Reducing Code Management Overhead in Software-Managed Multicores

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

***Abstract*—Software-managed architectures, which use scratchpad memories (SPMs), are a promising alternative to cached-based architectures for multicores. SPMs provide scalability but require explicit management. For example, to use an instruction SPM, explicit management code needs to be inserted around every call site to load functions to the SPM. such management code would check the state of the SPM and perform loading operations if necessary, which can cause considerable overhead at runtime. In this paper, we propose a compiler-based approach to reduce this overhead by identifying management code that can be removed or simplified. Our experiments with various benchmarks show that our approach reduces the execution time by 14% on average. In addition, compared to hardware caching, using our approach on an SPM-based architecture can reduce the execution times of the benchmarks by up to 15%.**

I. Introduction

Using scratchpad memories (SPMs) instead of caches can considerably reduce power and area overhead [1]. The simplified hardware, however, shifts the work of memory management from hardware to software and requires executing additional management instructions in software. Multicore architectures based on SPMs are, therefore, called software-managed multicore (SMM) architectures, where each core has its local SPM. Instructions or data can be transferred into an SPM by direct memory access (DMA) operations.

One way to manage instructions on an SPM is overlaying [2]. Overlaying divides SPM space into different regions, with each function allocated to one of the region. Before every function call, a management function must be called to check the SPM state to see if the called function is loaded in the SPM and if not, performs a DMA operation. Similarly, the management function needs to be called again right before the called function returns back to the caller, because the caller function might have been evicted by the called function, in case they share the SPM space.

There are *two sources of overhead* in such code management. The *long-latency DMA operations* are one source of overhead. The SPM allocation determines the memory space sharing among functions, and a poor allocation scheme can increase the overhead of DMA operations by causing frequent reloading of functions. Another source of overhead, which is of our interest in this paper, comes with *calling the management function* at every call site. This can cause a noticeable overhead as merely checking the SPM state involves

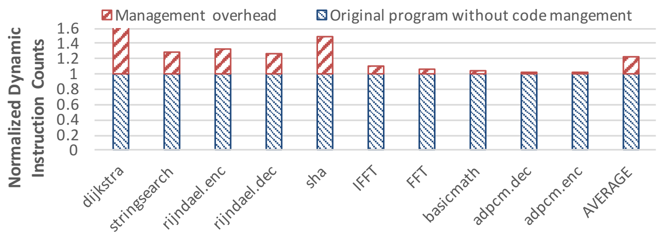

Fig. 1: The overhead of management function calls is manifested as the increase in the dynamic instruction counts.

multiple table lookups to obtain the information about the called function (address, size, load status, etc.). If loading the called function turns out to be indeed required, there will be additional overhead for updating the SPM state.

Most previous overlaying-based code management approaches are focused on the first overhead regarding DMA operations; they try to find memory allocation schemes that can minimize function reloadings [3], [4], [5], [6], [7]. For example, allocating a caller and a called function in a loop to non-overlapping memory ranges eliminates the competition for SPM space between them. Kim *et al.* [8] proposed a technique to split functions into smaller partitions to facilitate finding efficient allocation schemes. These approaches, however, do not address the second overhead (regarding management overhead) and blindly insert management function calls to every call site even though some of them may not be necessary. For instance, even if a function is loaded into its private space in the SPM, a management function has to be called every time the function is called, only to find out that the function is already loaded in the SPM. Figure 1 shows such overhead of executing code management instructions *in vain*. It shows that over a set of typical embedded applications [9], 18% of executed instructions on average are from management functions. Note that this is the *lower bounds* of the overhead when the SPM size is large enough to assign a private region to every function. For smaller SPM sizes, the number of loading operations will increase as the conflicts between functions increases, thus the overhead of executing management instructions will also increase.

In this paper, we propose a compiler-based approach to reduce the overhead of management function calls. Given the allocation of functions to the SPM, our analysis statically

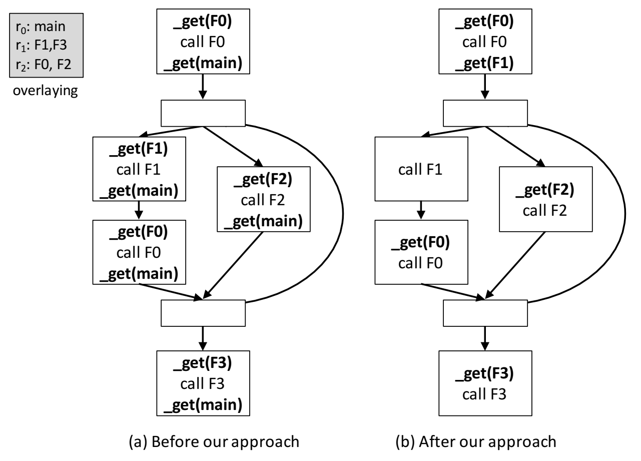

Fig. 2: Removing unnecessary management function calls.

determines whether a function can be safely assumed to be loaded before each call site, and whether the caller is always loaded after the call site. If a function is guaranteed to be already loaded at a call site, we label the call site as *always-hit* and do not insert a management function call. Similarly, a call site in a loop is labeled as *first-miss*, if the called function cannot be guaranteed to be always loaded but once loaded, is never evicted until the end of the loop. In this case, we hoist the management function call to the loop preheader so that it is executed only once before the loop.

The static instruction cache analysis based on abstract interpretation [10], [11] also tries to identify always-hit and first-miss cache lines. However, within nested loops, their first-miss analysis [11] is only able to identify cache lines that are first-miss in the outermost loop. On the other hand, our first-miss analysis can identify any first-miss call site including those that are first-miss only within inner loops. In addition, our solution consists of two steps: i) static analysis for finding always-hits and first-misses; ii) inserting management function calls based on the analysis result. The cache analysis techniques only deal with the first part and cannot be directly used for reducing overhead of code management on SPM.

For evaluation, we use the state-of-the-art function allocation technique [7], and various benchmarks from Mibench suite [9], with varying SPM sizes. The results show that our approach reduces execution time by 14% on average. In addition, our approach reduces the execution time by 9% on average and up to 15% compared to hardware caching, even with conservative measurement.

II. Motivating Example

Figure 2 shows that current code management techniques may insert unnecessary management function calls, and how we can avoid them. The code management function, referred as `_get` in the rest of our discussion, is inserted around each function call. The `_get` function checks if the required function is currently in the region it is allocated to. If not, it loads the function into the SPM. The SPM space is divided into three regions r_0, r_1 and r_2. Functions are allocated to the regions respectively as follow: {`main`}, {`F1,F3`}, {`F0,F2`}. Previous code management approaches insert code management functions around every function call as in Figure 2(a). In contrast, our analyses enables us to remove some of the management function calls as shown in Figure 2(b). For example, throughout the execution, `main` will never be evicted since it is the only function mapped to r_0, so none of the calls of `_get(main)` after each function call is necessary. Also, since we know `F1` is the only function called within the loop in region r_1, it will not be evicted after it is loaded into the region for the first time. On the other hand, the `_get` function called before each call to `F0` and `F2` in the loop are required, since the exact order of execution is not know at compile-time, so we have to conservatively assume either of them may be evicted in previous iterations from r_2.

III. Our Approach

The flow of our approach is as follows: the compiler takes as input a program, and generates a control flow graph (CFG) for each function. All the CFGs, as well as the mapping between functions and regions, are then fed as input to our analyses. The output of our analyses are as follow: i) before each function call, whether the called function is always-hit/first-miss; ii) after the function call, whether the caller function is always-hit/first-miss. The result is then used to insert *only necessary* management function calls accordingly.

Our analyses are a type of classic forward iterative data-flow analyses. The always-hit analysis is performed on the CFG of a whole program whereas the first-hit analysis is done on each individual loop. We associate a data-flow value with each program point before and after each statement respectively. A data-flow value records the current state of each SPM region. If a statement is a function call, we save an additional intermediate state that records the SPM state right before the call returns (so we can know if the caller has been evicted and needs to be brought back). Whenever a function call happens, the called function replaces the existing function in the region in which the called function is allocated. Thus, at any time, there is only one function in each region of a data-flow value. When we have to combine data-flow values from multiple paths, we perform a join operation based on a *meet* operator, defined later for each analysis. Like any other iterative data-flow analyses, the analysis stops when SPM states are no longer updated.

A. Always-hit Analysis

We explain always-hit analysis with an example shown in Figure 3. The output/input SPM state of the incoming/outgoing basic block is shown in each edge. The intermediate SPM states right before function calls return are not shown, since they are easy to figure out based on other states. The prefix `f` denotes the states in the first iteration, `o` denotes the states in the rest of iterations, and `f/o` when they are the same. In the first iteration, we ignore the data-flow value from back edges, since the source of a back edge has not been visited when the

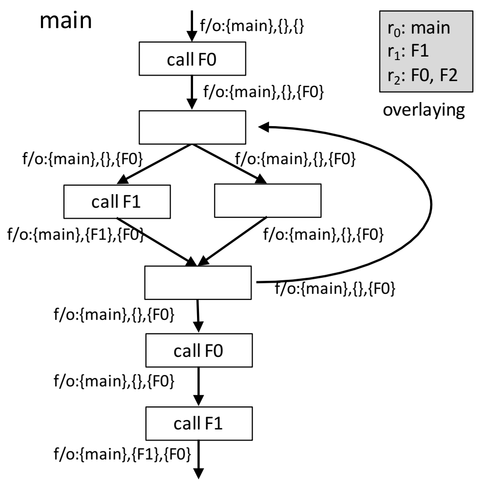

Fig. 3: Always-hit analysis in the *main* function.

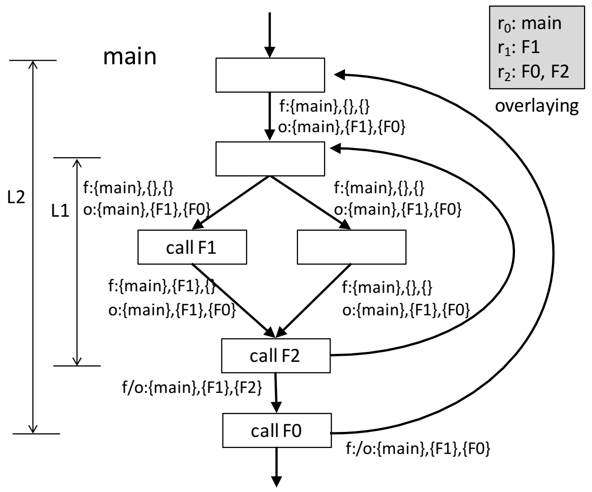

Fig. 4: First-miss analysis in loop `L2` of the `main` function.

destination is accessed, and the value is meaningless anyway. Initially only `main` is in the SPM. For simplicity, we assume none of `F0`, `F1` and `F2` calls any other functions.

When a basic block has multiple predecessors, all incoming states are joined to obtain the input SPM state to the basic block. The meet operator is defined as

$$\bigcup^{ah}(ss_1, ss_2) = \begin{cases} r_i \leftarrow ss_1(r_i) & \text{if } ss_1(r_i) = ss_2(r_i) \\ r_i \leftarrow null & \text{otherwise,} \end{cases}$$

where ss_1 and ss_2 are the two SPM states to join, r_i denotes the state of the i-th region, and $ss_1(r_i)$ and $ss_2(r_i)$ represent the i-th region states in the two incoming SPM states respectively. This operator ensures that in the resulting SPM state, the only functions left are those have been loaded and never evicted in all possible paths leading to the program point. For example, $\bigcup^{ah}(\{\{main\}, \{F1\}, \{F0\}\}, \{\{main\}, \{\}, \{F0\}\})$ would be evaluated as $\{\{main\}, \{\}, \{F0\}\}$.

In the example, the second call to `F0` (after the loop) is always-hit, since `F0` is in `r2` of the input SPM state of the basic block. We do not need to insert `_get(F0)` there. Similarly, `_get(main)` after every call can be skipped.

B. First-miss Analysis

We explain first-hit analysis with an example shown in Figure 4, in the outer loop `L2` in `main`. Again, we assume `F0`, `F1` and `F2` do not have any function calls. The output state of the back edge of `L2` (initially empty) is used as the input of its header at the beginning of each iteration. Since `L2` must be executed within its parent function, `main` must have been brought into the SPM. Therefore, the input SPM state of the loop head is {`main`}, {}, {}. When entering the inner loop `L1`, the output of its back edge is ignored, as the eviction of any function will anyway be reflected in the output of its exit edge (from `call F2` to `call F0`).

The meet operator in the first-miss analysis is defined as

$$\bigcup^{fm}(ss_1, ss_2) = \begin{cases} r_i \leftarrow ss_1(r_i) & \text{if } ss_1(r_i) = ss_2(r_i) \\ r_i \leftarrow ss_1(r_i) & \text{if } ss_2(r_i) = NULL \\ r_i \leftarrow ss_2(r_i) & \text{if } ss_1(r_i) = NULL \\ r_i \leftarrow null & \text{otherwise.} \end{cases}$$

This ensures that only the functions that would never be evicted after their initial loadings are left. For example, $\bigcup^{fm}(\{\{main\}, \{F1\}, \{F0\}\}, \{\{main\}, \{F1\}, \{\}\})$ would be evaluated as $\{\{main\}, \{F1\}, \{F0\}\}$.

The call to `F1` is first-miss in `L2`, since `F1` is in `r1` of the input SPM state of the call statement. Notice that our first-miss analysis is done loop by loop, so while the call to `F2` is not classified as first-miss in `L2`, it will be, in the first-miss analysis for `L1`. On the other hand, the first-miss analysis in static cache analysis [11] is done for all loops at once. As a result, it is not able to identify the call to `F2` in the above example as first-miss, as the analysis would find out that `F2` will be evicted in the outer loop `L2` by `F0`.

IV. Evaluation

A. Experimental setup

We apply our analyses to CMSM [7], the state-of-the-art function-level code management technique, to see how much of the overhead we can reduce. We implemented both analyses as transformation passes in an LLVM compiler [12] and compiled benchmarks from Mibench [9] with the passes enabled. Then, we collected performance statistics gem5 simulator [13]. We modeled an SPM in gem5 and also implemented DMA operations. The cost of a DMA transfer consists of setup time and transfer time. The setup time is set to 91 nanoseconds (about 291 CPU cycles), and the data transfer time is set to 0.075 nanoseconds per byte (0.24 CPU cycles) for each byte of data. These specs are borrowed from IBM Cell BE [14].

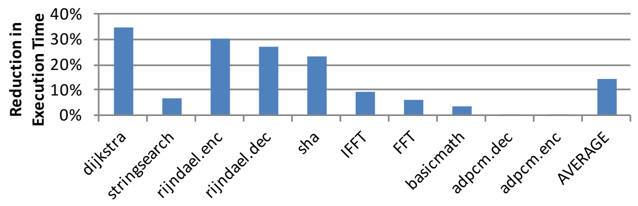

Fig. 5: By reducing management overhead, the execution times are reduced by over 14% on average.

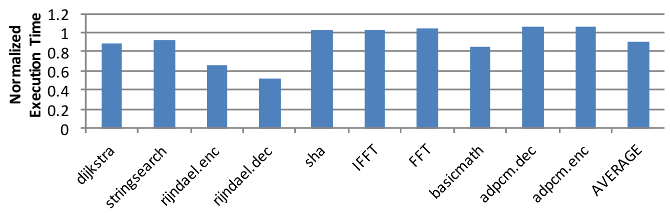

Fig. 6: Execution times normalized to those with hardware caching. Using our approach leads to better or at least comparable performance to hardware caching.

B. Code Management Overhead Reduction

Figure 5 shows the reduction in execution time after using our approach. The number of regions is set to be the half of the number of functions in each benchmark. This choice demonstrates the average-case performance between two extremes: i) the SPM space is so restrictive that all functions have to be mapped to one region, and ii) the SPM space is so large that each function can be placed in a separate region. On average, the execution time is reduced by more than 14%.

Benchmarks that receive insignificant overall performance improvement, such as `adpcm.dec` and `adpcm.enc`, have only a few function calls, so the overhead of code management was already negligible before our approach. For `stringsearch`, while it has many function calls, it has only three functions, with the `main` function calling the other two in a loop. Since there are only two regions, two of the three functions have to be mapped to one region, causing them to evict each other at every iteration. The management overhead are necessary and cannot be reduced.

C. Comparison with Hardware Caching

We compare our approach with caching in a cache-based architecture. The cache-based system has a 2-way L1 instruction cache with 64-byte cache lines on gem5 simulator. The sizes of the SPM are the same as the experiments in section IV-B. Cache size for each benchmark is set to the smallest power of two that is no less than the SPM size. Cache miss latency is the same as the DMA setup time. This configuration is conservative since it leads to significantly larger cache sizes than SPM sizes in several benchmarks, `sha`, `IFFT`, `FFT`, `adpcm.dec` and `adpcm.enc`.

Figure 6 shows the normalized execution time of benchmarks with our approach compared to hardware caching. The overhead of code management in a cache-based architecture is measured as the number of cache misses times the cache miss penalty, while the overhead in the SMM architecture is measured as the sum of the time spent executing instructions of code management function calls and DMA cost.

In several benchmarks, using an SPM-based architecture with our approach can significantly reduce the execution time. However, caching is better in `adpcm.dec` and `adpcm.enc`, in which most of the execution time is spent on small loops that are small enough to fit in the instruction cache. However, even in these cases, the execution times are comparable, and the differences are not more than 6%.

V. Conclusion

In the context of managing code blocks in SPMs, we propose two analyses that find the locations where the outcomes of checking can be safely guaranteed. Based on the analysis results, we can safely remove or hoist the management code to reduce the overhead associated with the management. With various benchmarks and various memory configurations, our experimental results show that our techniques can reduce the execution time by 14% on average. Using our approach on an SPM-based architecture, we observe that the execution times of benchmarks are significantly less or at least comparable to those on a cache-based architecture.

References

[1] B. Redd, S. Kellis, N. Gaskin, and R. Brown, "The Impact of Process Scaling on Scratchpad Memory Energy Savings," *Journal of Low Power Electronics and Applications*, vol. 4, no. 3, p. 231, 2014. [Online]. Available: http://www.mdpi.com/2079-9268/4/3/231

[2] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., 1999.

[3] 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*, 2010.

[4] C. Jang, J. Lee, B. Egger, and S. Ryu, "Automatic Code Overlay Generation and Partially Redundant Code Fetch Elimination," *ACM Trans. Archit. Code Optim.*, vol. 9, 2012.

[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*, 2013.

[6] Y. Kim, D. Broman, J. Cai, and A. Shrivastava, "WCET-aware Dynamic Code Management on Scratchpads for Software-Managed Multicores," in *Proc. of RTAS*, 2014.

[7] J. Lu, K. Bai, and A. Shrivastava, "Efficient Code Assignment Techniques for Local Memory on Software Managed Multicores," *ACM Trans. Embed. Comput. Syst.*, vol. 14, 2015.

[8] Y. Kim, J. Cai, Y. Kim, K. Lee, and A. Shrivastava, "Splitting Functions in Code Management on Scratchpad Memories," in *Proc. of ICCAD*, 2016.

[9] 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 IWCC*, 2001.

[10] C. Ferdinand and R. Wilhelm, "Efficient and Precise Cache Behavior Prediction for Real-TimeSystems," *Real-Time Syst.*, vol. 17, 1999.

[11] C. Cullmann, "Cache persistence analysis: Theory and practice," *ACM Trans. Embed. Comput. Syst.*, vol. 12, 2013.

[12] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of CGO*, 2004.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, 2011.

[14] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, 2006.