Splitting Functions in Code Management on Scratchpad Memories

Youngbin Kim[1] Jian Cai[2] Yooseong Kim[2]
Kyoungwoo Lee[1] Aviral Shrivastava[2]
[1]Department of Computer Science, Yonsei University, Seoul, Korea
[2]Compiler-Microarchitecture Lab, Arizona State University, Tempe, AZ
[1]{yb.kim,kyoungwoo.lee}@yonsei.ac.kr
[2]{jcai19,yooseong.kim,aviral.shrivastava}@asu.edu

ABSTRACT

As the number of cores increases, cache-based memory hierarchy is becoming a major problem in terms of the scalability and energy consumption. Software-managed scratchpad memories (SPM) is a scalable alternative to caches, but the benefit comes at the cost of explicit management of data. For instance, an instruction SPM needs a code management techniques to load code blocks to the SPM. This paper presents a technique to split functions into smaller functions, to break away with the fundamental limitations of function-level code management. Our function-splitting technique is able to generate more efficient mappings by modifying the characteristics of functions to be more suitable for function-level code management. We propose two optimization policies to improve performance and reduce size respectively. The performance optimization policy improves performance by 16% on average, which can only be achieved by using 20% more SPM space if without function-splitting The size optimization policy can reduce the minimum SPM size requirement by 31% while increasing only 7% execution time.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers, Memory management, Optimization

1. INTRODUCTION

It is becoming more difficult to scale cache-based memory hierarchy in multi-/many-core processors. As a result, many processors use scratchpad memories (SPMs) to avoid the complex and power-hungry circuitry in caches, e.g. tag arrays and lookup logics. While SPMs are more power-efficient than caches [7], the elimination of these hardware components requires SPMs to be managed by software explicitly.

When an SPM is used as an instruction memory, code management techniques [15, 29, 30, 10, 11, 27, 17, 6, 16, 5, 20, 25] are required to load code blocks to the SPM. Particularly, function-level code management [10, 11, 27, 17, 6, 16, 5, 20, 25] uses overlay mechanism to load code blocks at the granularity of functions. Such an approach divides an SPM into regions, each of which is a unique address range, and finds a surjective map of functions to regions. This management scheme is similar to direct-mapped caches. A function is loaded to the SPM as a whole before its execution, by a direct memory access (DMA) operation, only when it is not already loaded. Loading a function into a region always evicts any function currently being in the region.

Loading functions introduces additional overhead to the execution time of the program, for executing a DMA operation and waiting for the DMA operation to be completed. Existing techniques try to generate mapping that will minimize such overhead, e.g. separating the caller and called function of a frequent function call into two regions to prevent them evicting each other at each call. The quality of mapping can significantly affect the performance of applications [27, 17, 6, 16, 5, 20].

Regardless of mapping techniques, the quality of mapping is fundamentally limited by the sizes of functions and their call pattern. For example, consider a function call in which the sum of the sizes of the caller function and the callee function exceeds the SPM size. Any mapping technique would have to map two functions to the same region due to the size limitation. This implies that every time the function call executes, both functions need to be reloaded, as the two functions will replace each other in the SPM every time. This can cause a significant overhead in a loop. On the other hand, asymmetry of function sizes easily results in poor utilization of SPM space. As a region can hold only one function at a time, a large difference between the function sizes within a region translates to severe memory fragmentation.

In this paper, we present a technique that splits functions to improve the characteristics of functions so that mapping algorithms can generate better mappings. We propose two optimization policies: *performance optimization* and *size optimization*. The performance optimization policy enables separating two functions with caller-callee relationship in a loop by splitting out the loop as a separate function. This can improve performance of several benchmarks in our experiments by 16% on average. To achieve a comparable performance without using our function-splitting, we would require 20% larger SPM sizes. On the other hand, the size

ICCAD'16 November 7–10, 2016, Austin, TX
© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

optimization policy focuses on reducing the memory size requirement by cutting the largest functions in half repeatedly. This reduces the minimum SPM size requirement by 31% overall, with only 7% execution time increased.

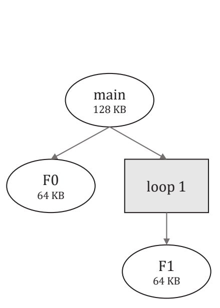
2. RELATED WORK

The work proposed in this paper lies in the context of managing scratchpad memory (SPM). SPM management has been extensively studied in the literature. The earliest approaches statically allocate code [2], data [3], or both [28, 31] to reduce the execution time or energy consumption of a given program. In static management techniques, selected code or data are allocated in the SPM all at once at loading time and stay loaded during execution. This approaches have limitations of not being able to exploit localities of large programs, so SPM management techniques have evolved into dynamic management. Dynamic management techniques update the contents of the SPM in run-time to cater different parts in large programs at a time. While there are dynamic techniques for managing data [18, 32], our technique is in the context of *dynamically loading code blocks* [15, 29, 30], especially at the granularity of *functions* [10, 11, 27, 17, 6, 16, 5, 20, 25]. Our function-splitting technique helps function-level management techniques utilize a given SPM space more efficiently (with smaller code blocks and less fragmentation) and overcome its limitation of requiring the largest function to fit in the SPM (by breaking the largest function into smaller pieces).

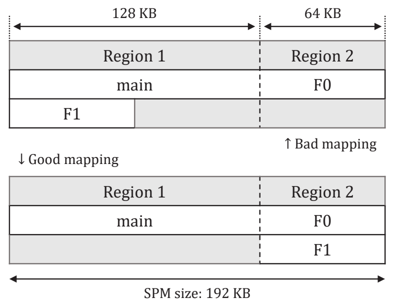
Function-splitting is similar to existing approaches in compiler optimization, called *function outlining* or *partial inlining* [34, 22, 26]. These techniques make part of a function form another function (outlining) so that only the selected part of the function is inlined (partial inlining) reducing the code size expansion. Outlining techniques typically try to extract a rarely-executed part into another function in order to reduce the overhead of calling outlined functions. In contrast, our function-splitting technique aims either to reduce the memory footprints of functions during the execution of loops, or to make function sizes as small as possible. Unlike partial inlining techniques, our technique does not increase the code sizes of the caller functions.

Function-splitting can be seen as a form of a *graph partitioning* problem. Traditionally, graph partitioning is studied in the context of non-iterative data-flow analysis [1] or parallel data-flow analysis [23, 24]. These approaches find single-entry regions, called intervals or regions, in a control flow graph (CFG) of a function. Even earlier than these approaches, in 70s, the graph partitioning problem has been studied for page overlay managers. Several approaches have been proposed to divide a given CFG into subgraphs [4, 19], each of which is then mapped to a memory page.

More recently, the graph partitioning problem is studied in the context of on-chip memory management. Whitham and Audsley [33] presented an optimal graph partitioning technique for instruction SPM, with a goal of minimizing the worst-case execution time (WCET) of a given program as well as keeping the partition sizes smaller than the SPM size. Hepp and Brandner [14] proposed a function-splitting technique for *Method cache* [9], a special type of instruction cache that loads at the granularity of variable-sized code regions rather than cache blocks of a fixed size. Their objective is to keep the sizes of partitions smaller than a specified size. As graph partitioning techniques, all these approaches are based on CFG traversals, whereas our approach performs a source-to-source transformation implemented as a compiler front-end stage. This is intended to be used with function-level code management techniques that use function overlay mechanism. This difference makes our approach actually create new functions and insert function calls to those newly created functions in the source code. In contrast, in the previous techniques for on-chip memory management [33, 14] the created partitions are still part of the existing function; branches at partition boundaries need to be modified to maintain the original control flow even when the partitions are loaded to nonconsecutive memory ranges.



(a) Graph representation of a program.



(b) Two possible mappings, when the SPM size is 192 KB.

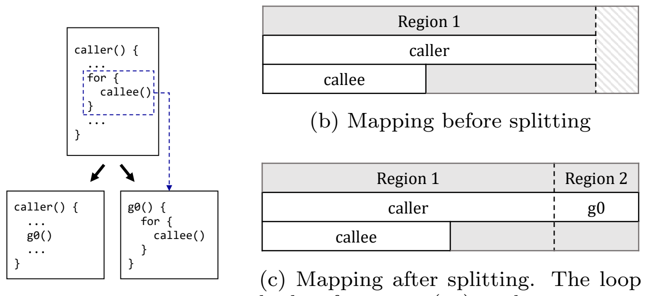
Figure 1: Given a program in (a), there are two possible mappings shown in (b).

This paper is the first approach to break away with the function-level granularity in such code management techniques. We present two heuristics built on insights either to increase the memory space utilization or to overcome the limitations of function-level code management.

3. FUNCTION-LEVEL CODE MANAGEMENT

Function-level code management uses overlaying that uses function-to-region mapping [27] to decide where to load each function of a given program. This mapping can significantly affect the execution time of the program. Figure 1 shows an example. In the given graph, the `main` function calls both `F0` and `F1` function. The function call of `F1` is called in `loop1`, which iterates multiple times. Given the function sizes and SPM size, there can be different mappings, both of which are shown in the figure.

The overhead of a mapping is measured by *interference*, which is the total bytes of data transferred caused by the contention over an SPM space among functions that are mapped to the same region. If we assume `loop1` in Figure 1a iterates 5 times, and `main` first calls `F0` before it executes `loop1`, then the control flow proceeds in the following order: `main, F0, main, F1, main, F1, main, F1, main, F1, main, F1, main`. The interference of the first mapping (the one on the top in Figure 1b) is therefore 128 + 64 + (64 + 128) * 5 = 1152 KB. The mapping maps `main` and `F1` function to the same region. As a result, they will evict each other at every iteration of `loop1`, which consists of most of the interference. On the other hand, the second mapping (the one on the bottom in Figure 1b) maps `main` and `F1` into two different regions. This mapping thus successfully avoids the two functions alternatively evicting each other in


```
caller() {
  ...
  for {
    callee()
  }
  ...
}
```
```
caller() {
  ...
  g0()
  ...
}
```
```
g0() {
  for {
    callee()
  }
}
```

(a) Example of function-splitting

(b) Mapping before splitting

(c) Mapping after splitting. The loop body of `caller` (`g0`) and `callee` are separated, removing the interference between them.

Figure 2: Splitting `caller` enables a better mapping shown in (c), in which `g0` and `callee` can remain loaded during the execution of the loop.

Algorithm 1 Performance Optimization

```
Require:
    $S_{spm}$: Available SPM size

 1: $M' \leftarrow$ mapping before function splitting
 2: $C' \leftarrow$ code management overhead caused by M'
 3: $L \leftarrow$ All the loops that contain function calls
 4: while $L$ is not empty do
 5:     $l \leftarrow$ SELECT($L$)
 6:     Extract $l$ as a new function from its parent function
 7:     $S \leftarrow$ The overall size of regions after splitting
 8:     $M \leftarrow$ mapping after function splitting
 9:     $C \leftarrow$ code management overhead caused by $M$
10:     if $S > S_{spm}$ or $C <= C'$ then
11:         Return $M'$
12:     end if
13:     $M' \leftarrow M$
14:     $C' \leftarrow C$
15: end while

16: function SELECT($L$)
17:     $minCost \leftarrow \min_{l \in L}$ COST($l$)
18:     $L' \leftarrow \{l \mid l \in L,$ COST($l$) $== minCost\}$
19:     return the smallest $l \in L'$
20: end function

21: function COST($l$)
22:     $f \leftarrow$ The parent function of $l$
23:     $L' \leftarrow \{x \mid x \in L$ and $f$ is called in loop $x\}$
24:     if $|L'| = 0$ then
25:         return 0
26:     else
27:         return $\max_{x \in L'}$ COST($x$)+1
28:     end if
29: end function
```

each iteration of `loop1`. The interference becomes 128 + 64 + 128 + 64 = 384 KB. Therefore, the second mapping is preferred to the first one for incurring the less overhead of code management.

4. OUR APPROACH

In this section, we propose a technique that enables a generation of better mapping by splitting functions. It take two steps to split a function: i) choose program points for splitting, ii) create new functions based on the chosen program points. Depending on the purpose of splitting functions (either for improving performance or enabling code management), the first step may vary. Once program points are given after the first step, we extract the specified code and create new functions accordingly. Function calls to the new functions are then inserted in the original caller functions.

We present two different policies of choosing program points for splitting in section 4.1: *performance optimization* and *size optimization*. The former is used to improve performance by splitting functions to better exploit SPM space and get a more efficient overlaying scheme, while the latter is used to enable code management when the SPM space is not large enough to accommodate the largest function. Section 4.2 then explains given the identified program points, how to create new functions while keeping the program semantics unchanged.

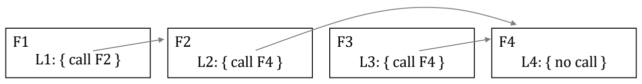
4.1 Where-to-split: Choosing Splitting Points

4.1.1 Performance Optimization

The overhead of code management happens around each function call. Before the function call, the code management instructions should check if the called function is in its region, and load the function if it is not loaded already. After the called function returns, the code management instructions should also check if the caller is still loaded and load it in case it is evicted during the function call. As a result, the more frequently a callsite is executed, the more code management overhead it incurs. We, therefore, target on function calls within loops, and try to extract the loops as new functions whenever profitable to performance.

Figure 2 shows how splitting functions help reduce code management overhead. In Figure 2a, function `caller` calls function `callee` in a loop. Originally, SPM space is barely large enough to map both functions to the same region `Region 1`, as Figure 2b shows. As a result, every time before the function call happens, the `callee` function must be loaded into `Region 1` in the SPM and evicts `caller` (which must have been loaded into SPM earlier before it is called). Right after the function call returns, `caller` must be loaded into `Region 1` to resume its execution, which in turn evicts `callee`. Such code management overhead happens in every iteration of the loop. If we can single out the in `caller` and make it a new function, then we can generate a new mapping as Figure 2c shows. The new function, `g0`, which the loop now resides in, are mapped to a different region with the `callee` function. While the splitting introduces instructions for the extra function call (the call to `g0`), such mapping avoids the code management overhead caused by the repeated eviction of `caller` and `caller` before `caller` is split, and thus can be used to reduce number of memory transfers and eventually improve performance of applications.

Algorithm 1 shows the algorithm we use for splitting functions with performance optimization policy. The algorithm first generates a mapping M' given the SPM size S_{spm} (line 1), and estimates its code management cost as C' (line 2). The generation of mapping and the estimation of its overhead are carried out using the state-of-the-art approach from



	L1	L2	L3	L4
L'	∅	{ L1 }	∅	{ L2, L3 }
COST(Ln)	0	1	0	2

Figure 3: Cost of extracting a loop is related to its maximum possible nesting level.

[25]. The algorithm collects all the loops that contain function calls in a set L (line 3). It then singles out an individual loop of the program (line 5), and split the parent function of l into two functions—a new function that contains l only, and another function that contains the leftover of the parent function with a call to the new function in place of the loop (line 6). Once the function splitting is done, a new mapping M is generated (line 8), as well as the estimation of the its overhead C (line 9), again using the approach from [25]. If any of the following two conditions is met, then the algorithm stops: i) C is smaller or equal to C', or ii) S is greater than S_{spm} (line 10). The first condition implies the overhead is not reduced by the new mapping, while the second condition implies the available space of SPM is used up. If neither of the above conditions are met, then the algorithm substitutes the new mapping M and its overhead C for the old mapping M' and C' respectively, and continue the algorithm until either of the two condition is true.

The order of loops selected in line 5 in Algorithm 1 is specified by `SELECT` function defined in line 14. Given a set of loops L, `SELECT(`L`)` finds the loop that incur the minimum cost to the program (line 15). When a loop is extracted as a new function, a function call to the new function must be inserted in place of the loop in its original parent function. This will incur extra instructions due to function call overhead and code management for the inserted function call. Therefore, these instructions should be avoided to the greatest extent. Function `COST` defined in line 18 measures the cost of loop by estimating the maximum possible nesting levels of the loop. The deeper the loop, the higher the cost. Figure 3 shows an example of using `COST` function to measure the cost of creating a new function out of a loop. The loop `L1` and `L3` are not nested within any other loop, so the cost of them is measured as 0. The parent function of `L2`, `F2`, is called within `L1` in `F1`. Therefore, its cost is measured as 1. Finally, since `L4` can be either executed within `L3`, or within `L2`, which is further nested within `L1`. The maximum possible depth is therefore 2, which is reflected in its cost.

4.1.2 *Size Optimization*

None of the previous function-level code management is able to generate any code mapping when the size of SPM is smaller than the largest function, since the largest function cannot be placed into the SPM. Size optimization policy aims to enable code mapping in this case, while incurring as less overhead as possible.

Algorithm 2 shows the algorithm we use for splitting functions with size optimization policy. In each iteration, we select the current largest function (line 5) and split it using function `SPLIT` (line 6). If the overall size of regions after function splitting is not decreased, then the algorithm fails to find any mapping with overall size of regions less than

Algorithm 2 Size Optimization

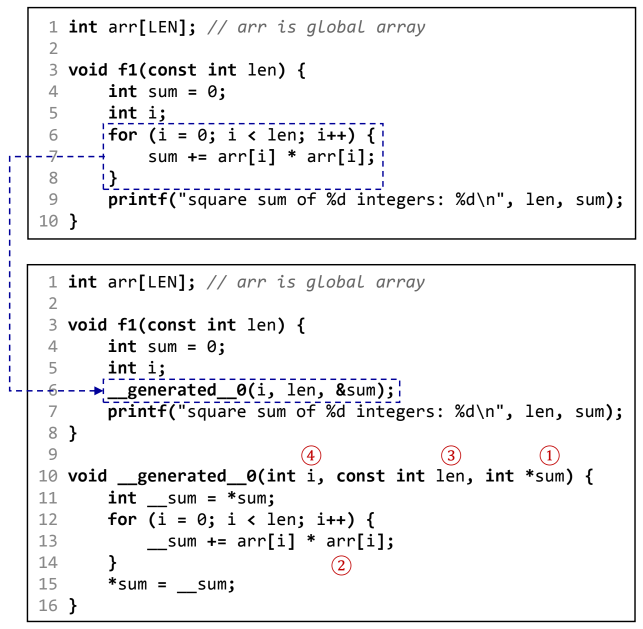
```
Require:
    $S_{spm}$ : Available SPM size
    $S_{param} \leftarrow$ constant for estimating size overhead of parameter passing

 1: $M' \leftarrow$ mapping before function splitting
 2: $S' \leftarrow$ total size used by regions before splitting
 3: $F \leftarrow$ all the functions in the program
 4: while $S' > S_{spm}$ do
 5:     $f \leftarrow$ extract the largest function in $F$
 6:     SPLIT($f$)
 7:     $S \leftarrow$ the overall size of regions after function splitting
 8:     $M \leftarrow$ mapping after function splitting
 9:     if $S > S'$ then
10:         Fail to find a solution
11:     end if
12:     $M' \leftarrow M$
13:     $S' \leftarrow S$
14: end while
15: Return $M'$

16: function SPLIT($f$)
17:     $n \leftarrow$ the number of instruction in $f$
18:     $min \leftarrow$ size of $f$
19:     $k' \leftarrow 0$
20:     Let $i_k$ be the $k$th instruction of $f$
21:     for $k$ from 2 to $n$ do
22:         $F_0, F_1 \leftarrow \{i_1, ..., i_k\}, \{i_{k+1}, ..., i_n\}$
23:         $P \leftarrow$ parameters of $F_1$
24:         $S_0, S_1 \leftarrow$ size of $F_0$, size of $F_1$
25:         $newSize_k \leftarrow \max(S_0, S_1) + |P| \cdot S_{param}$
26:         if $newSize_k > min$ then
27:             $k' \leftarrow k$
28:             $min \leftarrow newSize_k$
29:         end if
30:     end for
31:     split $f$ into $\{i_1, ..., i_{k'-1}\}$ and $\{i'_k, ..., i_n\}$
32:     Update $F$
33: end function
```

the available SPM size (line 9-10). Otherwise, we return the final mapping when the required SPM size falls below the available SPM size.

The `SPLIT` function defined in line 14 finds the program points to split in the given function f. For each instruction i_k in f, the `SPLIT` function estimates the change of required SPM size after splitting the given function f into two smaller functions, F_0 and F_1, at the program point right after i_k (line 20). Since the size of required SPM space is decided by the overall size of all the regions, and the size of each region is decided by the largest function mapped to the region, we use the sum of i) the maximum size of F_0 and F_1 and ii) the number of parameters/arguments required by F_1 to estimate the change of required SPM space after function splitting (line 23). Lastly, the function splitting that results in the minimum SPM space are chosen (line 27).

4.2 Creating New Functions

Once program points are chosen for function splitting, the code blocks specified by the program points are extracted from the original functions and new functions are created



```
int arr[LEN]; // arr is global array

void f1(const int len) {
    int sum = 0;
    int i;
    for (i = 0; i < len; i++) {
        sum += arr[i] * arr[i];
    }
    printf("square sum of %d integers: %d\n", len, sum);
}
```

```
int arr[LEN]; // arr is global array

void f1(const int len) {
    int sum = 0;
    int i;
    __generated__0(i, len, &sum);
    printf("square sum of %d integers: %d\n", len, sum);
}

void __generated__0(int i, const int len, int *sum) {
    int __sum = *sum;
    for (i = 0; i < len; i++) {
        __sum += arr[i] * arr[i];
    }
    *sum = __sum;
}
```

Figure 4: Creating a new function given the program points to split.

with these code blocks. Function calls to the new functions are created to call the new functions in the original functions.

Figure 4 shows such an example. Assume the given program points specify that the code block of the loop in the dotted box should be split from function `f1`. A new function `__generated__0` is then created enclosing the loop. A call to `__generated__0` is inserted in `f1` to replace of the code of the loop. The new function takes as input all the variables that it accesses (all the variables accessed in the loop in the original code of `f1`). However, depending on the characteristics of their accesses, these variables are passed as parameters in different way. Any dead variable (④) at the exit of `__generated__0` is passed by value. A variable is live at a program point if after the program point it may be read prior to any writes of the variable. Otherwise, the variable is dead at the program point, either because that its value is not read or overwritten by another write before it is read. Therefore, if a variable is dead when `__generated__0` returns, its value at the time of return will not be used and can be safely discarded. Similarly, any variable whose value (③) will not be modified in `__generated__0` are passed by value, since its value is anyway the same before and after `__generated__0` is called. If a variable is live at the exit and is modified (①), then a reference to its address is passed instead, since the changes must be visible to the reads after `__generated__0` returns by definition of live variables. In particular, since global variables can be accessed in any function directly, they do not need to be passed explicitly as parameters, e.g. the global array **`arr`** (②).

5. EVALUATION

5.1 Experimental Setup

We implement our function splitting technique in Clang, a C front-end for LLVM compiler infrastructure [21]. We compile benchmarks from MiBench benchmark suite [13] without and with function splitting respectively (both with `O3` optimization flag enabled), on top of the state-of-the-art mapping technique CMSM [25]. We then run the generated binaries and collect the execution statistics on gem5 simulator [8].

We built the SPM as a memory-mapped region in gem5. The DMA operation is implemented to transfer data between the SPM and the main memory. The overhead of each DMA operation consists of two parts: setup time and transfer time. Setup time is the constant time required whenever the DMA request is initiated. Transfer time is the additional time proportional to the size of data transferred. We set 91 nanosecond as the setup time, and 0.075 nanosecond per byte as the transfer rate. CPU clock frequency is set to 3.2 Ghz. The setup time and transfer rate are about 291 and 0.24 CPU cycles respectively. This setup models the IBM Cell BE processor [12], which is an SPM-based multicore architecture.

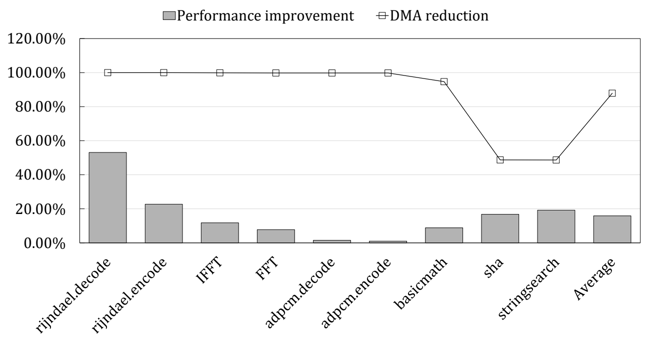


Figure 5: Size and performance of benchmarks with *perf* when compared to using *1-region* configuration.

5.2 Performance Optimization

5.2.1 *Comparisons with Typical Configurations*

We evaluated the performance optimization policy by comparing the performance of benchmarks before and after applying function splitting with performance optimization policy. We first measured performance of benchmarks with code mapping generated by CMSM before function splitting under two set of SPM sizes. For ease of discussion, we refer the two configurations as *1-region* and *2-regions* respectively. In *1-region*, the SPM size for each benchmark is equal to the code size of the largest function, so that CMSM maps all the functions to one region. In *2-regions* configuration, we increase the SPM size so that CMSM will generate two regions. We then measure the performance of the same benchmarks with code mapping generated by CMSM after applying function splitting with performance optimization policy. The sizes of the SPM are set to be large enough that there will be two regions. We refer the configuration as *perf*.

Figure 5 shows the comparison of performance between *1-region* and *perf*. With the function splitting with performance optimization policy, performance of benchmarks is improved by 16% performance. The improvement comes from the reduced communication caused by DMAs. As more regions are generated after the function splitting, less functions are mapped to each region on average, which effectively abates the competition of each region. Benchmarks such

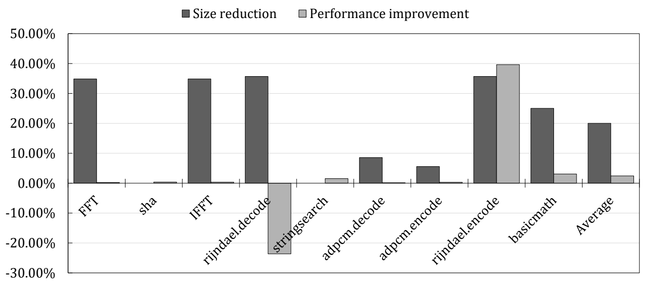


Figure 6: Size and performance comparison between *perf* and *2-regions*.

as `rijndael.decode` achieves great improvement of performance. These benchmarks follow the same pattern of execution: DMAs that are issued by calling a large function repeatedly in a small loop contributes a large portion of execution time. The number of DMAs (or the overhead of DMAs) is then greatly reduced by splitting after the called function and the new function—extracted from the loop—are mapped into separate regions. In benchmarks such as `adpcm.decode` and `adpcm.encode`, however, computation contributes to most of its execution time, rather than communication. In these cases, the DMA overhead only contributes a slight portion, therefore the improvement due to function splitting is insignificant. However, as the dotted line in Figure 5 shows, DMA is almost eliminated in all benchmarks, including those with less significant improvement of performance.

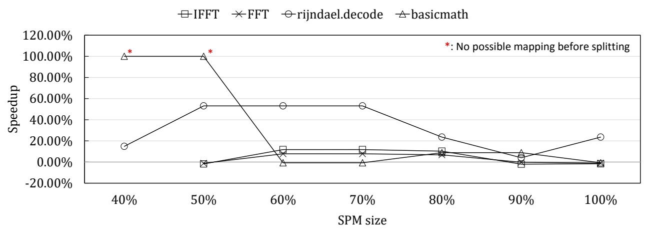
While the performance is improved, *perf* requires more SPM space than *1-region* on average. This is because *1-region* maps all the functions in one region. The required SPM size is equal to the code size of the largest function. On the other hand, *perf* have two regions, with one of the regions requires as much space as *1-region* to hold the largest function (if it is not the function to split). However, as Figure 6 shows, when being compared with *2-regions*, *perf* is able to reduce SPM size requirement by 20%, while still improving performance by 2.4% on average, up to more than 39%.

Benefits from performance optimization varies depending on applications. Benchmarks such as `(I)FFT` and `adpcms` show size reduction with negligible overhead since the mapping algorithm generated optimal mapping by exploiting split function. On the other hand, in `basicmath`, function splitting gives more choice to trade-off SPM size and performance. In this case, function splitting expands design space by generating configuration giving slightly worse performance with 25% less SPM usage, compared to `2-regions`.

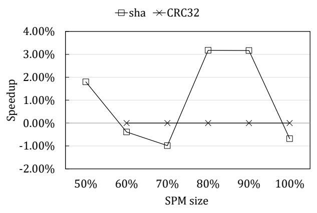
5.2.2 Scalability

Figure 7 shows the performance improvement of the optimization when the SPM size varies. The SPM size is given as a ratio to the total code size of each benchmark. As depicted in the figure, most benchmarks experience the performance improvement especially when the SPM size is small. As the SPM size increases, the improvement slows down. This is natural since the optimization gives performance improvement by relaxing the size requirement to avoid frequent conflicts. When the SPM size is large enough, high-quality mapping can be generated without function splitting.

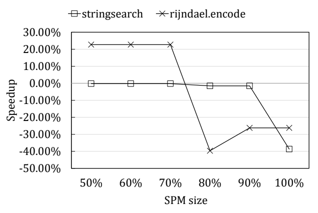
Benchmarks show different sensitivity depending on their



(a) Function splitting is beneficial for all different sizes. `basicmath` also enables mapping with a smaller SPM.



(b) Impact of the splitting is small and not sensitive to the SPM size



(c) Benefit of splitting is dependent on the SPM size

Figure 7: Performance improvement from function splitting when various SPM sizes are given. The size in x axis represents the ratio to the total code size.

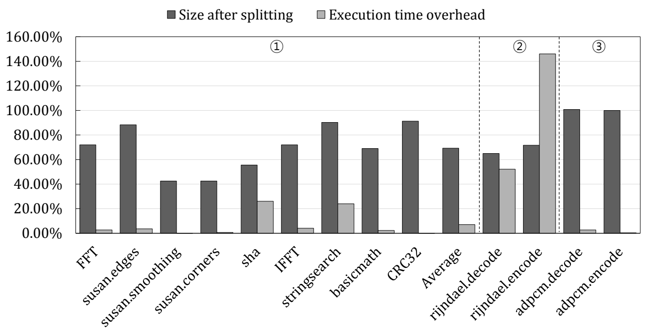


Figure 8: Size and performance overhead after splitting using size optimization policy when desired number of regions is 1.

characteristics. Benchmarks depicted in Figure 7a show improved performance with all SPM sizes. On the other hand, benchmarks in Figure 7b are generally not sensitive to the SPM size and function splitting has marginal effects on them. A third class of benchmarks in Figure 7c shows the varying performance improvement as the SPM size increases. In these benchmarks, function splitting is beneficial when the SPM size is small, but its overhead restricts generation of good mappings when the SPM size is large enough.

5.3 Size Optimization

Figure 8 shows the result of function splitting using size optimization policy. We compared the size and performance before and after splitting, when the desired number of regions is 1. The results show the average code size reduction of 31% with 7% performance overhead. Performance overhead is 2% on average for all the benchmarks excluding `sha` and `stringsearch`.

We can classify the benchmarks into 3 categories. Each categories is shown in Figure 8 by number. Benchmarks in

① ((I)FFT, susans, sha, stringsearch, adpcms, basicmath and CRC32) have the most suitable structure for the size optimization. In these benchmarks, large functions have loops which call smaller function frequently. Therefore, splitting large function successfully reduces the requirement of SPM size but does not introduce many additional function calls. This policy is also applicable to benchmarks that does not many function call, such as susans.

On the other hand, two rijndaels in ② show high performance overhead. This comes from their application structure, where small main driver function calls large computation function in a loop. They show high reduction in SPM size since the function sizes are skewed, but also have high performance overhead. By splitting large function which is called frequently, the number of function call and management overhead from this function increases.

Benchmarks in ③, adpcms, cannot take advantage of the splitting. In these cases, there are no large enough functions to split. Splitting function will actually increase the function sizes, due to the extra instructions and temporary variables introduced for new function calls.

6. CONCLUSION

In function-level code management, the quality of function-to-region mapping determines performance of a program. Although many researcher have put significant efforts into finding efficient mappings to improve performance, the quality of mapping is limited by the sizes of functions and their call patterns. This paper presents a technique to split functions into smaller functions to break away from this fundamental limitation of function-level code management. We present two different optimization policies; while the performance optimization policy tries to minimize management overhead, the size optimization policy focuses on reducing the minimum SPM size requirement of a program.

In our experiments with several benchmarks, we observed that using our function-splitting technique is effective in improving performance and reducing the size requirement. With performance optimization policy, execution time can be reduced by 16% on average. To achieve a comparable performance without function-splitting, SPM sizes have to be 20% larger. On the other hand, the size optimization policy can reduce the SPM size by 31% from the minimum SPM requirement before splitting, increasing the execution time only by 7% on average.

We will extend this work by developing an optimal function-splitting scheme that can find optimal splitting points for performance while keeping the function sizes within any given SPM size.

7. ACKNOWLEDGMENTS

This work was partially supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. B0101-16-0644, Research Project on High Performance and Scalable Manycore Operating System), and by funding from National Science Foundation grants CCF 1055094 (CAREER), and CNS 1525855.

8. REFERENCES

[1] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, Mar 1976.

[2] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A Post-compiler Approach to Scratchpad Mapping of Code. In *Proc. of CASES*, pages 259–267, 2004.

[3] O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, Nov 2002.

[4] J.-L. Baer and R. Caughey. Segmentation and Optimization of Programs from Cyclic Structure Analysis. In *Proc. of AFIPS*, pages 23–36, 1972.

[5] 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*, pages 1–9, 2013.

[6] 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*, pages 287–296, 2010.

[7] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proc. of CODES*, pages 73–78, 2002.

[8] 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. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[9] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl. A Method Cache for Patmos. In *Proc. of ISORC*, pages 100–108, 2014.

[10] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization. In *Proc. of CASES*, pages 223–233, 2006.

[11] 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*, 59(8):1047–1062, Aug 2010.

[12] 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*, 41(1):63–70, 2006.

[13] 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*, pages 3–14, 2001.

[14] S. Hepp and F. Brandner. Splitting Functions into Single-entry Regions. In *Proc. of CASES*, pages 17:1–17:10, 2014.

[15] A. Janapsatya, A. Ignjatović, and S. Parameswaran. A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric. In *Proc. of ASPDAC*, pages 612–617, 2006.

[16] C. Jang, J. Lee, B. Egger, and S. Ryu. Automatic code overlay generation and partially redundant code fetch elimination. *ACM Trans. Archit. Code Optim.*, 9(2):10, 2012.

[17] S. C. Jung, A. Shrivastava, and K. Bai. Dynamic code mapping for limited local memory systems. In *Proc. of ASAP*, pages 13–20, 2010.

[18] M. Kandemir and A. Choudhary. Compiler-directed Scratch Pad Memory Hierarchy Design and Management. In *Proc. of DAC*, pages 628–633, 2002.

[19] B. W. Kernighan. Optimal Sequential Partitions of Graphs. *J. ACM*, 18(1):34–40, Jan 1971.

[20] Y. Kim, D. Broman, J. Cai, and A. Shrivastava. WCET-aware dynamic code management on scratchpads for Software-Managed Multicores. In *Proc. of RTAS*, pages 179–188, 2014.

[21] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. of CGO*, pages 75–86, 2004.

[22] J. P. Lee, J. J. Kim, S. M. Moon, and S. Kim. Aggressive Function Splitting for Partial Inlining. In *Proc. of INTERACT*, pages 80–86, 2011.

[23] Y.-F. Lee and B. G. Ryder. A Comprehensive Approach to Parallel Data Flow Analysis. In *Proc. of ICS*, pages 236–247, 1992.

[24] Y.-F. Lee, B. G. Ryder, and M. E. Fiuczynski. Region Analysis: A Parallel Elimination Method for Data Flow Analysis. *IEEE Trans. Softw. Eng.*, 21(11):913–926, Nov 1995.

[25] J. Lu, K. Bai, and A. Shrivastava. Efficient Code Assignment Techniques for Local Memory on Software Managed Multicores. *ACM Trans. Embed. Comput. Syst.*, 14(4):1–24, Dec 2015.

[26] R. Muth and S. Debray. Partial Inlining. Technical report, Department of Computer Science, University of Arizona, 1997.

[27] 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*, pages 569–582. 2008.

[28] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. of DATE*, pages 409–415, 2002.

[29] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

[30] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE TVLSI*, 14(8):802–815, Aug 2006.

[31] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *Proc. of DATE*, 2004.

[32] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *Proc. of CODES+ISSS*, pages 104–109, 2004.

[33] J. Whitham and N. Audsley. Optimal program partitioning for predictable performance. In *Proc. of ECRTS*, pages 122–131, 2012.

[34] P. Zhao and J. N. Amaral. Ablego: A Function Outlining and Partial Inlining Framework: Research Articles. *Softw. Pract. Exper.*, 37(5):465–491, Apr 2007.