# Static Function Prefetching for Efficient Code Management on Scratchpad Memory

Youngbin Kim
*Department of Computer Science*
*Yonsei University*
Seoul, Korea
yb.kim@yonsei.ac.kr

Kyoungwoo Lee
*Department of Computer Science*
*Yonsei University*
Seoul, Korea
kyounwgoo.lee@yonsei.ac.kr

Aviral Shrivastava
*Compiler-Microarchitecture Lab*
*Arizona State University*
Tempe, AZ
aviral.shrivastava@asu.edu

***Abstract*—As cache-based memory hierarchy is becoming a primary factor which limits the scalability and power efficiency of multi-core systems, scratchpad memory (SPM) has been studied as an alternative to cache. When SPM is used as an instruction memory, code management techniques are required to load code blocks on SPM using DMAs. In these techniques, code blocks are generally loaded on-demand to avoid loading incorrect block — unlike cache (e.g. tag arrays), SPM does not have mechanism to detect and recover from faults. While on-demand loading guarantees no fault, it leads to considerable performance overhead since it serializes the execution of DMA and CPU. This paper presents a technique to insert prefetching instructions for function-level code management to enable overlapping execution between DMA engine and CPU. Our technique inserts DMA instructions statically at compile time and does not rely on any profiling or run-time resources. Our evaluation shows that static prefetching can reduce CPU idle time due to DMAs by 58.5% and achieves 14.7% of average performance improvement on the benchmarks showing high overhead due to DMAs.**

***Index Terms*—Scratchpad Memory, Code Management, Compilers**

## I. Introduction

As the number of cores in a system increases, cache-based memory hierarchy is becoming a primary factor which restricts the scalability and power efficiency of the system. Scratchpad memory (SPM) is a software-controlled on-chip SRAM memory, which can be used as an alternative to cache. SPMs are more efficient in terms of power and area as well as more scalable than caches since they lack the power-hungry hardware for caching (e.g., tag array and address calculation logic). On the other hand, the elimination of the caching logic comes at the cost of the explicit data management; data should be moved between SPMs and memory explicitly by software, usually via DMA operations.

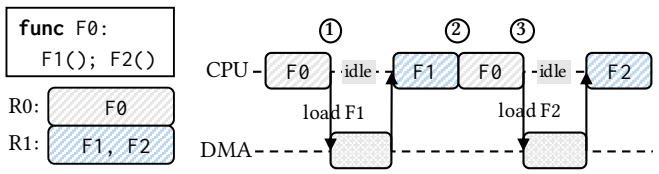
Different memory areas – heap, stack, global, and code – are usually managed with their own algorithms since each area presents unique data access patterns. In this paper, our focus is on the code management, which has significant impact on the application performance [1]–[3]. When a SPM is used as an instruction memory, code management techniques are required to load code blocks. In particular, function-level management [1]–[8] has been popular since functions are easily relocatable at link time, which allows code overlay mechanism. Such an approach divides the SPM into regions, each of which has a unique address range, and maps each function to one of the regions. At run-time, the entire function is loaded on its region using a DMA operation before the function call, only if it is required. The newly loaded function evicts the function previously loaded on that region.

Since each DMA incurs performance overhead (CPU has to wait until the load finishes), several studies proposed techniques to find more efficient mappings [1], [2], [4]–[7]. These approaches attempted to minimize the total number of DMA executions by separating caller and callee of frequently executed calls (e.g., function calls in a loop) into different regions. The execution time of a program is significantly affected by the quality of the mapping [1]–[3].

Although the literature extensively investigated the mapping algorithms (*where* to load the code blocks), the issue of *when* to load them has not been addressed completely. While there exist a fair amount of researches on prefetching in cache-based systems, existing SPM-based code management techniques generally rely on *on-demand* loading which is to start loading the callee right before its call. This is because SPM-based systems have no mechanism to recover from *fault* at run-time; in cache-based systems, incorrect prefetches can be detected by comparing tags at run-time and the correct code block can be recovered. On the contrary, prefetching incorrect functions on SPM makes CPU silently execute incorrect codes and mostly leads to the crash of the program.

While on-demand loading is *safe* — free from the possibility of fetching incorrect code blocks, it brings considerable performance overhead since it makes CPU idle during the execution of the DMA. The overhead becomes more significant as SPM manages code in larger granularity (i.e. function) than cache in general — control is blocked until the entire function is loaded. Indeed, our experimental study shows that the CPU execution time is wasted on waiting for DMA completions by up to 58.8%. Prefetching techniques can be utilized to efficiently mitigate this overhead for such a case; however, as discussed, inserting prefetch instructions is not trivial since prefetch in SPM-based system can be invoked only when the next callee is clearly known, rather than in a speculative manner.

In this paper, we propose a compilation technique to statically insert prefetching instructions for function-level code management. We present an algorithm to find efficient *and* safe




(a) Example function and mapping

(b) Execution of the program with the given mapping without prefetching

Fig. 1: In function-to-region mapping, SPM is divided into regions and each function is mapped to one of the regions.

prefetch locations for every call instruction in the program. Based on the found prefetch locations, we propose intelligent DMA insertion techniques which insert most efficient management functions depending on the context. The contributions of our technique can be summarized as follows.

- Our technique brings additional performance improvement over the state-of-the-art mapping technique [9] without need for modifying the mapping algorithms. Our evaluation shows that our technique reduces the CPU idle time by 58.5% compared to on-demand loading within the *same* mapping and the SPM size. It enables the improvement of the execution time by 14.7% on average among the benchmarks having high CPU idle time.
- Our technique only relies on the static analysis of the program and does not require any profiling information or run-time data structure. It makes our technique easily applicable on a new program without any requirement or dependency. Also, the benefits of SPM-based system (e.g. power efficiency, scalability, and predictability) can fully remain since our technique does not require additional run-time resource.
- Our proposal is orthogonal to the mapping algorithms and other optimization techniques. Our technique can be applied to any mapping with the interface of function-to-region mapping, simultaneously with the optimizations such as [3] (reducing fragmentation) and [8] (eliminating unnecessary management functions).

The rest of the paper is organized as follows. In section II, we introduce the idea of function-level code management along with the related works. Section III describes detailed implementation of our static prefetching technique. We present our evaluation and discussion of our technique in section IV. Section V concludes the paper.

## II. Background and Related Work

### A. Function-level Code Management

Function-level code management uses function-to-region mapping [1] to decide where to load each function of a program. Figure 1a shows a program example and its mapping: three functions with two SPM regions where `F0` (assigned to a region `R0`) calls `F1` and `F2` (assigned to the same region `R1`). Figure 1b illustrates the run-time execution of the program without prefetching. When `F1` is called (①), it should be loaded via DMA since `R1` is initially empty and CPU becomes idle. On the other hand, when `F1` returns to `F0` (②), no DMA is required since `F0` is already loaded on the SPM (region `R0`). Finally, when `F2` is called (③), it should be loaded by a DMA since currently `F1` is loaded on `R1`. As mentioned, this conventional approach is safe but distant from efficiency — CPU is completely blocked while DMA executes. Our proposal is to bring more efficient code management by enabling overlapping execution between CPU and DMA engine through code prefetching (e.g. `F1` and `F2`).

### B. Related Work

This work lies in the context of code management on scratchpad memories. Several studies have proposed code management techniques at the granularity of a function. Especially, our work is based on function-to-region mapping [1], [2], [4] which splits SPM space into regions and maps functions to one of the regions. It is advantageous and distinguished from other works in two ways: 1) it manages all the data on SPM (CPU does not need to access main memory) and 2) it does not rely on any profiling information. Many approaches require CPU capable of directly accessing memory [6], [7], [10] or rely on profiling information [7], [10], [11], which limit the applicability of their techniques.

Several studies have addressed scheduling of DMA instructions in a basic form in the area of *stack* and *global data* management. In these works, a program is divided into sections (or regions), usually based on the loops and functions. Then, most frequently used data in each section is copied into SPM before entering the section. However, these techniques cannot be directly applied on our code management since they map only subset of data onto SPM [12]–[14]. Other techniques [9], [15] rely on expensive software caching to manage heap data, which is sub-optimal for managing data showing much predictable access patterns such as code.

In the context of compiler-based data prefetching technique on SPMs, Soliman [16] presented a prefetch-aware data allocation technique to minimize worst-case execution time of a program. Yang [17] also proposed an array prefetch technique to overlap DMA and CPU execution over multiple iterations of a loop. However, these works only focused on prefetching *data* and can be applied to limited structure of a program [17]. Our work exploits the unique access patterns of code so that the most efficient prefetch methods are inserted for each function call depending on the context (Section III-C).

In more recent works, Cai [8] and Kim [3] proposed optimization techniques for function-level code management. Cai [8] applied cache analysis on the code management with SPMs to remove unnecessary management codes. Kim [3] presented a technique that splits functions of the code into smaller ones so that they can be managed more efficiently with less fragmentation. These approaches are beneficial since they reduce the total amount of management functions in a program and can be applied simultaneously with our technique. However, they have not addressed the issue of when to load for the code that still needs the management. It results in

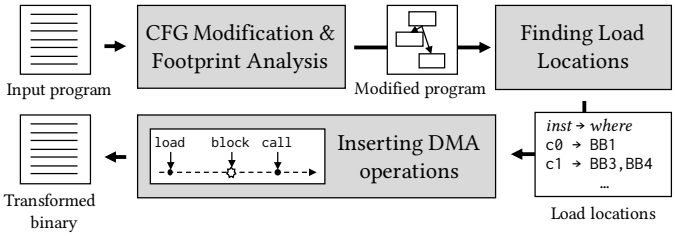



Fig. 2: Our technique modifies the original program through three steps for effective function prefetching.

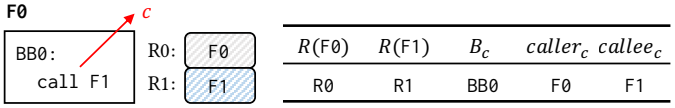



Fig. 3: The notations used in Section III with an example

a large execution time overhead on some applications where most of the management functions are necessary thus cannot be removed.

Finally, our approach can be compared to the works in non-sequential instruction prefetchers [18]–[21]. Like our technique, these prefetchers aim to prefetch instructions which are not in the next cache line, such as codes after function calls or branches. However, these works are different from ours and cannot be directly applied to SPM management in two reasons: first off, most of them rely on the extended hardware to learn and speculate the next required code block at run-time, except [19] which proposes software prefetching. Secondly and more importantly, mis-speculation of the next code is not allowed in SPM-based management; while prefetching a wrong line in cache-based architecture just incurs a performance penalty, it results in an incorrect execution of the function in the system managed by SPMs.

## III. Our Approach

Our technique is to improve the parallelism between DMA and CPU execution by statically inserting prefetch instructions for all function calls in the given program. It consists of three steps shown in Figure 2: CFG modification & Footprint analysis, Finding load locations, and Inserting DMA operations. In the first step, we modify the CFGs (Control Flow Graphs) of the program for the analysis and collect relevant information for prefetching. In the second step, we find where to insert prefetch instructions for each function call using static analysis. Finally, we insert the DMA instructions and the management functions at the right locations.

Throughout the section, we use the example function `main` depicted in Figure 4 to describe our method. The original CFG of the function is presented in figure 4a. Regions and footprints (section III-A) of all functions are described in Figure 4b. For the sake of brevity, instructions other than `call`, which do not affect our algorithm, are omitted in the figure. Also, following notations are used in this section. Figure 3 illustrates these notations with a simple example.

$R(f)$: the region where the function $f$ is mapped.
$B_i$: basic-block where the instruction $i$ resides.
$caller_c$: the function having call instruction $c$.
$callee_c$: the function called by call instruction $c$.
$F(f)$: set of regions that $f$ may change during its execution. Detailed description is in section III-A.
$L(c)$: set of basic-blocks that prefetch instructions should be located for call instruction $c$. Detailed description is in section III-B.

### A. CFG Modification & Footprint Analysis

*1) Basic-block Splitting:* To maximize the efficiency of prefetching, it is desirable to start prefetch the next function as soon as the region is available. Therefore, it is natural to insert prefetch instruction right after the call instruction — when the callee finishes using the region. In this step, we split basic-blocks of the program at every call instruction. It makes every basic-block has *at most* one call instruction, and if it has, it would be the last instruction before the terminator (`branch` or `return` instruction). This makes the head of every basic-block the best candidate for inserting prefetch instruction; whenever call instruction is finished, the control moves to another basic-block.

Figure 4c shows the CFG after this transformation along with the split points illustrated with the red dotted lines. An unconditional branch is inserted after the call instruction and the remaining instructions form a new basic-block. Note that the additional branches do not affect the performance of the final binary. Code generator does not insert unnecessary `jump` instruction if the target basic-block has only one incoming branch.

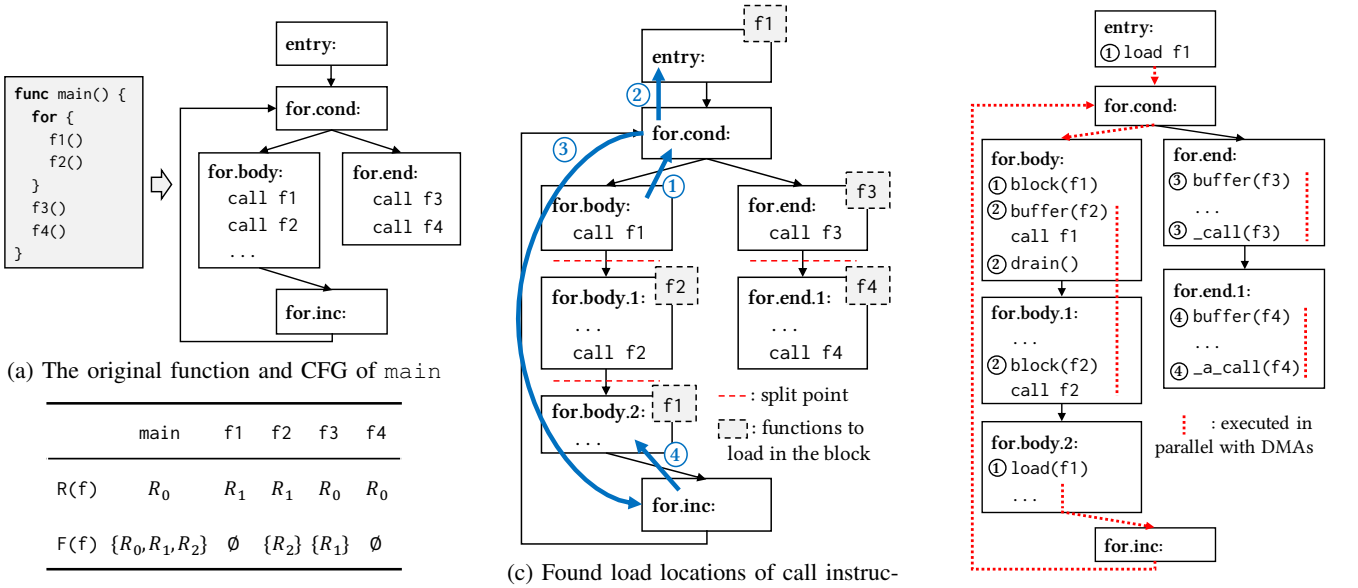
*2) Finding Footprint:* When a function $f$ is called, $R(f)$ is always used to load the code of $f$. However, calling certain functions can result in modifying more regions other than $R(f)$ when $f$ has a call to another function. For example, if `f1` has a call to `f2`, the region $R(\texttt{f2})$ can also be modified with the code of `f2` during the execution of `f1` if the control reaches the call instruction. Such information is essential to guarantee safeness of prefetching — e.g., prefetching for $R(\texttt{f1})$ *and* $R(\texttt{f2})$ cannot be placed before calling `f1` since both regions can be modified during the execution of `f1`.

In this step, we find *Footprint* for every function in the program. Footprint $F(f)$ is defined as the regions that function $f$ may change during its execution, by calling other functions. Using this notation, we can say $R(\texttt{f2}) \in F(\texttt{f1})$ in the case of the example above. In other words, when a function `f` is called, $R(\texttt{f})$ is always written by `f` and the regions in $F(\texttt{f})$ *may* be overwritten depending on the execution. It can be defined in more formal form as follows.

$$G(f) = \{\, g \mid g \text{ is a function called by} f \,\}$$

$$F(f) = \begin{cases} \emptyset & \text{if } G(f) = \emptyset \\ \{\, R(g) \mid g \in G(f) \,\} \cup \bigcup_{g}^{G(f)} F(g) & \text{otherwise} \end{cases}$$

This information is used to guarantee the safeness of prefetching: function $f$ can be safely prefetched before the function




(a) The original function and CFG of `main`

| | main | f1 | f2 | f3 | f4 |
|---|---|---|---|---|---|
| R(f) | $R_0$ | $R_1$ | $R_1$ | $R_0$ | $R_0$ |
| F(f) | $\{R_0, R_1, R_2\}$ | $\emptyset$ | $\{R_2\}$ | $\{R_1\}$ | $\emptyset$ |

(b) Regions and footprints of the functions

(c) Found load locations of call instructions

(d) The final CFG with management codes

Fig. 4: CFG modification and load location finding on the example function `main`.

$g$, if $R(f) \neq R(g)$ and $R(f) \notin F(g)$ (discussed in more detail in section III-B). Otherwise, $f$ should be loaded after the execution of $g$ since $g$ may overwrite $R(f)$ in its execution.

### B. Finding Load Locations

We present an algorithm to find where to place prefetch instructions for every call instruction in the program. We use notation $L(c)$ for the set of basic-blocks where the DMA issues should be inserted for a call instruction $c$. Specifically, $L(c)$ should satisfy two conditions to guarantee no fault occurs: i) load should be issued before the execution of $c$ for *every* control path and ii) once prefetched, the region $R(callee_c)$ should not be overwritten by a different function before the execution of $c$.

To deal with the condition i), we first set *target* basic-block $T_c$ which represents the desirable prefetch location. If $c$ resides in a loop, $T_c$ is set to the predecessor of loop header; otherwise, it is set to the immediate dominator of $B_c$. A basic-block $d$ is a *dominator* of $b$ if every path from the $entry$ to $b$ must go through $d$, where $entry$ is the first basic-block of the function. Among the dominators, immediate dominator can be recognized as the closest block from $b$. Thus, if we put a DMA instruction on it, every path reaching $B_c$ is affected. Then, the algorithm walks through the paths from $B_c$ to $T_c$ to guarantee the condition ii); if a block $B'$ (located between $B_c$ and $T_c$) writes to the region $R(callee_c)$, the prefetch instruction for $c$ should be inserted after $B'$ executes, instead of $T_c$.

Algorithm 1 shows a recursive method to find all load locations of call instruction $c$. The names and descriptions of five input parameters – $c$, $current$, $NEXT$, $target$ and $VISITED$ – are listed. Initially, we set $current = B_c$ and $NEXT = P$ where $P$ is a set of predecessors of $B_c$. $VISITED$ should be $\emptyset$ since we have not visited any basic-block yet; this will be filled as the algorithm proceeds. Therefore, initial call should be $FIND(c, B_c, P, T_c, \emptyset)$ and it returns the load locations $L(c)$.

**Algorithm 1** Finding Load Locations

**Input:**
$c$: the `call` instruction the algorithm is working on
$current$: current basic-block in the walk-through
$NEXT$: a set of basic-blocks we want to visit next
$target$: *target* basic-block $T_c$
$VISITED$: a set of basic-blocks visited in the walk-through

```
1:  function FIND(c, current, NEXT, target, VISITED)
2:      locations ← ∅
3:      if current = target then
4:          return {current}
5:      for next ∈ NEXT do
6:          if next ∈ V then
7:              continue
8:          if ∃ call instruction x ∈ next then
9:              if R(callee_c) = R(callee_x) or
10:             R(callee_c) ∈ F(callee_x) or
11:             R(callee_c) = R(caller_x) then
12:                 locations ← locations ∪ {current}
13:                 continue
14:         VISITED ← VISITED ∪ {current}
15:         P ← predecessors of next
16:         locations ← locations ∪
17:                     FIND(c, next, P, target, VISITED)
18:     return locations
```

Starting from basic-block $current$, the algorithm recursively visits its predecessor $next$ (line 5) until it reaches $T_c$ (line 3). At each basic-block, it checks whether $next$ violates the condition ii); if $next$ has a call instruction $x$, there are three cases where $x$ overwrites the region of callee ($callee_c$). In these cases, prefetch instructions cannot be inserted before

$x$ (line 8-11):

- $R(callee_c) = R(callee_x)$: callee of $c$ and $x$ use the same region. The region is available only after $x$ is finished (line 9).
- $R(callee_c) \in F(callee_x)$: in the execution of $callee_x$, the region $R(callee_c)$ may be overwritten (line 10).
- $R(callee_c) = R(caller_x)$: $callee_c$ is mapped to the region of currently executing function ($caller_x$). It implies $x$ eventually returns to the $caller_x$ and the region is available only after $x$ returns (line 11).

If one of the above conditions are met, the algorithm adds the current block $current$ into prefetch locations (line 12). Otherwise, it updates the visited block (line 14) and proceeds to the predecessor blocks (line 16-17). Note that the algorithm terminates, because it should either reach $target$, which dominates the starting basic-block $B_c$, or make cycles. Both cases are covered in the algorithm (line 3 and 6) and the recursion stops in either cases. For the case of recursive function calls, we do not add any extra prefetch management (leave $L(c) = \emptyset$); when a function calls itself, it is guaranteed that the function is already loaded on the SPM.

Figure 4c shows an example for the case of `f1`, where $T_c$ = `entry`. The algorithm goes through two paths since `for.cond` has two predecessors. The first path (①-②) terminates since it reaches $T_c$. On the other hand, the second path (①-③-④) stops at `for.body.2` since `f2` in the predecessor block is mapped to the same region `R1`.

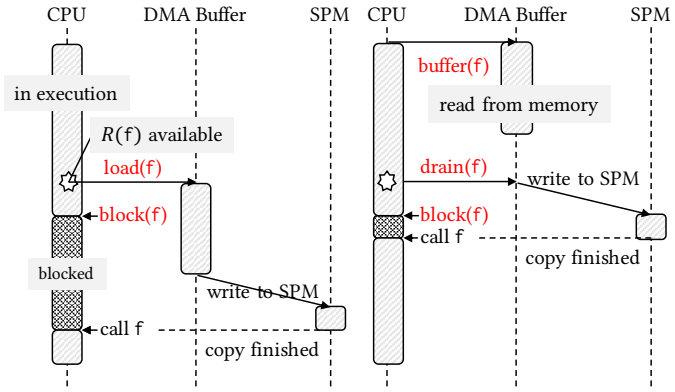
### C. Inserting DMA Operations

In this step, we present a procedure to insert the DMA operations based on the found load locations $L(c)$. Specifically, we propose two DMA insertion techniques: SFP (Static Function Prefetching) and A-SFP (Aggressive-SFP). SFP is a basic prefetch insertion technique consisting of two different management scenarios. In A-SFP, we propose more aggressive use of DMA buffer with the extra scenarios and management functions.

In each technique, we classify basic-blocks in $L(c)$ into two cases: *normal* and *conflict*. *Conflict* represents the case that $caller_c$ is evicted during the execution of $callee_c$. This happens when they are in the same region ($R(caller_c) = R(callee_c)$) or the region of the caller is changed during the execution of the callee ($R(caller_c) \in F(callee_c)$). This case should be managed differently since the management code in the caller itself can be overwritten during the execution of the callee. The other case — the code of $caller_c$ remains unchanged after $callee_c$ returns — will be denoted as *normal*.

First, we introduce management functions implemented for our technique and highlights the idea of aggressive buffer management of A-SFP (section III-C1). In next sections, we discuss the detailed implementation of SFP (section III-C2) and A-SFP (section III-C3).

*1) Management Functions:* The management functions implemented for our technique — `load`, `buffer`, `drain` and `block` — are illustrated in Figure 5. As shown in Figure 5a, `load(`$f$`)` makes DMA engine copy the function to its




(a) `load` and `block` (b) `buffer` and `drain`

Fig. 5: Prefetching can be initiated via either `load` or `buffer` (+ `drain`) management functions.

buffer and write the loaded code on $R(f)$ as soon as the buffer is filled. Then `block(`$f$`)` keeps the CPU blocked until the loading of $F$ is finished. In Figure 5b, two other management functions, `buffer` and `drain`, are presented. Unlike `load` function, `buffer(`$f$`)` requests the DMA engine to fill its buffer with the code of $f$ but not to copy the code to SPM automatically. Instead, the data in the buffer is copied to the memory after the program calls `drain()`. Note that all the required information, such as the address and the size of the functions, can be obtained from the function $f$ and are already known at the compile-time.

As Figure 5 implies, using `buffer` (Fig. 5b) can result in further parallelization than using `load` (Fig. 5a) since it can early-start the memory access, which consumes majority of DMA execution time. On the other hand, using `buffer` requires more aggressive implementation and cannot replace every instance of `load` since it holds the DMA buffer and restricts issuing new DMA invocation (detailed discussions are in Section III-C3). Therefore, we propose and compare two DMA insertion techniques: SFP (Static Function Prefetching), which manages prefetching using mostly `load`, and its aggressive extension A-SFP (Aggressive-SFP). A-SFP not only uses `load` management function but also exploits `buffer` whenever it is eligible.

*2) SFP (Static Function Prefetching):* SFP inserts the management functions in two different styles depending on the type of the blocks (*normal* or *conflict*). Figure 6 shows DMA insertion scenarios with different management functions and conflict conditions. As depicted in gray box, two cases (ⓐ and ⓑ) are fall into SFP.

In *normal* case (ⓐ in Fig. 6), we directly insert the management functions on $caller_c$ since it will not be overwritten during the execution of $callee_c$. First, `load` is inserted on the heads of basic-blocks $b \in L(c)$. Secondly, `block` is inserted before the call instruction $c$ to prevent executing $caller_c$ before the load is finished.

On the other hand, in *conflict* case (ⓑ in Fig. 6), we cannot directly insert all the management functions on $caller_c$

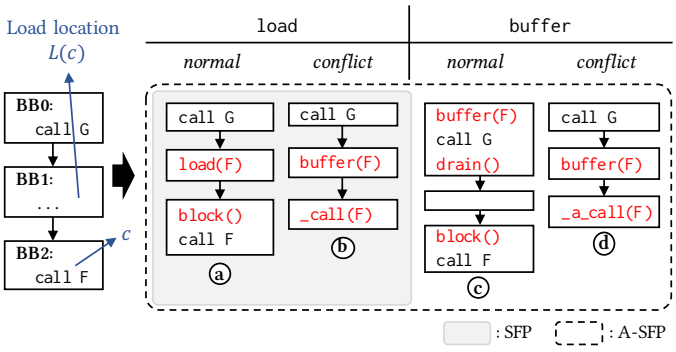



Fig. 6: An example showing DMA insertion scenarios where $c$ is in `BB2` and $L(c)$ = {`BB1`}. Depending on the techniques and conflictness, the management code can be inserted in four different styles.

```
   function _call
        (caller, callee, args):
1       drain()
2       block(callee)
3
4       ret ← callee(args)
5       load(caller)
6       block(caller)
7       return ret
```

```
   function _a_call
        (caller, callee, args):
1       drain()
2       block(callee)
3       buffer(caller)
4       ret ← callee(args)
5       drain()
6       block(caller)
7       return ret
```

Fig. 7: Two wrapper functions used in *conflict* case.

since the code in $R(caller_c)$ would be replaced with another function when $callee_c$ returns. To deal with this problem, we introduce wrapper functions, `_call` and `_a_call`, which are responsible for bringing back $caller_c$ before the control returns to it. SFP first inserts `buffer` on $L(c)$ to initiate the loading of $callee_c$ and replaces $c$ with `_call` wrapper. We assign a private SPM space for these wrappers so that they can bring back $caller_c$ without the concern of eviction. The wrappers requires additional ∼150 bytes of SPM space and can be reused across all the functions in the program.

Figure 7 illustrates the wrapper functions used in *conflict* case. SFP only uses `_call` while `_a_call` is utilized in A-SFP. The implementation of `_call` (left side of the figure) is straightforward; first, it waits for the load of $callee_c$ by `drain` and `block` (line 1 and 2). After the function is called (line 4), it executes `load` for $caller_c$ (line 5) since the caller could be overwritten by another function while executing $callee_c$. Once the load of the caller is completed (line 6), the control returns (line 7).

*3) A-SFP (Aggressive-SFP):* A-SFP extends SFP with two additional DMA insertion scenarios depicted in Figure 6 (ⓒ and ⓓ), which exploit `buffer` management function. The idea behind A-SFP is that 1) DMA transaction consists of two tasks (memory-to-buffer and buffer-to-SPM) and 2) memory-to-buffer takes majority of the time since it involves accessing slow main memory. While standard `load` executes both tasks at once, we divide it into two management functions: `buffer` (memory-to-buffer) and `drain` (buffer-to-SPM). By executing them separately, it is often possible to hide expensive memory access latency via invoking memory-to-buffer transaction first while the target SPM region is in use.

Figure 6 ⓒ shows the transformation with the *normal* case. In the example, A-SFP places `buffer` before `call G` to execute memory-to-buffer transaction while `G` is occupying target region. After `G` returns, `drain` is called to copy the loaded code from buffer to SPM. However, this pattern can be used only if following two conditions are met: i) the predecessor of $b \in L(c)$ has a `call` instruction $x$ and ii) $F(x) = \emptyset$. If there is no such $x$ (condition i)), it means there is no function to overlap execution. On the other hand, the condition ii) requires that $callee_x$ has no DMA issue; if we put `buffer` before $x$ where $F(x)$ is not empty, $callee_x$ cannot load other functions since the DMA buffer is held, assuming standard FIFO buffer. If these conditions are met, A-SFP puts `buffer` before $x$ and inserts `drain` after $x$ as in Figure 6 ⓒ. Also, `block` is inserted before $c$ to make control wait until the DMA finishes.

In *conflict* case (ⓓ in Fig. 6), the transformation is same as SFP except that A-SFP uses `_a_call` instead of `_call` wrapper. In the right side of Figure 7, the implementation of `_a_call` is presented. The only difference from `_call` is that `_a_call` uses `buffer` and `drain` (line 3 and 5) to overlap loading of $caller_c$ with the execution of the $callee_c$. Like in *normal* case, `_a_call` can be used only when $callee_c$ does not call any other function ($F(callee_c) = \emptyset$).

*4) Example:* Figure 4d shows the final CFG after inserting management codes. Especially, four functions in the example (`f1` ∼ `f4`) represent each different DMA insertion scenarios. Note that load locations for each function are depicted in Figure 4c; e.g. $L$(`f1`) = {`entry`, `for.body.2`}. While `f1` and `f2` are correspond to *normal* case, `f3` and `f4` are in *conflict* case since they are mapped to the same region as `main` ($R_0$). Among them, `f1` (① in Figure 4d) and `f3` (③) show the examples of SFP. On the other hand, the cases of aggressive management of A-SFP are illustrated with the functions `f2` (②) and `f4` (④). This example demonstrates that our technique can insert most efficient management functions depending on the different contexts. As a result, Figure 4d shows that most of the program execution can be overlapped with the DMAs (depicted in red lines).

## IV. Experimental Results

### A. Setup

We implement the proposed technique as a pass in LLVM [22] compiler infrastructure. MiBench benchmark suite [23] are compiled with `-O3` flag and used for the experiments. To find the mapping, CMSM [2] is used, which is recognized as a state-of-the-art mapping technique. Also, we extend the gem5 [24] simulator with an SPM and DMA engine. CPU frequency is set to 3.6 Ghz on a x86 out-of-order execution and DDR4 model provided by gem5 is used. The size of data cache is 64kB. Both the data cache and the instruction SPM

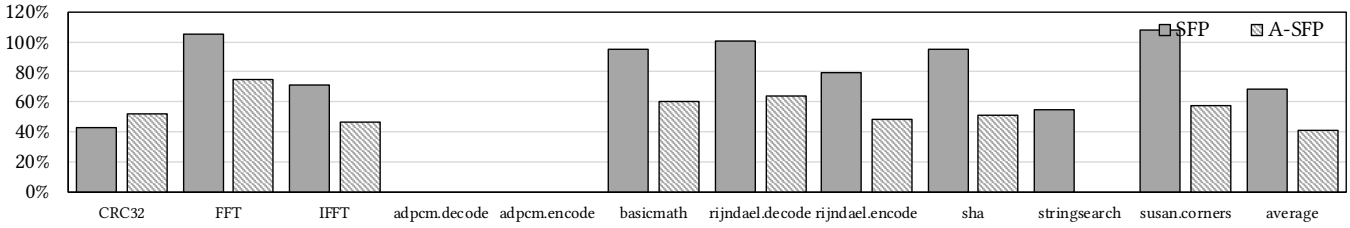



Fig. 8: DMA synchronization overhead of SFP and A-SFP when the number of regions is two, normalized to *baseline*.

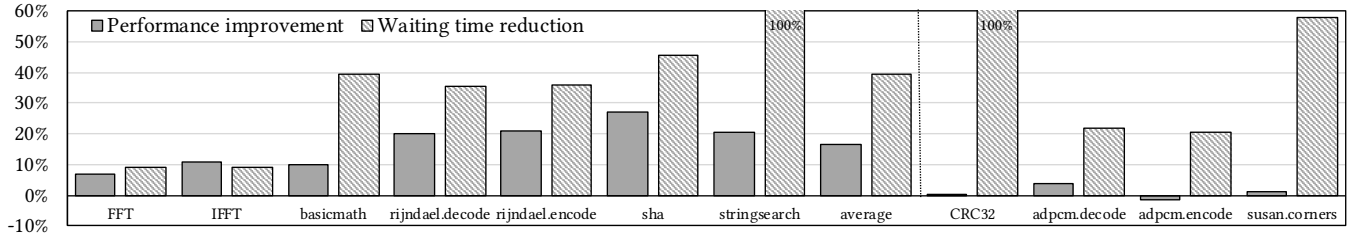



Fig. 9: Performance improvement and reduction on CPU idle time enabled by A-SFP.

have two-cycle latency. DMA buffer is assumed large enough for storing any single function in the program (∼2.5kB).

DMA overhead is modeled in two parts: setup time and transfer time. Setup time is a constant time required for every startup of DMA calls. This is set to 91 nanoseconds by referencing IBM Cell BE [25] processor which is a SPM-based multicore architecture. Transfer time is proportional to the size of data transferred and set to 0.24 cycles/byte based on the same reference architecture.

We evaluate our proposal with the *baseline* configuration, which starts loading callee right before every call instruction and waits until the load finishes (no prefetching). We first evaluate the benefit of aggressive buffer management of A-SFP in terms of CPU idle time reduction. Next, we present discussion about the impacts of A-SFP on execution time. Finally, we show the result about scalability evaluated with the different number of regions.

### B. Effectiveness of Aggressive Buffer Management

In this section, we evaluate and compare the effectiveness of SFP and A-SFP in terms of CPU idle time reduction. In this experiment, SPM size for each benchmark is set to have two regions. Given that the most of benchmarks consist of $2 \sim 6$ functions (Table I), this configuration is general [3], [8] and can incur all the different management scenarios presented in section III-C. Figure 8 presents the idle time reduction of each prefetching techniques, normalized to *baseline*. On average, SFP reduces the DMA synchronization overhead by 35.6% compared to the case without function prefetching. A-SFP further increases the reduction rate to 58.5% by exploiting aggressive DMA buffer management.

As shown in Figure 8, A-SFP is effective for most of the benchmarks but notably in `IFFT`, `rijndael.encode`, `sha` and `stringsearch`. It is beneficial over SFP especially for the case when two functions, which are mapped in the same region, are called sequentially in a short time. For example, in benchmark `stringsearch`, function `strsearch` is called right after calling `init_search`. Improvement from SFP is limited since it can start to load the second function (`strsearch`) only after the first function (`init_search`) finishes. On the other hand, A-SFP starts to load the second function *before* the first function is called. Thus, the DMA request for the second function can be performed in parallel with the execution of the first function, which results in the significant reduction on DMA synchronization overhead.

### C. Effectiveness on Execution Time

In this section, we discuss the impact of A-SFP on the overall execution time. For this experiment, we study the configuration with one or two regions where most of the benchmarks show more than 10% of CPU idle time. Table I shows the detailed information about the benchmarks with their CPU idle time and the number of regions. Four benchmarks whose waiting times do not exceed 10% are presented at the bottom. Given that the SPM size is 52.8% of the entire program size on average (Table I), we believe this configuration is realistic. It is also suitable for showing the relevance of A-SFP with the execution time since our technique is mainly designed for the environment with considerable amount of DMAs; although A-SFP works consistently with larger SPMs (section IV-D), the performance improvement is inherently bounded to the amount of DMAs and CPU idle time before applying prefetching.

Figure 9 shows the performance improvement in terms of overall execution time. Considering the benchmarks having larger than 10% of CPU idle time, the execution time is improved by 14.7% and idle time due to DMAs is reduced by 40.0% on average. Among the benchmarks, `FFT` and `IFFT` show the marginal improvement (∼10%). This is mainly because `FFT` and `IFFT` have only one region (only *conflict* case). At the same time, the largest function `fft_float` in two benchmarks cannot be managed by `_a_call` since it calls another function. On the other hand, `sha` shows

TABLE I: CPU idle time and the number of regions selected for the performance evaluation

| Benchmark | CPU idle time | # Regions | # Total Functions | # DMAs | SPM size / Code size |
|---|---|---|---|---|---|
| FFT | 17.0% | 1 | 6 | 8,205 | 55.0% |
| IFFT | 27.6% | 1 | 6 | 16,397 | 55.0% |
| basicmath | 36.3% | 2 | 3 | 38,013 | 61.8% |
| rijndael.decode | 58.5% | 2 | 5 | 38,985 | 39.8% |
| rijndael.encode | 59.1% | 1 | 5 | 38,989 | 39.8% |
| sha | 43.8% | 1 | 4 | 9,835 | 50.0% |
| stringsearch | 17.1% | 2 | 3 | 116 | 40.0% |
| CRC32 | 0.0% | 1 | 2 | 3 | 65.7% |
| adpcm.decode | 2.8% | 1 | 2 | 1,371 | 66.4% |
| adpcm.encode | 2.7% | 1 | 2 | 1,371 | 68.9% |
| susan | 3.7% | 1 | 9 | 10 | 37.4% |

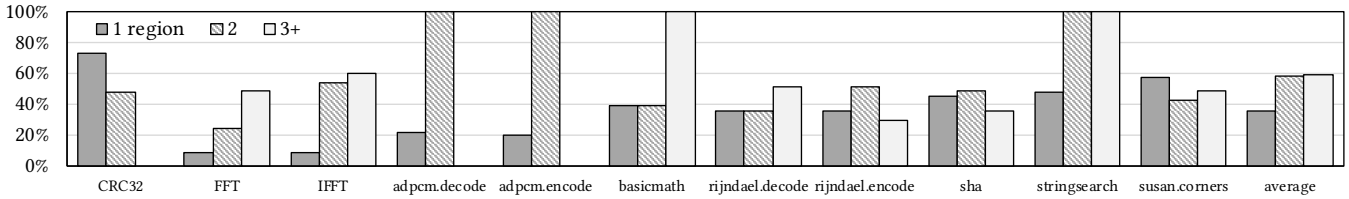



Fig. 10: Reduction on CPU idle time from static prefetching with various number of regions.

the largest performance improvement of 27.2% among the benchmarks even it has one region. This can be explained in two reasons; first, sha has a large performance improvement margin since the CPU idle time before applying prefetching is significant (43.8% in *baseline*). Secondly, most of the heavy functions in sha can be managed by buffer and _a_call, which further improve overlapping execution by exploiting DMA buffer.

Execution times of the benchmarks having less than 10% of CPU idle time – CRC32, adpcm, and susan – have not been improved significantly with our technique. Although their waiting times are reduced by 43.3% on average, execution times are changed within the boundary of 4% (1% improved on average). In adpcm.encode, the performance is reduced by 1.3%; insertion of management functions affects the branch predictors and slightly reduces the instruction-level parallelism.

### D. Scalability Evaluation

We evaluate A-SFP on the various SPM sizes to show its scalability. Figure 10 illustrates the reduced CPU idle time over the different number of regions. We show the results of one- and two-region cases separately and present the average improvement of the cases with more than two regions (labeled as 3+) — most benchmarks have 3∼6 functions (Table I). The results of CRC32 and adpcm with 3+ are omitted since they cannot have more than two regions (have only two functions).

Figure 10 shows that A-SFP effectively reduces DMA synchronization overhead regardless of the SPM size. Especially, two-region and 3+ cases show the largest improvement (58.5 and 59.1%, respectively). All functions in one-region case fall into *conflict* case and thus should be managed by wrapper management functions (_call and _a_call). As discussed in Section III-C, these wrappers not only consist of more management instructions than *normal* case but also incur additional function call. On average, one-region case still shows the reduction of 36.0% in the DMA synchronization overhead.

## V. Conclusion

In SPM-based system, static code prefetching has not been extensively studied since such a system lacks a mechanism to recover from fault (incorrect prefetching). In this paper, we propose a technique to statically inserts DMA instructions to prefetch codes on function-level code management. The algorithm to find safe and efficient prefetch locations is presented along with the two management methods, SFP and A-SFP. Our evaluation shows that our technique can reduce the time waiting on DMAs by 58.5% on average with 2-region configuration and it is scalable with any number of regions. Reduction of waiting time results in the performance improvement of 14.7% on average among the benchmarks which show high CPU idle time. As a future work, we would study a mapping algorithm which is aware of the code prefetching.

## Acknowledgements

This work was partially supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035, Research on High Performance and Scalable Manycore Operating System) and by NRF-2015M3C4A7065522 (Next-generation Information Computing Development Program, NRF, MSIT).

## References

[1] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee, "Sdrm: simultaneous determination of regions and function-to-region mapping for scratchpad memories," in *Proc. of HiPC*, pp. 569–582, Springer, 2008.

[2] K. Bai, J. Lu, A. Shrivastava, and B. Holton, "Cmsm: an efficient and effective code management for software managed multicores," in *Proc. of CODES+ISSS*, p. 11, IEEE Press, 2013.

[3] Y. Kim, J. Cai, Y. Kim, K. Lee, and A. Shrivastava, "Splitting functions in code management on scratchpad memories," in *Proc. of ICCAD*, pp. 1–8, IEEE, 2016.

[4] S. chul Jung, A. Shrivastava, and K. Bai, "Dynamic code mapping for limited local memory systems," in *Proc. of ASAP*, pp. 13–20, IEEE, 2010.

[5] M. A. Baker, A. Panda, N. Ghadge, A. Kadne, and K. S. Chatha, "A performance model and code overlay generator for scratchpad enhanced embedded processors," in *Proc. of CODES+ISSS*, pp. 287–296, ACM, 2010.

[6] C. Jang, J. Lee, B. Egger, and S. Ryu, "Automatic code overlay generation and partially redundant code fetch elimination," *ACM Trans. Archit. Code Optim.(TACO)*, vol. 9, no. 2, p. 10, 2012.

[7] A. Janapsatya, A. Ignjatović, and S. Parameswaran, "A novel instruction scratchpad memory optimization method based on concomitance metric," in *Proc. of ASP-DAC*, pp. 612–617, IEEE Press, 2006.

[8] J. Cai, Y. Kim, Y. Kim, A. Shrivastava, and K. Lee, "Reducing code management overhead in software-managed multicores," in *Proc. of DATE*, pp. 1241–1244, European Design and Automation Association, 2017.

[9] K. Bai and A. Shrivastava, "Automatic and efficient heap data management for limited local memory multicore architectures," in *Proc. of DATE*, pp. 593–598, IEEE, 2013.

[10] B. Egger, S. Kim, C. Jang, J. Lee, S. L. Min, and H. Shin, "Scratchpad memory management techniques for code in embedded systems without an mmu," *IEEE Transactions on Computers*, vol. 59, no. 8, pp. 1047–1062, 2010.

[11] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri, "A post-compiler approach to scratchpad mapping of code," in *Proc. of CASES*, pp. 259–267, ACM, 2004.

[12] L. Li, L. Gao, and J. Xue, "Memory coloring: A compiler approach for scratchpad memory management," in *Proc. of PACT*, pp. 329–338, IEEE, 2005.

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

[14] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. of CASES*, pp. 276–286, ACM, 2003.

[15] P. Chakraborty, P. R. Panda, and S. Sen, "Partitioning and data mapping in reconfigurable cache and scratchpad memory–based architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, p. 12, 2016.

[16] M. R. Soliman and R. Pellizzoni, "Wcet-driven dynamic data scratchpad management with compiler-directed prefetching," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[17] Y. Yang, M. Wang, H. Yan, Z. Shao, and M. Guo, "Dynamic scratchpad memory management with data pipelining for embedded systems," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 13, pp. 1874–1892, 2010.

[18] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *Proc. of HPCA*, pp. 225–236, IEEE, 2005.

[19] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *SIGARCH Comput. Archit. News*, vol. 19, pp. 40–52, ACM, 1991.

[20] C.-K. Luk and T. C. Mowry, "Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors," in *Proc. of MICRO*, pp. 182–194, IEEE Computer Society Press, 1998.

[21] C.-K. Luk and T. C. Mowry, "Architectural and compiler support for effective instruction prefetching: a cooperative approach," *ACM Trans. Comput. Syst.*, vol. 19, no. 1, pp. 71–109, 2001.

[22] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proc. of CGO*, p. 75, IEEE Computer Society, 2004.

[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. of WWC*, pp. 3–14, IEEE, 2001.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[25] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, *et al.*, "The microarchitecture of the synergistic processor for a cell processor," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, pp. 63–70, 2006.