WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching*

Muhammad R. Soliman[1] **and Rodolfo Pellizzoni**[2]

1 **University of Waterloo, Waterloo, ON, Canada**
`mrefaat@uwaterloo.ca`
2 **University of Waterloo, Waterloo, ON, Canada**
`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 WCET bounds, such techniques generally assume that the processor is stalled while waiting to reload the content of the 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.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases scratchpad, LLVM, prefetching, real-time, genetic algorithm

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.24

1 Introduction

The performance of computer programs can be significantly affected by main memory latency, which has largely remained similar in recent years [18]. As a consequence, cache prefetching has been extensively researched in the architecture community [16]. 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. 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, there has been significant attention to the management of on-chip memory in recent times. 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 [9]. These techniques allow the derivation of tighter WCET

* This work was supported in part by NSERC DG 402369-2011 and CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

bounds by statically determining if a memory instruction will access the on-chip memory or the 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 memory 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 the data is used in the program, and we prefetch it into the 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 efficiently 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. For simplicity and as a proof of concept, we implement our mechanism using a dedicated SPM controller, but we argue that a similar scheme could be supported by other platforms with the required DMA functionality. 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 SPM 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 [12], 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 2. We then introduce a motivating example in Section 3. We detail the region-based program representation in Section 4, and our proposed allocation mechanism in Section 5. Section 6 discusses the allocation algorithm, and Section 7 introduces the WCET abstraction for our prefetch mechanism. Finally, we present the compiler implementation in Section 8 and experimental results in Section 9, and provide concluding remarks in Section 10.

2 Related Work

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 [17, 24, 2, 29, 5, 7]. Other mechanisms optimize the allocation for WCET in real-time systems [21, 26, 11, 6]. In general, management techniques are divided between static or dynamic. Static methods partition the data memory between SPM and main memory with fixed allocation of the SPM at compile-time [2, 21]. On the other hand, dynamic methods adapt to the changing working data set of the program by moving objects between SPM and main memory during run-time [17, 24, 6, 7, 26, 29]. Since our proposed scheme allows us to more efficiently hide the cost of data transfers, we focus on dynamic allocation.

The closest related work in the scope of dynamic methods for data SPM are [29, 5, 8], which apply prefetching through DMA. In [29], the authors proposed a data pipelining technique for SPM that utilizes DMA to achieve data parallelization for multiple iterations of a loop based on the iteration access patterns of arrays. The work in [5] 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 the energy and maximize average performance. In [8], the authors add a dedicated 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) [19, 22, 3]. Under PREM, the data and code of a task are fetched into on-chip memory before execution, preferably using DMA. A variety of co-scheduling schemes (see for example [15, 1]) 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 [14], the proposed approaches assume manual code modification, which we find unrealistic in practice. An automated compiler tool-chain is described in [14], but since it relies on profiling, it cannot guarantee WCET bounds.

3 Motivating Example

In this section, we present an example that shows the benefit of data prefetching in SPM-based systems. Given a set of data *objects* used by a program, the general SPM 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/modified 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 of 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 be 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

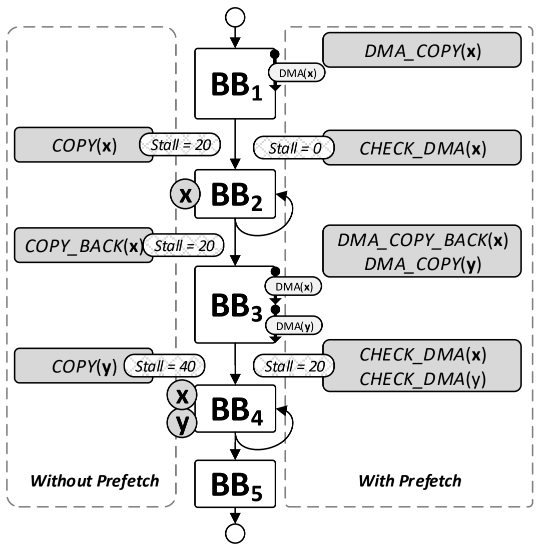


Figure 1 Motivating Example.

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 40 cycles (fetch only as y is not modified, so there is no need to write it back to main memory). The optimal allocation would choose x as it has a net benefit of 70 cycles versus 30 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 resulting in total cost of $20 + 20 + 40 = 80$ cycles. 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 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.

4 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* [10] 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 $G_f = (N_f, E_f)$ be the CFG for function f, where N_f is the set of nodes representing basic blocks and E_f 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.

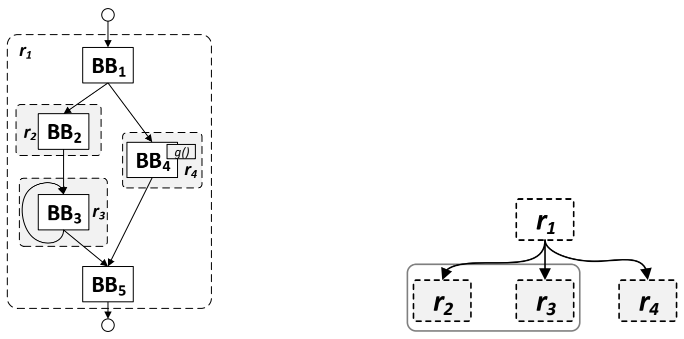
In the rest of the paper, we use the term *allocation* to refer to the act of reserving a space for an object in the SPM at a given address during the execution of the program code. In our solution, we restrict the allocation of objects on a per-region basis: space for an object is reserved upon entering 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.

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

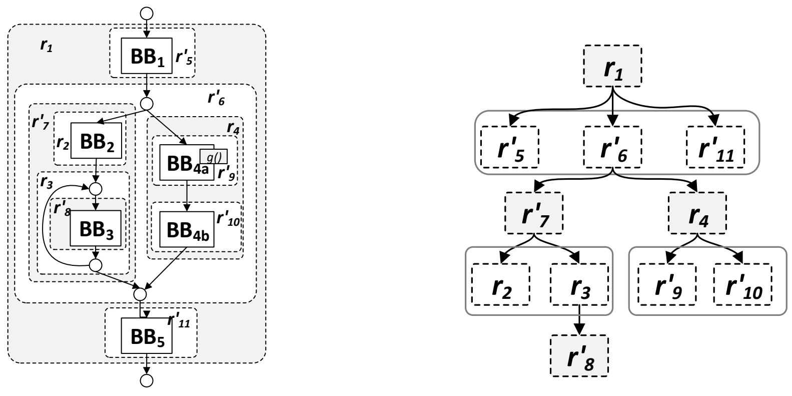
- Each call to a function in G_f is split into a separate *call node.*
- A *merge/split node* is inserted before/after a basic block/call node with multiple entry/exit edges.

Note that after the transformation, every node in $\bar{G}_f$ 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



(a) Program CFG **(b)** Region tree

Figure 2 Program CFG G_f and region tree.



(a) Refined program CFG **(b)** Refined region tree

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

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 [25] that adding maximal sequential regions to the tree still results in a unique 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 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. Regions r_1, r'_7 and r_4 are sequential regions. We refer to r'_9 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.

5 Allocation Mechanism

We now present our proposed allocation mechanism in detail. 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. Data pipelining techniques for loops and field-based allocation for data structures 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 consider global and stack objects only for allocation. We rely on pointer analysis to determine the references of the load/store instructions. For stack objects, we convert large local objects to global objects before allocation as we discuss in Section 8. While our method can be extended to handle dynamic pointers as shown in technical report [20], we do not discuss it here due to space constraints.
- For simplicity, we consider a system comprising a single core running one program with no preemption. 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 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 4, 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.

[1] For example, the Freescale MPC5777M SoC used in previous work [22] includes both SPM memory and a dedicated I/O processor that could be used to implement the described management functionalities.

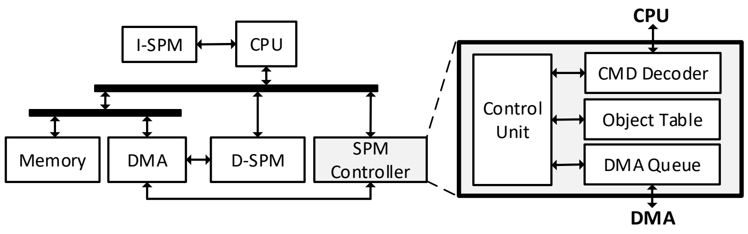


Figure 4 SPM-based System.

5.1 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. Note that we assume physical memory addresses in this implementation, *i.e.*, no virtual address mapping is used. 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 parallelization 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.

The SPM controller consists of *command decoder*, *object 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 and the DMA queue.

The object table tracks the state of the program objects. Note that only the subset of program objects that can be allocated in the SPM are tracked by the object table. An object table with 32 entries is sufficient in our tests. However, if the number of objects in the program exceeds the number of entries in the object table, the object table can hold only the allocated objects at each program point. An entry in the object table contains the main memory address, the size of the object, the SPM address and allocation flags that reflect 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 the object when de-allocated if it is used
`U`	(U)sed in the SPM
`USERS`	number of current users of the object

The `USERS` field records the number of allocations that have issued an `ALLOC` command for the object and are still using the object in the SPM, *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 field in Section 5.3. The DMA is configured with source address, size and destination address extracted from an entry in the object table. DMA operations are added to the DMA queue that allows scheduling multiple memory transfers and executing them in FIFO order.

5.2 Allocation Commands

`ALLOC` command reserves the space in the SPM and schedules a DMA transfer if necessary. The command has the following syntax: `ALLOC.XX (TBL_IDX, MEM_ADDR)`, where `TBL_IDX` is the table index for the object and `MEM_ADDR` is the allocation address in the SPM. There are four versions of `ALLOC.XX` command according to the directives `XX`: `ALLOC`, `ALLOC.P`, `ALLOC.W`, `ALLOC.PW`. The `P` directive directs the controller to prefetch the object from main memory. The SPM controller will schedule a prefetch transfer for the object and set `PF_OP` flag in the object entry. If the `P` directive is not used, the object is allocated directly and flag `A` is set. Otherwise, flag `A` is set once the prefetch transfer completes. The `W` directive sets the `WB` flag in the object entry which directs the controller to copy back the object to the main memory when de-allocated. The `P` directive 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 modified, *e.g.* writing some elements of an array. The `W` directive 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.

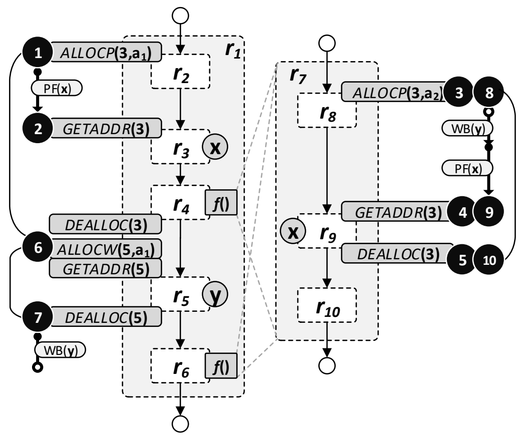
`DEALLOC` command de-allocates the object in table index `TBL_IDX` from the SPM: `DEALLOC (TBL_IDX)`. If the `WB` and `U` flags are set in the object entry at `TBL_IDX`, the controller will schedule a write-back transfer, set `WB_OP` flag and reset `A` flag. Otherwise, the object will be de-allocated by simply resetting `A` flag.

`GETADDR` command returns the current address of the object associated with entry `TBL_IDX` in the object table: `GETADDR (TBL_IDX)`. If `PF_OP` or `WB_OP` flag is set in the table index `TBL_IDX`, the controller stalls until the DMA completes transferring the object. If no transfer is scheduled or after the transfer finishes, the controller returns the SPM address if `A` flag is set and the main memory address 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 [26]; hence, we do not add extra overhead to the critical path of the processor.

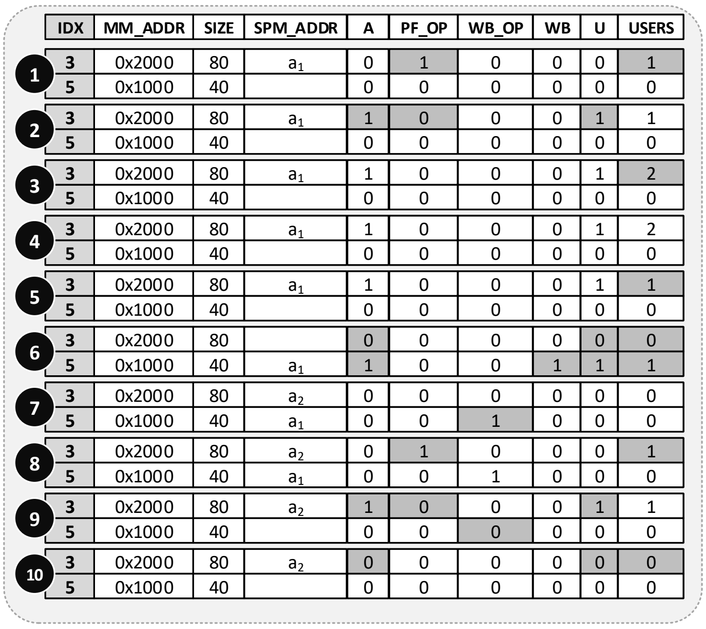
If a prefetch transfer has been scheduled and a `DEALLOC` command is issued for the object to be prefetched, the transfer is canceled as the object is not needed anymore. Also, if a write-back transfer has been scheduled for an object and it was followed by `ALLOC.XX` for the same object, the transfer 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.

5.3 Example

Figure 5 shows an allocation example for two objects x and y corresponding to entries `3` and `5` in the object table. Figure 5a shows the CFG of two functions where r_1-r_{10} represent

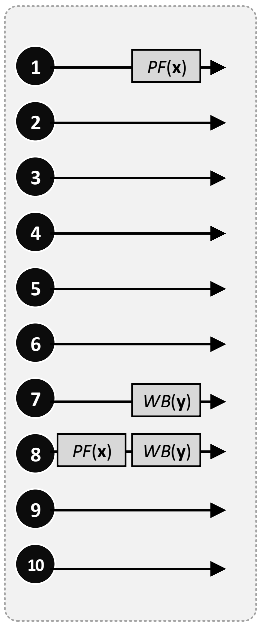


(a) Allocation Process



	IDX	MM_ADDR	SIZE	SPM_ADDR	A	PF_OP	WB_OP	WB	U	USERS
1	3	0x2000	80	a_1	0	1	0	0	0	1
	5	0x1000	40		0	0	0	0	0	0
2	3	0x2000	80	a_1	1	0	0	0	1	1
	5	0x1000	40		0	0	0	0	0	0
3	3	0x2000	80	a_1	1	0	0	0	1	2
	5	0x1000	40		0	0	0	0	0	0
4	3	0x2000	80	a_1	1	0	0	0	1	2
	5	0x1000	40		0	0	0	0	0	0
5	3	0x2000	80	a_1	1	0	0	0	1	1
	5	0x1000	40		0	0	0	0	0	0
6	3	0x2000	80		0	0	0	0	0	0
	5	0x1000	40	a_1	1	0	0	1	1	1
7	3	0x2000	80	a_2	0	0	0	0	0	0
	5	0x1000	40	a_1	0	0	1	0	0	0
8	3	0x2000	80	a_2	0	1	0	0	0	1
	5	0x1000	40	a_1	0	0	1	0	0	0
9	3	0x2000	80	a_2	1	0	0	0	1	1
	5	0x1000	40		0	0	0	0	0	0
10	3	0x2000	80	a_2	0	0	0	0	0	0
	5	0x1000	40		0	0	0	0	0	0

(b) Object Table: Entries (`3` , `5`)



(c) DMA Queue

Figure 5 Allocation Example.

regions. x is read in r_3 and r_9, and y is written in r_5. Note that function f, comprising regions r_7 to r_{10}, is called from two different call regions, r_4 and r_6. 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. Then, it is evicted to empty enough space for y to be allocated in r_5. However, x is also allocated inside function f at a different address a_2. Note that f could also be called from other, non represented call regions in the program, and assigning address a_2 to x might be required to fit x in the SPM together with other objects allocated in the unrepresented call regions.

We use program points ❶ to ❿ to follow the allocation process. Entries `3` and `5` of the object table are traced in Figure 5b; and the DMA queue is shown in Figure 5c. At ❶, x is allocated to address a_1 with `P` directive. In the object table, `PF_OP` is set to indicate x is being prefetched and `USERS` is incremented. A prefetch transfer $PF(x)$ is scheduled in the DMA queue. At ❷, `GETADDR (3)` checks entry `3` for the address; if $PF(x)$ has not finished at this point, the CPU stalls until the prefetch finishes; then x is allocated, `PF_OP` is reset, and `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, an allocation of x to a_2 is issued; however, 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 (3)` at ❹ returns a_1. When x is deallocated at ❺, `USERS` is decremented in entry `3`. 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. There is no need to write-back x as `WB` was not set. y is also allocated to a_1 at ❻ with `W` directive. So, no prefetch is scheduled and flags `A` and `WB` are set in entry `5`. Before y is used in r_5, `GETADDR (5)` is executed which sets `U` flag in entry `5` and returns address a_1. At ❼, y is deallocated and a write-back transfer is scheduled as both `WB` and `U` flags are set. f is called again in r_6. At ❽, a prefetch is issued for x to a_2. The DMA queue will have both $WB(y)$ and $PF(x)$ scheduled. At ❾, $WB(y)$ has finished while $PF(x)$ is still ongoing. The execution is stalled until the prefetch for x is complete, then `GETADDR (3)` returns a_2. Finally at ❿, x is deallocated.

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 to a_2 is not used. If instead f is called from r_6, x is allocated to a_2 and the object 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.

6 Allocation Problem

We now discuss how to determine a set of allocations for the entire program with the objective to minimize the WCET of the program. For the remaining of the section, we use 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. Finally, 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 $\wp(r_8^{\sigma \cup r_4}) = r_7^{\sigma \cup r_4}$.

We begin by formalizing the conditions under which a set of allocations is 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 in a region r_n, we have to assign an address in the SPM to the object. Hence, an allocation solution is represented by an assignment to the following decision variables over all regions $r_n \in R$ and all objects $v_j \in V$:

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

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

An allocation solution is feasible if the allocated objects fit in the SPM at any possible program point. As discussed in Section 5.3, 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 in a region-context r_n^σ:

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

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

We can determine the value of the helper variables based on the allocation:

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

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

$$\forall v_j, r_n^\sigma : avail_{\wp(r_n^\sigma)}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} = address_{\wp(r_n^\sigma)}^{v_j}. \tag{2}$$

$$\forall v_j, r_n^\sigma : \neg avail_{\wp(r_n^\sigma)}^{v_j} \land alloc_{r_n}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} = assign_{r_n}^{v_j}. \tag{3}$$

Equations 2, 3 specify the address in the SPM. If the object was already available in the parent region-context, then the address is the same. Otherwise, if the object is allocated in r_n, then the address is the one assigned by the allocation.

Example: refer to the example in Figure 5, where x is allocated with assigned address a_1 in r_2, r_3 and r_4 and with address a_2 in r_8 and r_9. For context $\sigma \cup r_6$, we have $avail_{r_7^{\sigma\cup r_6}}^{x} = 0$, since x is not available in r_6^σ, the parent of $r_7^{\sigma\cup r_6}$. Hence, we also have $address_{r_8^{\sigma\cup r_6}}^{x} = a_2$. However, for context $\sigma \cup r_4$ we obtain $avail_{r_7^{\sigma\cup r_6}}^{x} = 1$ and $address_{r_7^{\sigma\cup r_6}}^{x} = a_1$, hence $address_{r_8^{\sigma\cup r_6}}^{x} = a_1$.

Finally, given the object availability and address for each region-context pair, we can express the feasibility conditions for the allocation problem such that the allocated objects fit within the SPM and concurrent allocated objects have non-overlapping address ranges.

$$\forall v_j, r_n^\sigma : avail_{r_n^\sigma}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} + S(v_j) \leq S_{SPM}. \tag{4}$$

$$\begin{aligned} &\forall v_j, v_k, r_n^\sigma, j \neq k : (avail_{r_n^\sigma}^{v_j} \land avail_{r_n^\sigma}^{v_k}) \Rightarrow \\ &\qquad (address_{r_n^\sigma}^{v_j} + S(v_j) \leq address_{r_n^\sigma}^{v_k}) \lor (address_{r_n^\sigma}^{v_k} + S(v_k) \leq address_{r_n^\sigma}^{v_j}). \end{aligned} \tag{5}$$

Equation 4 states that if v_j is in the SPM during the execution of r_n^σ, then it must fit within the SPM size. Equation 5 states that if both v_j and v_k are in the SPM during the execution of r_n^σ, then their addresses must not overlap.

As long as Equations 4, 5 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 5.2 based on the usage throughout the whole sequence.

▶ **Example.** *Refer to the example in Figure 5, where x 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 5.3. As in Equation 2, 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 $avail_{\wp(r_n^\sigma)}^{v_j} = 0$).

6.1 WCET Optimization

For a given allocation solution $\{alloc_{r_n}^{v_j}, assign_{r_n}^{v_j} | \forall v_j, r_n\}$, 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 prefetching approach, the cost of DMA operations depends on the overlap: since the 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 7. 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 the best allocation; we discuss it next in Section 6.2.

Finally, we note that the proposed region-based allocation scheme is a generalization of the approaches used in related work on dynamic allocation. In [21], 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 these structures such as loops, conditional statements and functions as regions. The dynamic

Algorithm 1 Address Assignment

```
Input: region information, $\{alloc^{v_j}_{r_n} | \forall v_j, r_n\}$
 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:     $end\_addr_{r_n} = \max_{\sigma}\{end\_addr_{\wp(r^{\sigma}_n)}\}$
 5:     if $r_{n-1}$ is not sequentially composed with $r_n$ then
 6:         for all $v_j$ such that $alloc^{v_j}_{r_n}$ do
 7:             $assign^{v_j}_{r_n} \leftarrow end\_addr_{r_n}$
 8:             $end\_addr_{r_n} \leftarrow end\_addr_{r_n} + S(v_j)$
 9:     else
10:         for all $v_j$ such that $alloc^{v_j}_{r_n} \wedge alloc^{v_j}_{r_{n-1}}$ do
11:             $assign^{v_j}_{r_n} \leftarrow assign^{v_j}_{r_{n-1}}$
12:         for all $v_j$ such that $alloc^{v_j}_{r_n} \wedge \neg alloc^{v_j}_{r_{n-1}}$ do
13:             Compute $assign^{v_j}_{r_n}$ using best fit based on already assigned addresses
14:         $end\_addr_{r_n} \leftarrow \max_{v_j \text{ s.t. } alloc^{v_j}_{r_n}}\{assign^{v_j}_{r_n} + S(v_j)\}$
```

allocation in [24] is based on program points around loops, if statements and functions which can be matched with an entry/exit of a region. In [6], 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 9 we compare our proposed scheme against a standard dynamic allocation approach with no overlap using the same region-based program representation and search heuristic.

6.2 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.*, $fit > 0$.
- **Evolution Operations:** The evolution process incorporates 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 $P(t+1)$.

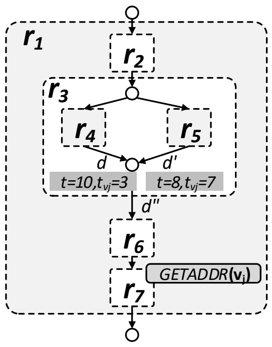


Figure 6 WCET Example: Merging states from different paths.

- **Termination:** The algorithm is terminated after k generations 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 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 allocated in a region cannot overlap with any object that is available in a parent; hence, Equation 5 is 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. Finally, the allocation is considered feasible only if the end address never exceeds the SPM size; this guarantees that Equation 4 is also satisfied.

7 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 6. We rely on the standard approach of Data Flow Analysis (DFA) [27], where the detailed state of the hardware is generalized into an *abstract state* based on the theory of abstract interpretation [4, 23]. 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, processor clock cycles.

Consider as an example the execution of the CFG and associated region tree in Figure 6 in a context σ. Assume that the analysis for the path through r_4^σ has determined 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 $t = 8, t_{v_j} = 7$, *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).

If we do not want to keep both states after the branch, a trivial solution would be to merge them by computing a join state 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. Due to the inherent complexity in the theory of abstract interpretation, in this section we present the main intuition about our abstraction and why it results in a safe WCET bound; a formal proof of correctness is provided in the technical report [20].

Intuitively, every 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 6, 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 $d'.t = 8, d'.t_{v_j} = 7$. The join state $d'' = d \vee d'$ is then computed as follows:

$$d''.t = t_{\max} = \max(d.t, d'.t), \tag{6}$$

and for every timer t_{v_j}:

$$d''.t_{v_j} = \max\big(d.t_{v_j} - (t_{\max} - d.t), d'.t_{v_j} - (t_{\max} - d'.t)\big). \tag{7}$$

Based on Equations 6, 7, 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. 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 7.

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 allocation commands for both the same and other objects. Due to the presence of the max term in Equation 7, modeling the WCET problem as an ILP (a technique also known as implicit path enumeration) would require adding a large number of auxiliary variables. Therefore, we propose to instead compute the WCET using a structure-based approach [27] using 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: $d \leftarrow$ ANALYZE_REGION($r_1, \emptyset, d$)
 2: return $d.t + \max_{v_j}\{d.t_{v_j}\}$

 3: function ANALYZE_REGION($r, \sigma, d$)
 4:     if $r$ is trivial region then
 5:         $d \leftarrow$ STATE_TRANSFER($r, \sigma, d$)
 6:         if $r$ calls a region $r_n$ then
 7:             $d \leftarrow$ ANALYZE_REGION($r_n, \sigma \cup r, d$)
 8:     else
 9:         for all paths $p_i$ in $r$ do
10:             $d_i \leftarrow d$
11:             for all subregions $r_n$ along $p_i$ do
12:                 $d_i \leftarrow$ ANALYZE_REGION($r_n, \sigma, d_i$)
13:         $d \leftarrow$ JOIN($r, \sigma, \{d_i\}$)
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 6, the availability and address of objects in the SPM 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. 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.

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 [23], as well as the value of program variables, which can be used to exclude invalid paths (flow analysis) and compute loop bounds [13].

8 Implementation

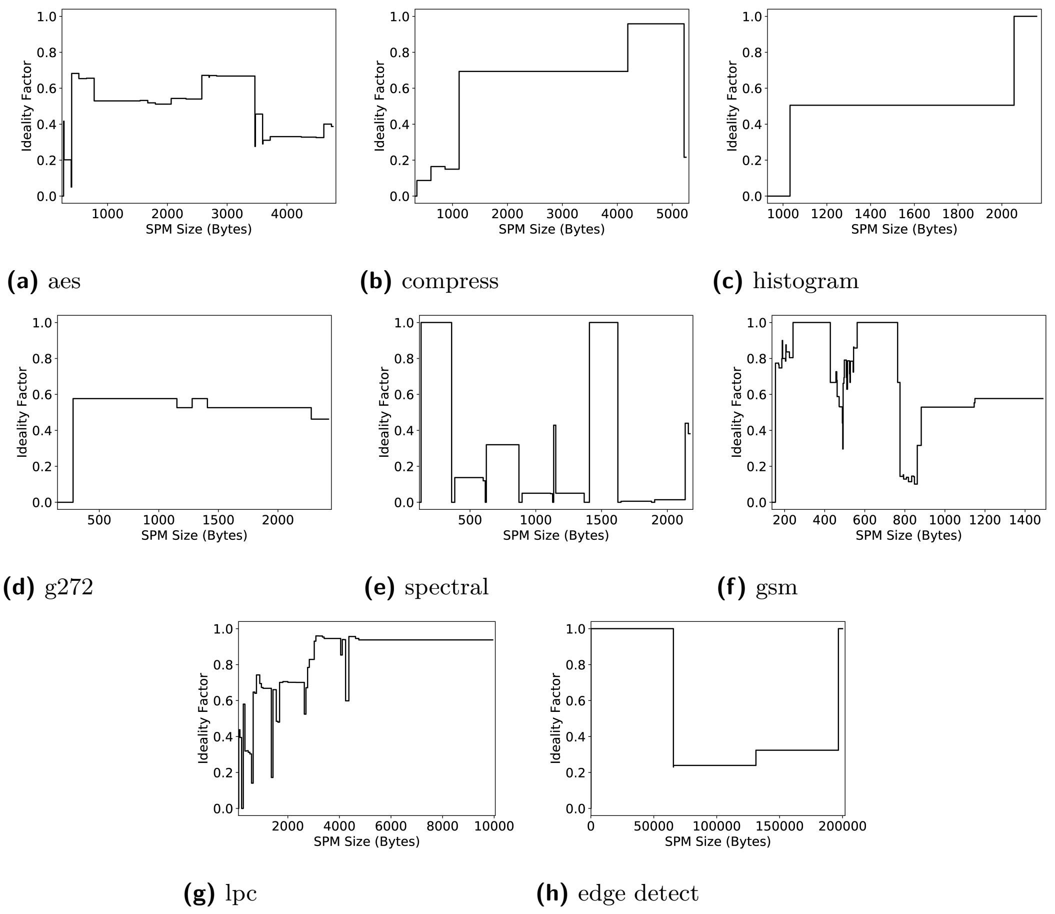
A compiler-integrated flow is used to implement the allocation algorithm. The flow analyzes the program, runs the allocation algorithm, applies the required transformations, and generates an executable. We integrated our flow with the open-source LLVM compiler [12]. The following passes are applied on the Intermediate Representation (IR).

- **Convert Stack Objects to Globals.** Each function frame in the stack has two components: (1) temporary spilled registers and calling context; (2) 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 [11]. This reduces the maximum stack size, so that the stack 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 6.2, 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 compiler 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.

9 Evaluation

The evaluation of the prefetching approach for data SPM is performed using a simple MIPS processor model with a 5-stages pipeline and no branch predictor. For memory instructions, we consider a latency for a word access of 10 cycles to main memory, 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 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. Note that the stack always resides in the SPM as its size



(a) aes **(b)** compress **(c)** histogram

(d) g272 **(e)** spectral **(f)** gsm

(g) lpc **(h)** edge detect

Figure 7 Ideality factor.

becomes small after converting stack variables to globals as discussed in Section 8 and its access rate is usually high.

As the benchmarks available for real-time systems are usually small kernels, we focus on the performance of the prefetching algorithm compared to dynamic allocation rather than the total profit of the allocation. We were not able to apply the algorithm to other suites with more realistic applications, *e.g.* SPEC2000, as they have system calls, recursion, unknown loop bounds, and calls to standard libraries which makes it unsuitable to derive WCET estimation. We plan to explore other benchmarks in the future.

We consider three cases: (1) dynamic allocation without prefetching (*base*); (2) dynamic allocation with prefetching (*pref*); (3) and dynamic allocation with no cost (*ideal*). We define the *base* case as the worst case for prefetching where the CPU has to stall for every memory transfer. We also define the *ideal* case as the dynamic allocation with no cost for memory transfer. Figure 7 shows the *ideality factor* as a function of the size of the SPM. The ideality factor is computed as: $ideality\ factor = \frac{WCET(base)-WCET(pref)}{WCET(base)-WCET(ideal)}$. The denominator of the ideality factor represents the best hypothetical improvement in WCET that prefetching can achieve relative to the base dynamic allocation and the numerator is the improvement for the prefetching case. The ideality factor is an indication for the performance of the prefetching approach, with a value of 1 indicating a performance equivalent to the ideal case, *i.e.*, there is no allocation cost as all memory transfers are overlapped with CPU execution. 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 to the size that can fit all objects.

The solving time for the allocation algorithm depends on the number of regions in the program, the number of objects that can fit in the SPM 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.

9.1 Results Analysis

Benchmark 'histogram' has two main arrays with size 1024 bytes each. When the size of the SPM is 1024 bytes, it can fit only one of them and dynamic allocation is able to arbitrate between the two arrays. Prefetching can overlap part of the cost needed for dynamic allocation. When the SPM size is 2048, both arrays can fit in the SPM and also prefetching technique can hide the whole memory time required to transfer the arrays as it can overlap the transfer of one array with the use of the other array in the SPM.

For benchmark 'g272', prefetching technique can only overlap part of the memory transfer as the live range of the used objects are overlapped, *i.e.*, the chance to transfer one object while using the others is low.

Benchmark 'edge_detect' has three arrays with size 64 Kbyte each and a small array with size 36 bytes. For small SPM size, only the small array can fit and prefetching can overlap its memory transfer time. When the SPM can fit one of the large arrays at any program point, prefetching can overlap 25% of memory transfer time. Similarly, prefetching can overlap 33% of the transfer time when the SPM can fit two large arrays. When the SPM can fit all the large arrays, all the memory transfer time can be hidden through prefetching.

The other benchmarks have more objects and the live ranges are more nested. The ideality factor varies as the SPM size increases and larger objects or more objects can fit in the SPM; hence more memory transfers are introduced. If the added space is used to arbitrate for objects, prefetching might not have enough time to overlap the memory transfers. If the space allows objects to exist in the SPM simultaneously, prefetching performs better as it has more opportunity to overlap the memory transfers.

Although the dynamic prefetching approach is able to exploit the opportunities to hide the latency of memory transfers, object-based allocation fails short in terms of space and time in some cases. That is, considering only objects that can fit entirely in the SPM at each program point limits the allocation efficiency. We argue that the benefit of our approach will increase for smaller allocation granularity. So, we plan to extend the framework to be able to allocate parts of an array or data structure to allow more allocation flexibility, increase the efficiency of allocation for smaller SPM sizes, and provide more chances for prefetching.

10 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 the 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. Using techniques like data pipelining can enhance the efficiency of the allocation algorithm for small SPM sizes by allocating portions of an object. We plan to integrate these mechanisms in our framework in future work.

Acknowledgements. We would like to thank Ondřej Lhoták for the helpful discussions and support on compiler design and implementation.

References

1 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*, pages 285–296, April 2015.

2 Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, November 2002.

3 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*, pages 1–8, Oct 2015.

4 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*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

5 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*, 14(3):279–291, March 2006.

6 Jean-Francois Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, ECRTS'07, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.

7 Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, December 2005.

8 Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Annual Design Automation Conference*, DAC'04, pages 238–243, New York, NY, USA, 2004. ACM.

9 Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2):32:1–32:36, November 2015.

10 Richard Johnson, David Pearson, and Keshav 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*, PLDI'94, pages 171–185, New York, NY, USA, 1994. ACM.

11 Sungjun Kim. Using scratchpad memory for stack data in hard real-time embedded systems. In *Proceedings of the Memory Architecture and Organization Workshop*, 2011.

12 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Gener-*

ation and Optimization: Feedback-directed and Runtime Optimization, CGO'04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

13 Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, School of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2002.

14 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*, pages 1–10, Aug 2014.

15 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS'15, pages 87–96, New York, NY, USA, 2015. ACM.

16 Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2):35:1–35:35, August 2016.

17 Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *ACM Trans. Embed. Comput. Syst.*, 8(3):21:1–21:32, April 2009.

18 David Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Elsevier, 2012.

19 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*, pages 269–279, April 2011.

20 Muhammad R. Soliman and Rodolfo Pellizzoni. Data Scratchpad Prefetching for Real-time Systems. Technical report, University of Waterloo, UWSpace, 2017. URL: `http://hdl.handle.net/10012/11837`.

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

22 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)*, pages 1–11, April 2016.

23 Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, 2004.

24 Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

25 Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. In *Proceedings of the 6th International Conference on Business Process Management*, BPM'08, pages 100–115, Berlin, Heidelberg, 2008. Springer-Verlag.

26 J. Whitham and N. Audsley. Studying the applicability of the scratchpad memory management unit. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 205–214, April 2010.

27 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem: Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. `doi:10.1145/1347375.1347389`.

28 Xuejun Yang, Li Wang, Jingling Xue, Tao Tang, Xiaoguang Ren, and Sen Ye. Improving scratchpad allocation with demand-driven data tiling. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES'10, pages 127–136, New York, NY, USA, 2010. ACM.

29 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*, volume 2, pages 358–365, Aug 2009.