \gamma(d')$ is maintained for any DMA operation insertion and timer removal.

As in Lemma 27, a timer $\mathbb{T}^{pr}_x$ can only be removed in step 1; but since $exec =$ True and x points to a single object v in BB, this guarantees that any DMA operation on object v generated by $d.\mathbb{T}^{pr}_x$ in $\mathbf{s}$ must be removed in $\mathbf{s}'$. The only operation that can be inserted is a write-back in step 2. Assuming the operation is for object v, it must hold that before the DEALLOC, the WB flag for v is set, $USERS = 1$ and x points to v. Based on Lemma 26, this implies that the P flag for the corresponding ALLOC is set, hence $wb = 1$ in Definition 21. Following the same reasoning as in Lemma 27, it then follows that the added write-back operation in $\mathbf{s}'$ is generated by $d'.\mathbb{T}^{pr}_x$. ■

Lemma 29: Let $\mathbf{s}$ be the program state after the instruction(s) for a GETADDR command has been decoded and processed, but before any change to the state of the SPM controller (including stalling the CPU) is made, and let $\mathbf{s}'$ be the state after the changes (if any). Furthermore, let d' be computed based on abstract state d according to Definition 22, where $\Delta = \Lambda$ is the maximum trailing length of any timer in d on which the GETADDR stalls. Then $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d')$.

Proof: Let $\bar{\Delta}$ be the amount of time that the program stalls due to the GETADDR command. Then $\mathbf{s}'.t = \mathbf{s}.t + \bar{\Delta}$, any DMA operation with trailing length less than or equal to $\bar{\Delta}$ in $\mathbf{s}$ is removed from $\mathbf{s}'$, while all other operations have a trailing length reduced by $\bar{\Delta}$. Let also $K = d.t - \mathbf{s}.t \geq 0$. Based on the SPM controller behavior in Section V, $\bar{\Delta}$ is the maximum trailing length of any DMA operation that stalls the GETADDR. We consider two cases: 1) $\bar{\Delta} \leq K$; then, there might be no timer in d that generates the maximum length operation, hence we can only assert $\Delta \geq 0$. 2) $\bar{\Delta} > K$; then, there must a timer $\mathbb{T}$ in d such that $\bar{\Delta} \leq d.\mathbb{T}.t + K$. Since this timer can generate the operation, by definition GETADDR stalls on the timer. Hence, we have $\bar{\Delta} \leq \Delta + K$. Combining the two cases we obtain:

$$\Delta \geq (\bar{\Delta} - K)^+. \tag{25}$$

Now consider $K' = d'.t - \mathbf{s}'.t$; to prove the inclusion of $\mathbf{s}'$ in d', we have to show that K' is non-negative. Note that based on Definition 22, we have $d'.t = d.t + \Delta$, and for each timer: $d'.\mathbb{T} = d.\mathbb{T} - \Delta$. Hence, we obtain: $K' = d.t + \Delta - \mathbf{s}.t - \bar{\Delta} = K + \Delta - \bar{\Delta}$. Substituting Equation 25 then yields: $K' \geq K + (\bar{\Delta} - K)^+ - \bar{\Delta} \geq K + (\bar{\Delta} - K) - \bar{\Delta} = 0$. It then remains to prove that operations in $\mathbf{s}'$ can be generated by d'.

Therefore, consider any operation in $\mathbf{s}'$ with trailing length k' greater than K'; we have to prove that the operation is generated by a timer in d'. Let k be the trailing length of the operation in $\mathbf{s}$, then $k = k' + \bar{\Delta}$. We then obtain: $k = k' + \bar{\Delta} > K' + \bar{\Delta} = K + \Delta - \bar{\Delta} + \bar{\Delta} = K + \Delta \geq K$. Since $k > K$, then there must exist a timer $\mathbb{T}$ in d that generates the operation, with $k \leq d.\mathbb{T}.t + K$. Note this implies $d.\mathbb{T}.t \geq k - K = k' + \bar{\Delta} - (K' - \Delta + \bar{\Delta}) > K' + \bar{\Delta} - K' + \Delta - \bar{\Delta} = \Delta$; since $d.\mathbb{T}.t > \Delta$, it thus holds $d.\mathbb{T}.t = d'.\mathbb{T}.t + \Delta$ (i.e., timer $\mathbb{T}$ is not reset in d'). We then obtain: $k' = k - \bar{\Delta} \leq d.\mathbb{T}.t + K - \bar{\Delta} = (d'.\mathbb{T}.t + \Delta) + (K' - \Delta + \bar{\Delta}) - \bar{\Delta} = d'.\mathbb{T}.t + K'$. Therefore, the trailing length

of $d'.\mathbb{T}$ is sufficient to generate the DMA operation in $\mathbf{s}'$, completing the proof. ■

Lemma 30: Consider an interval of time where the program executes with no change to the state of the SPM controller (including stalling the CPU) during the interval, and let t_{comp}, t_{mm} and t_{spm} be the maximum computation time, main memory and SPM time for the interval, assuming that all load/stores to any object v_i (pointer p_k) access main memory iff $spm_{r_j}^{v_i} = 0$ (respectively, $spm_{r_j}^{p_k} = 0$) for all regions that contain the interval. Furthermore, let $\mathbf{s}$ be the program state at the beginning of the interval, $\mathbf{s}$ the state at the end of the interval, and d' be computed based on abstract state d according to Definition 22 with $\Delta = t_{comp} + t_{mm} + t_{spm}$ and $\Lambda = t_{comp} + t_{spm}$. If the latency for access to main memory is greater than or equal to the latency for access to the SPM, then $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d')$.

Proof: Let $\bar{t}_{comp}, \bar{t}_{mm}$ and $\bar{t}_{spm}$ denote the actual computation, main memory and SPM times for the interval, rather than the upper bounds. Then by definition we have $t_{comp} \geq \bar{t}_{comp}$ and $t_{mm} \geq \bar{t}_{mm}$. Note that for the SPM time it might hold $\bar{t}_{spm} \geq t_{spm}$, since some load/stores operations that are assumed to access main memory might access the SPM in the actual program execution; however, since memory latency is at least equal to SPM latency, it must still hold $t_{mm} + t_{spm} \geq \bar{t}_{mm} + \bar{t}_{spm}$. Now define $\bar{\Delta} = \bar{t}_{comp} + \bar{t}_{mm} + \bar{t}_{spm}$ and $\bar{\Lambda} = \bar{t}_{comp} + \bar{t}_{spm}$; note we must have $\bar{\Delta}, \bar{\Lambda} \geq 0$. Finally, let $\delta = \Delta - \bar{\Delta}$ and $\lambda = \bar{\Lambda} - \Lambda$. Note $\delta \geq 0$, and furthermore: $\delta + \lambda = \Delta - \Lambda - (\bar{\Delta} - \bar{\Lambda}) = t_{mm} - \bar{t}_{mm} \geq 0$.

By assumption on the behavior of the interval, $\mathbf{s}'.t = \mathbf{s}.t + \bar{\Delta}$, any DMA operation with trailing length less than or equal to $\bar{\Lambda}$ in $\mathbf{s}$ is removed from $\mathbf{s}'$, while all other operations have a trailing length reduced by $\bar{\Lambda}$. Also based on Definition 22: $d'.t = d.t + \Delta$. Let $K = d.t - \mathbf{s}.t \geq 0$, we then have $K' = d'.t - \mathbf{s}'.t = d.t + \Delta - \mathbf{s}.t - \bar{\Delta} + \delta = K + \delta$; thus $K' \geq K \geq 0$, and to satisfy the inclusion $\mathbf{s}' \in \gamma(d')$ it remains to show that operations in $\mathbf{s}'$ can be generated by d'.

Therefore, consider any operation in $\mathbf{s}'$ with trailing length k' greater than K'; we have to prove that the operation is generated by a timer in d'. The trailing length of that operation in $\mathbf{s}$ must be $k = k' + \bar{\Lambda} > K' \geq K$; hence, there must be a timer $\mathbb{T}$ in d that generates that operation, such that:

$$k \leq d.\mathbb{T}.t + K. \tag{26}$$

This implies $d.\mathbb{T}.t + K \geq k' + \bar{\Lambda} > K' + \Lambda + \lambda$, and thus $d.\mathbb{T}.t > (K' - K) + \Lambda + \lambda = \Lambda + \delta + \lambda \geq \Lambda$. Based on Definition 22, we have $d'.\mathbb{T}.t = d.\mathbb{T}.t - \Lambda$, and since $d.\mathbb{T}.t > \Lambda$, it thus holds $d.\mathbb{T}.t = d'.\mathbb{T}.t + \Lambda$ (i.e., timer $\mathbb{T}$ in not reset in d'). Substituting the expression for $d.\mathbb{T}.t$ in Equation 26 yields: $d'.\mathbb{T}.t + \Lambda + K = d'.\mathbb{T}.t + \Lambda + K' - \delta \geq k = k' + \Lambda + \lambda$, which is equivalent to: $d'.\mathbb{T}.t + K' \geq k' + \delta + \lambda \geq k'$. Therefore, the trailing length of $d'.\mathbb{T}$ is sufficient to generate the DMA operation in $\mathbf{s}'$, completing the proof. ■

Note that while we proved Lemma 30 for the dual-ported SPM case, the Lemma is also valid for the single-port case where $\Lambda = t_{comp}$, $\bar{\Lambda} = \bar{t}_{comp}$, since it still holds $\delta + \lambda = t_{mm} + t_{spm} - \bar{t}_{mm} - \bar{t}_{spm} \geq 0$.

Theorem 31: Equation 22 holds for the described DMA Abstraction $(D, \leq_D)$ with abstract transfer function $\hat{\mathscr{T}}_{e,\sigma}(d)$.

Proof: We need to show $S \subseteq \gamma(d) \Rightarrow \mathscr{T}'_{e,\sigma}(S) \subseteq \gamma(\hat{\mathscr{T}}_{e,\sigma}(d))$ for every edge $e : BB \to BB'$. Since by definition $\mathscr{T}'_{e,\sigma}(S) = \cup_{\mathbf{s} \in S} \mathscr{T}_{e,\sigma}(\mathbf{s})$, this is equivalent to showing that $\forall d \in D, \forall \mathbf{s} \in \gamma d$, if the execution can flow along edge e from state $\mathbf{s}$ with $d' = \hat{\mathscr{T}}_{e,\sigma}(d)$ and $\mathbf{s}' = \mathscr{T}_{e,\sigma}(\mathbf{s})$, it must hold: $\mathbf{s}' \in \gamma(d')$.

Since in Definition 25 we have described $\hat{\mathscr{T}}_{e,\sigma}(d)$ as an iterative transformation based on the scratchpad commands within basic block *BB*, we apply the same technique to $\mathscr{T}_{e,\sigma}$, and describe the transformation of the program state $\mathbf{s}$ based on a sequence of instruction intervals and ALLOC/DEALLOC/GETADDR commands. Note that since basic blocks in the extended CFG do not contain branches or function calls, every execution of *BB* along e has the same sequence of intervals/commands as the one considered by $\hat{\mathscr{T}}_{e,\sigma}(d)$.

Without loss of generality, let N be the total number of intervals and commands. Let us define a set of abstract states $\{d^0, \ldots, d^N\}$ and program states $\{\mathbf{s}^0, \ldots, \mathbf{s}^N\}$, where $d^0 = d, \mathbf{s}^0 = \mathbf{s}$ and for $0 < i \leq N$, d^i and $\mathbf{s}^i$ represent the abstract and program state after the N^{th} interval/command in the sequence. Then by definition: $d' = d^N, \mathbf{s}' = \mathbf{s}^N$. Now note that based on Lemma 30 for intervals and Lemmas 27, 28, 29 for commands, it holds $\mathbf{s}^{i-1} \in \gamma(d^{i-1}) \Rightarrow \mathbf{s}^i \in \gamma(d^i)$. Hence, by induction on i, it also holds: $\mathbf{s}^0 \in \gamma(d^0) \Rightarrow \mathbf{s}' = \mathbf{s}^N \in \gamma(d^N)$, concluding the proof. ■

Applying Definition 25 requires a precise knowledge of the position of each command in basic block *BB*. For simplicity of implementation, it can also be useful to formulate an analysis where the only available timing information are upper bounds to the computation, memory and SPM times t_{comp}^{BB}, t_{mm}^{BB} and t_{spm}^{BB} for the entire basic block, rather than individual intervals, and only the relevant ordering of SPM commands in the basic block is known.

Definition 32 (Imprecise Abstract Transfer Function): Consider a CFG edge $e : BB \to BB'$, and let t_{comp}^{BB}, t_{mm}^{BB}, t_{spm}^{BB} represent the maximum computation time, main memory and SPM time for basic block *BB*, assuming that all load/stores to object v_i (pointer p_k) access main memory iff $spm_{r_j}^{v_i} = 0$ (respectively, $spm_{r_j}^{p_k} = 0$) for all regions that contain *BB*. We can then compute an abstract transfer function $\hat{\mathscr{T}}_{e,\sigma}(d)$ by applying an iterative set of transformations of the abstract state d using functions *ELAPSE*, *ALLOC*, *DEALLOC* based on the order of ALLOC, DEALLOC and GETADDR commands in *BB*:

- Order the set of transformations as follows: first, apply transformations for each GETADDR command and each ALLOC/DEALLOC command that depends on a GETADDR or is followed by another ALLOC/DEALLOC that depends on a GETADDR, in the order in which the commands appear in *BB*; then, apply the transformation for *BB*'s execution time; then, apply transformations for each ALLOC/DEALLOC commands that has not been considered yet, in the order in which they appear in *BB*.
- For a GETADDR command on object/pointer x, transform the state into $ELAPSE(d, \Delta, \Delta)$, where Δ is the maximum trailing length of any timer on which the GETADDR stalls.
- For an ALLOC command on object/pointer x, transform the state into $ALLOC(d, x, a, BB, pr, exec)$, where a is the SPM address of the ALLOC and $pr = 1$ if the P flag is set.
- For a DEALLOC command on object/pointer x, transform the state into $DEALLOC(d, x, a, BB, wb, exec)$, where a is the SPM address of the DEALLOC and $wb = 1$ if the W flag is set in the ALLOC command corresponding to this DEALLOC.
- To transform the state based on *BB*'s execution time, apply function $ELAPSE(d, t_{comp}^{BB} + t_{mm}^{BB} + t_{spm}^{BB}, t_{comp}^{BB} + t_{spm}^{BB})$.

Intuitively, the imprecise transfer function works as follows: we assume that all ALLOC/DEALLOC commands that

do not depend on a GETADDR are "pushed" to the end of the basic block, since doing so adds prefetch and write-back operations at the last possible time, hence maximizing the blocking that can be suffered by following basic block. On the other hand, GETADDR commands (and depending ALLOC/DEALLOC) are "pulled" to the beginning of the basic block, since this maximizes the amount of blocking that the GETADDR suffers due to DMA operations started in preceding basic blocks.

Theorem 33: Equation 22 holds for the described DMA Abstraction $(D, \leq_D)$ with abstract transfer function $\hat{\mathscr{T}}_{e,\sigma}(d)$.

Proof: [Proof Sketch]

Consider $e : BB \to BB'$, and let $d' = \hat{\mathscr{T}}_{e,\sigma}(d)$ and $d'' = \tilde{\mathscr{T}}_{e,\sigma}(d)$. As in the proof of Theorem 31, we have to show that $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d'')$, where $\mathbf{s}'$ is the program state after the execution of *BB* along e starting from program state $\mathbf{s}$.

Next note that the only case in which commands can be reordered in Definition 32 is when an ALLOC/DEALLOC that does not depend on any GETADDR is pushed to the end of the basic block. By definition, the ALLOC/DEALLOC command cannot operate of any timer that are checked by a GETADDR; hence, reordering the commands in this way cannot change the behavior of the SPM controller. Therefore, it remains to argue that the following three changes will maintain the order $d' \leq_D d''$: 1) moving a GETADDR to the beginning of the basic block; 2) moving a non-dependent ALLOC/DEALLOC to the end of the basic block; 2) moving a dependent ALLOC/DEALLOC (or an ALLOC/DEALLOC followed by a dependent one) to the beginning of the basic block.

GETADDR. Let Δ be the blocking time of the GETADDR for the precise abstraction ($\hat{\mathscr{T}}_{e,\sigma}(d)$). Since in $\tilde{\mathscr{T}}_{e,\sigma}(d)$ the GETADDR is moved at the beginning of the interval, the trailing length of any timer on which GETADDR can stall must be greater than or equal to the trailing length in the precise abstraction; hence, $\tilde{\Delta} \geq \Delta$, where $\tilde{\Delta}$ is the blocking time for the imprecise abstraction. Following the same argument as in Lemma 29, we have $\tilde{\Delta} \geq \Delta \geq \bar{\Delta}$, where $\bar{\Delta}$ is the actual blocking time for $\mathbf{s}$, which then implies $\mathbf{s}' \in \gamma(d'')$.

Non-dependent ALLOC/DEALLOC. Note that moving an ALLOC/DEALLOC while keeping the same order of dependent commands does not change which timers are removed (if any). Hence, consider any timer $\mathbb{T}$ added by the ALLOC/DEALLOC. Since the command is moved to the end of the basic block, it must hold $d'.\mathbb{T}.t \leq d''.\mathbb{T}.t$, and hence $d' \leq_D d''$; since $\mathbf{s}' \in \gamma(d')$ by Theorem 31, then $\mathbf{s}' \in \gamma(d'')$ by monotonicity of γ.

Dependent ALLOC/DEALLOC. Any timer added by a dependent ALLOC/DEALLOC will by definition cause a GETADDR in *BB* to stall. Similarly, any ALLOC/DEALLOC followed by a dependent command will increase the maximum trailing length of any timer, hence increasing the trailing length of the dependent timers. Therefore, moving these commands to the beginning of the basic block immediately before the GETADDR cannot decrease the program stall time compared to the precise abstraction, meaning $\tilde{\Delta} \geq \Delta$ and $\mathbf{s}' \in \gamma(d'')$ from Lemma 29.

■

VIII. Implementation

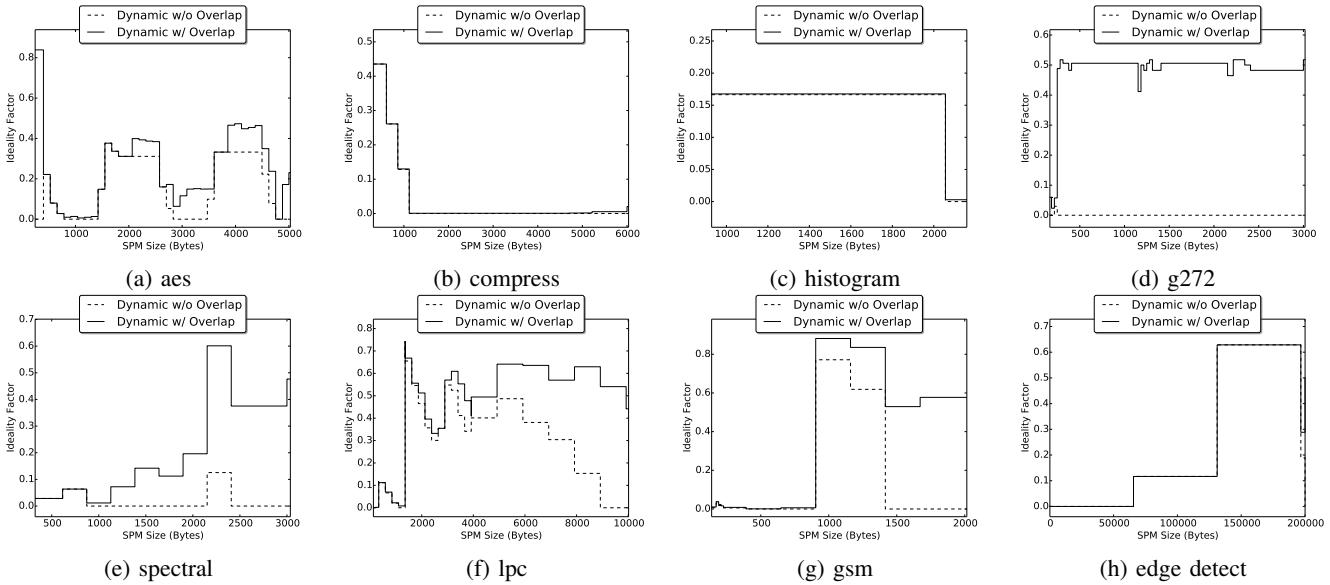
A compiler-integrated flow is used to implement the allocation algorithm. The flow analyzes the program, runs the allocation algorithm, applies the required transformations, and generates an executable. We integrated our flow with the open-source LLVM compiler [3]. The program is compiled to the optimized IR representation, then the allocation is implemented using the following passes.

- **Convert Stack Variables to Globals.** Each function frame on the stack has two components: a) temporary spilled registers and calling context; b) local objects. Allocating the full stack might be unfeasible if the maximum stack depth does not fit in the SPM. In order to allow a flexible allocation scheme for the stack, a pass is implemented to promote large local objects to global objects [12]. This reduces the maximum stack size, so that it can be considered an object in the allocation algorithm, and allows allocating local objects either in main memory or in the SPM without the need to manage multiple stacks.
- **Region Tree Generation.** We use the provided region analysis in LLVM to construct the refined region tree.
- **SPM allocation.** The allocation algorithm generates an optimized allocation solution. As discussed in Section VI-B, we compute the fitness of a feasible solution by analyzing the WCET. So, the code is transformed to insert the allocation commands and to modify the memory references and then the program is analyzed for WCET.
- **Code Transformation.** Transforming the code includes inserting allocation commands and modifying memory references. As each region is defined by two edges, we simply insert a new basic block with the allocation commands (**ALLOC/DEALLOC**) on this edge (entry/exit). Most of these basic blocks are optimized by the compiler and integrated with other basic blocks when possible. For **GETADDR** commands, we find the first instruction that references an object after an allocation/de-allocation and insert **GETADDR** before it. After that, all the references to the object until the next allocation/de-allocation are modified to the address returned by **GETADDR** command.
- **WCET Analysis.** The LLVM IR code is compiled to assembly code for the target processor. The execution time for each basic block is extracted from the program assembly based on the processor model. We use the information from the back-end to conduct the WCET analysis.
- **Code generation.** The assembly code for the final allocation is generated and a linker script that specifies the memory sections is used to produce the executable.

IX. Evaluation

The evaluation of the prefetching approach for the data SPM allocation is performed using a simple model of MIPS processor with a 5-stages pipeline and no branch predictor. For memory instructions, we consider a latency for a word access to main memory of 10 cycles, 1 cycle to SPM and 1 cycle to the SPM controller. For the DMA, we use a similar model as in [28] such that the latency to initialize the transfer to/from main memory is 10 cycles and the latency per word is 2 cycles.

We consider three cases: 1) static allocation only; 2) dynamic allocation without prefetching; 3) and dynamic allocation with prefetching. As discussed in Section VI-A, we applied our allocation algorithm for all three cases as it can serve as an alternative heuristic to the static and dynamic allocation approaches in the previous work, thus allowing a fair comparison. Note that the stack always resides in the SPM as its size becomes small after reducing its depth by the converting stack variables to globals as discussed in Section VIII and its access rate is usually high.



(a) aes (b) compress (c) histogram (d) g272

(e) spectral (f) lpc (g) gsm (h) edge detect

Fig. 10: Ideality factor

We tested the allocation algorithm for multiple benchmarks from UTDSP, MediaBench, and CHStone suites. We present 8 kernels from these suites. We avoided benchmarks that have the following criteria: 1) benchmarks with system calls, as we cannot analyze their WCET without the OS code; 2) benchmarks that access only the stack or have very small sizes for static and local objects.

We define the ideal case as the case of initially having all the data of the program in the SPM. Figure 10 shows the *ideality factor* as a function of the size of the SPM. The ideality factor is computed as $\big(WCET(static) - WCET(dynamic)\big)/\big(WCET(static) - WCET(ideal)\big)$, where the denominator represents the best hypothetical improvement in WCET relative to the static allocation and the numerator is the improvement for the dynamic case. The ideality factor represents how close the allocation is to the ideal case compared to the static allocation, with a value of 1 indicating a performance equivalent to the ideal case. We plot the ideality factor for two cases: w/o prefetching and w/ prefetching. For each benchmark, we vary the range of the SPM sizes starting from the size in which at least one object can fit in the SPM.

A. Results Analysis

As discussed in [5], the dynamic allocation without prefetching is more beneficial than the static allocation only for intermediate SPM sizes which can fit some but not all of the objects in the SPM. That is, for small sizes of the SPM where none of the objects can fit in the SPM and for large sizes where most of the objects can fit in the SPM, the benefit of dynamic and static allocation is similar without prefetching. For object-based approaches like our method, the range of the SPM sizes that shows benefit for the dynamic allocation is dependent on the number, sizes and live ranges of the objects of the program. The significance of dynamic allocation appears when there are multiple objects with distinct live ranges and the size of the SPM can fit some but not all of them.

Prefetching allows the allocation of objects with low access rate as it can reduce the transfer cost. When the size of the SPM is large enough to fit most of the objects, prefetching outperforms the static allocation in choosing the memory transfer points to minimize the transfer cost. For intermediate SPM sizes where dynamic allocation is useful, prefetching can still offer additional benefit by hiding the transfer cost when there are opportunities to overlap the memory transfers.

Histogram is an example of a program that does not provide space for optimization as it has two main arrays with similar size whose live ranges interfere. So, the dynamic allocation does not have the flexibility to reallocate the objects during run-time. The prefetching approach also does not gain much as the objects are used at the beginning of the program. For compress and edge-detect, we are able dynamically reallocate the objects. However, the program does not offer much overlap because the objects are used inside a loop or because the objects are very large and the program does not have enough overlap. The results for benchmarks aes, g272, spectral, lpc, and gsm show that the prefetching approach excels in the ranges where there is dynamic allocation and there are chances to overlap the memory transfers. When the static allocation is similar to the dynamic allocation, the prefetching approach can still achieve significant gains as it can overlap the transfer until the first use of the object.

The solving time for the allocation algorithm depends on the number of possible allocations, the size of the CFG of the program and the genetic algorithm parameters. In the experiments, we used a population of 100 chromosomes and termination parameters $k = 500, n = 10$. The solving time varied between few seconds to around 15 minutes. Inserting the allocation commands increases the executable code size by at most 1.2% for the tested programs.

X. Conclusions

In this paper, we introduced a framework for predictable data SPM prefetching. Our approach is automated within a compilation flow that is integrated with LLVM compiler. We provided a hardware/software design that includes an SPM controller, an allocation algorithm and a WCET analysis.

The experiments have shown the potential of our prefetching technique to provide a predictable mechanism to hide the latency of main memory transfers and efficiently manage the data SPM with low overhead.

Our framework can be extended to handle pointer-based memory accesses for static, stack and dynamically allocated objects. The performance of the allocation algorithm can be enhanced to tackle large objects and loops using transformations like tiling and data pipelining. We plan to integrate these mechanisms in our framework in future work.

REFERENCES

[1] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 35:1–35:35, Aug. 2016.

[2] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 32:1–32:36, Nov. 2015.

[3] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[4] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 3, pp. 21:1–21:32, Apr. 2009.

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

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

[7] Y. Yang, M. Wang, Z. Shao, and M. Guo, "Dynamic scratch-pad memory management with data pipelining for embedded systems," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 2, Aug 2009, pp. 358–365.

[8] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis, "A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 3, pp. 279–291, March 2006.

[9] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *J. Embedded Comput.*, vol. 1, no. 4, pp. 521–540, Dec. 2005.

[10] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Wcet centric data allocation to scratchpad memory," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, Dec 2005, pp. 10 pp.–232.

[11] J. Whitham and N. Audsley, "Studying the applicability of the scratch-pad memory management unit," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2010, pp. 205–214.

[12] S. Kim, "Using scratchpad memory for stack data in hard real-time embedded systems," in *Proceedings of the Memory Architecture and Organization Workshop*, 2011.

[13] J.-F. Deverge and I. Puaut, "Wcet-directed dynamic scratchpad memory allocation of data," in *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, ser. ECRTS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 179–190.

[14] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, "An integrated hardware/software approach for run-time scratchpad management," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 238–243.

[15] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 269–279.

[16] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric os for multi-core embedded systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–11.

[17] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on*, Oct 2015, pp. 1–8.

[18] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Memory-processor co-scheduling in fixed priority systems," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, ser. RTNS '15. New York, NY, USA: ACM, 2015, pp. 87–96.

[19] A. Alhammad, S. Wasly, and R. Pellizzoni, "Memory efficient global scheduling of real-time tasks," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2015, pp. 285–296.

[20] R. Mancuso, R. Dudko, and M. Caccamo, "Light-PREM: Automated software refactoring for predictable execution on cots embedded systems," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2014, pp. 1–10.

[21] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 171–185.

[22] J. Vanhatalo, H. Völzer, and J. Koehler, "The refined process structure tree," in *Proceedings of the 6th International Conference on Business Process Management*, ser. BPM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 100–115.

[23] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, pp. 1–69, Apr. 2015.

[24] R. Wilhelm *et al.*, "The worst-case execution-time problem : Overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[25] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[26] S. Thesing, "Safe and precise wcet determination by abstract interpretation of pipeline models," Ph.D. dissertation, Universitt des Saarlandes, Postfach 151141, 66041 Saarbrcken, 2004.

[27] T. Lundqvist, "A wcet analysis method for pipelined microprocessors with cache memories," Ph.D. dissertation, School of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2002.

[28] X. Yang, L. Wang, J. Xue, T. Tang, X. Ren, and S. Ye, "Improving scratchpad allocation with demand-driven data tiling," in *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '10. New York, NY, USA: ACM, 2010, pp. 127–136.