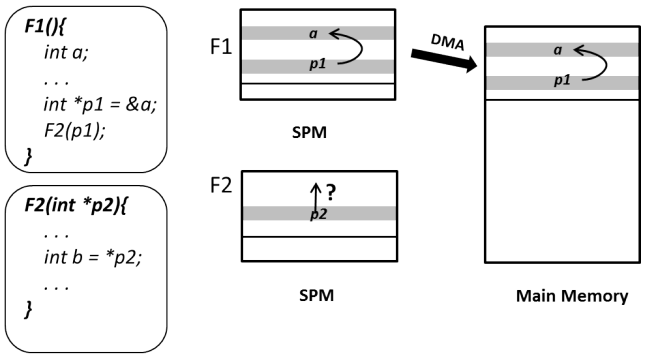
Efficient Pointer Management of Stack Data for Software Managed Multicores

Jian Cai, Aviral Shrivastava
Compiler Microarchitecture Laboratory
Arizona State University
Tempe, Arizona 85287 USA
{jian.cai, aviral.shrivastava}@asu.edu

***Abstract*—Scratchpad-memory (SPM) based memory hierarchy is a promising alternative to cache-based memory hierarchies, due to the difficulty in scaling caches to processors with high core count. However, explicit data management in software is required on SPM-based memory hierarchies. This paper focuses on optimizing the stack data management on SPM-based multicore processors, as memory accesses to call stack present in most applications. While previous works have developed techniques to enable correct stack pointer management, they have not optimized it. As a result, existing techniques still incur high overhead. This paper proposes an automated compiler-based scheme for efficient pointer management. Our experiments on MiBench benchmarks demonstrate that our scheme almost completely eliminates pointer management overhead. As a result, as compared to the state-of-the-art approach, our approach reduces the average execution time of the benchmarks by 52%. Furthermore, with our approach, the performance of stack management on SPM becomes better than hardware caches on average even with conservative estimates.**



(a) **F2** access the variable **a**, a stack variable defined in function **F1**, through pointer **p2**

(b) When **F2** is being executed, if the stack frame of **F1** has been evicted to the main memory, then pointer **p2** points to an invalid address.

Fig. 1: Pointer management problem.

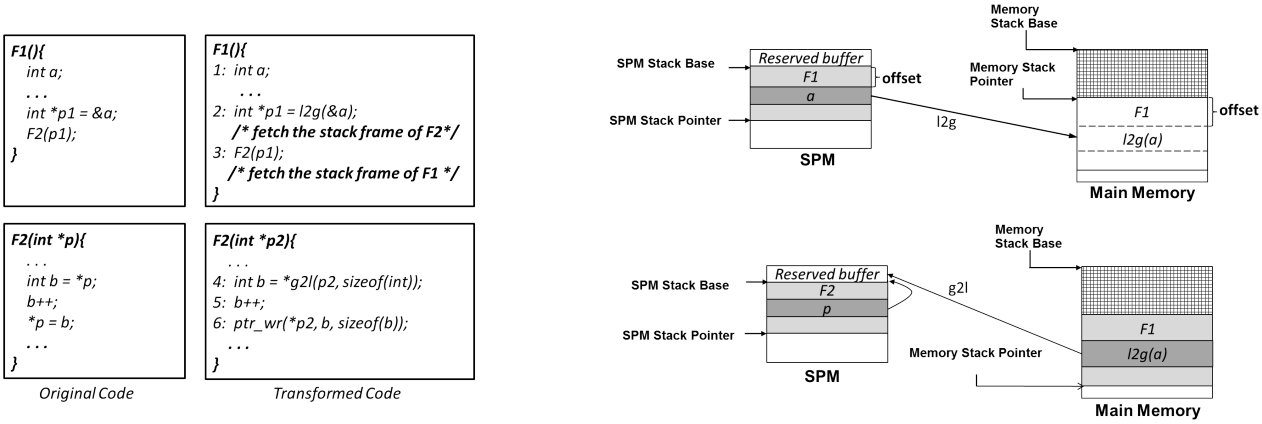
I. Introduction

Low-power, yet high core-count embedded processors cannot afford the overhead of coherent caches [1], [2], [3], [4]. The scratchpad memory (SPM) based system is a promising alternative, as it provides a fast, low-power, and scalable memory hierarchy—the SPM has 34% less area and consumes 40% less power than a cache of the same capacity [5]. Using SPMs instead of caches not only improves power, but also greatly simplifies the hardware design (and verification). SPMs shift the task of data management from hardware to the software, and therefore, multicore architectures with SPM-based memory hierarchy are termed Software Managed Multicore (SMM) architectures.

In SMM architectures, a core has to fetch data it needs to its local SPM before accessing it. Therefore we need techniques to manage data transfers between the SPM and the main memory. Among all the different types of data (heap, stack or global) to manage, optimized data management for stack data is especially important for performance. [21] shows (via profiling) that stack accesses account for around 64% of overall data accesses in Mibench, a benchmark suite of typical embedded applications [6].

State-of-the-art techniques to manage stack data on SMM architectures move stack data between SPM and main memory at the function call level. Therefore, these techniques need to solve two inter-related problems[7]. **i) Stack frame management:** stack frame of the function that is going to be executed must be brought into the SPM before it executes, and the stack frames of the functions that are not immediately needed may be evicted to the main memory. **ii) Pointer management:** if a stack frame of a function was evicted to the main memory, and the currently executing function accesses a local variable of the evicted stack frame (typically through a pointer), then the access is invalid, as shown in figure 1. This is because the pointer still contains the address of the local variable in the SPM before it is evicted. It is therefore vital to correct the address of the pointer, as otherwise the result of the execution will be incorrect.

A previous work, [7] solves the problem of pointer management, by instrumenting the code to translate the pointer address at each definition and use—A definition refers to the write of a new value to the pointer, while a use refers to the read of the value defined by the reaching definition or the last write. While this enables correct execution, it incurs high performance penalty. In this paper, we propose an efficient compiler technique for managing pointers to stack data on SMM architectures. The two key ideas of our approach are: i) instead of translating the pointer address at each use of the pointer inside a function, we translate it only once when it is passed as the argument of the function. As a result, our technique is able to remove a significant portion of the overall



(a) Code transformation.

(b) Illustration of pointer management functions.

Fig. 2: The way pointer management functions work.

translations. ii) if the stack frame of the function whose local variables are being accessed through pointers is guaranteed to be present in the SPM, then when any of the pointers is accessed, no translation is needed.

Experiments on benchmarks from the MiBench suite [6] show that our approach almost completely eliminates the pointer management overhead, and results in 52% reduction of the average execution time, as compared to the state-of-the-art pointer management technique [7] on top of the state-of-the-art stack frame management technique [8] on SMM architectures. We also compare the performance of our stack pointer management on SPM with that on a cache-based architecture. Even with conservative estimates, stack data management on SPM outperforms stack data management on a cache-based architecture by 12% on average.

II. Related Work

Stack management techniques in general can be divided into static approaches and dynamic approaches. Static approaches [9], [10], [11] map the most frequently used data to SPM and keep the allocation fixed throughout execution, while dynamic approaches allow the changes to the locations of stack data at run-time. Static approaches do not perform well since they do not take dynamic program behaviors into consideration. As a result, most recent works focus on dynamic SPM management techniques.

Many dynamic techniques [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [7], [8] have been developed to manage stack data on SPM. [12], [13], [14], [15] introduce new hardware functionality to manage the SPM, while the interest of this paper lies in providing software solution to simplified hardware. Among software solutions, [17] and [18] target on arrays specifically, [16] mainly focuses on managing stack data for recursive functions on SPM, while we manage all the data in stack. [19] and [20] both rely on profile information, therefore the input has to be representative for either of them to deliver high-quality output, which is generally difficult. In addition, both approaches have limited support for pointer management. [19] relies on pointer analysis to identify and translate the address for any pointer that refers to the variable that is moved from the main memory to the SPM. If the point analysis fails to identify any such accesses through pointers and thus the requested memory addresses are not translated, the execution may fail, since these accesses end up accessing incorrect locations. [20] simply does not support using stack pointers as call arguments. In this paper we are interested in generic approaches of stack management that do not rely on profiling or pointer analysis, and are able to manage pointers correctly. To our best knowledge, only the Circular Stack Management (CSM) [21], [7], [8] approaches provide such solution. In this paper we will compare our work with [8], which is the latest CSM work with the best performance.

III. Background

All the CSM papers use the pointer management proposed in [7]. The pointer management maintains two stack pointers: one in SPM and one in main memory (assume the stack grows from the higher address (stack base) to the lower address (stack top)). It proposes three pointer management functions: *l2g*, *g2l*, and *ptr_wr*.

Figure 2 explains the functionality of these pointer management functions. Figure 2a shows the original code and the transformed code with pointer management. Figure 2b illustrates *l2g* and *g2l* calls at line 2 and line 4 in the transformed code respectively, assuming the SPM is not large enough to hold both the stack frames of function *F1* and *F2*. When *F1* calls *F2*, the stack frame of *F1* must be evicted from SPM to the main memory to make space. The SPM address of stack variable *a* defined in *F1* which is passed to *F2* will become an invalid reference by the time it is accessed, since the entire frame of *F1* (thus the the stack variable *a*) will have been moved to the main memory. In this case, *l2g* function should be called on *a* in *F1* to calculate the address of the actual location of *a* in the main memory (line 2 in the transformed code in Figure 2a). Notice when *l2g(a)* is called, the stack frame of *F1* has not been evicted to the main memory yet. Therefore, the value of *l2g(a)* indicates the memory location *a* will be moved to. At that time, *l2g(a)* is smaller than the value of the memory stack pointer, and their distance is the same as the difference between the value of the SPM stack base and the SPM address of *a*, as *offset* in Figure 2b indicates (the upper figure of Figure 2b). After *F1* is evicted to the main memory, the value of memory stack pointer is decreased by the size of the stack frame of *F1*.

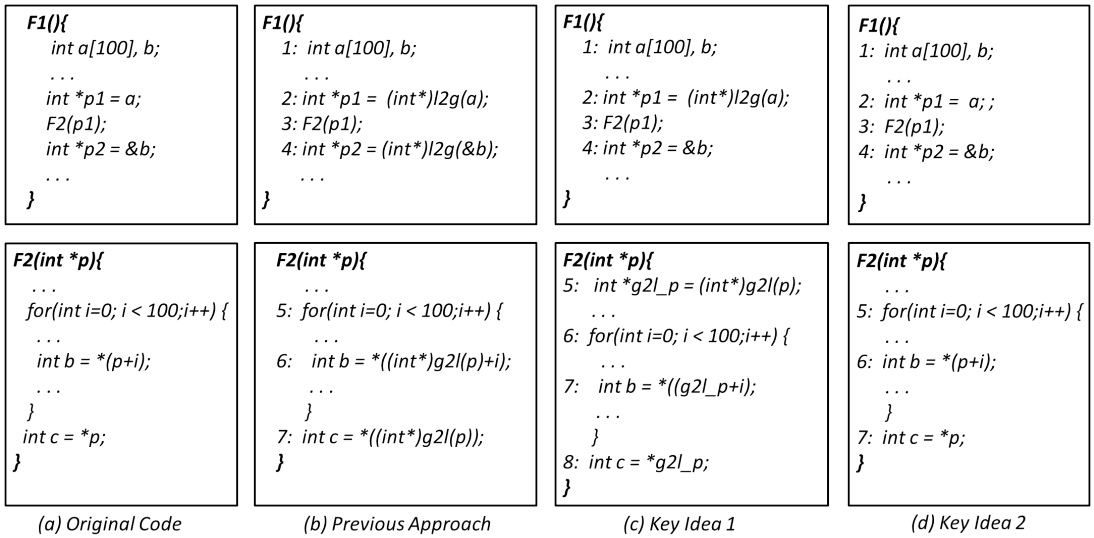


Fig. 3: The key ideas of our approach.

Consequently, *l2g(a)* refers to the actual location of *a*, which becomes larger than the value of memory stack pointer (the lower figure of Figure 2b).

The result of *l2g(a)* is passed as an argument to *F2* as its parameter *p*. *F2* then calls *g2l* function before dereferencing it (line 4 in the transformed code). Since cores cannot access main memory directly, *g2l(p)* allocates a local buffer in SPM, followed by a DMA instruction to read the value from the main memory location specified by *p*—or equally l2g(&a)—and then return the address of the local buffer. The correct value can then be read from the local buffer. Finally, *F2* modifies the value pointed by *p*, and calls *ptr_wr* to write back the modification from SPM to main memory (line 6 in the transformed code).

Pointer management functions will not alter program semantics. Consider the example in Figure 2 again, but this time we assume that the SPM is large enough to hold the stack frames of both *F1* and *F2*. Therefore, when *F2* is called, the stack frame of *F1* is still in the SPM, and we can safely remove the stack frame management around line 3 at the transformed code in Figure 2a. In such a case, the call to *g2l* in *F2* (line 4) will do nothing but reverting the address translation done by the *l2g* function in *F1* (line 2), and reading from the SPM address of *a*, which is the input to the *l2g* function. Similarly, *ptr_wr* will revert the translation and write to the SPM address of *a* directly. Whether a stack variable has been evicted from the SPM to the main memory can be told as below. If an address that is passed to *g2l* or *ptr_wr* is smaller than or equal to the value of memory stack pointer, then the stack variable the address refers (thus the enclosing stack frame) is still in the SPM. In this case, *g2l* or *ptr_wr* just need to revert the address translation done by *l2g* and read from or write to the SPM address. Otherwise, if the address is greater than the value of the memory stack pointer, then the stack variable has been evicted, and *g2l* and *ptr_wr* will go ahead and perform required DMA operations. Therefore, even though pointer management from [7] unnecessarily inserts extra pointer management functions, it can still ensure the correctness of the execution of programs.

Pointer management from [7] solves the problem for correctness, but not for performance. On the other hand, our approach removes unnecessary calls to pointer management functions and improves performance of applications noticeably.

IV. Key Ideas of Our Approach

While the state-of-the-art pointer management from [7] solves the pointer corruption issue correctly, it calls the *l2g* function at every definition of stack data pointers, and the *g2l* function on every use of the pointers, and results in unnecessary calls to pointer management functions, which eventually slows down the execution of programs.

Figure 3a and 3b show the original code and the code with the pointer management from [7]. The calls to *l2g* in line 4 in Figure 3b is not necessary, since the pointer *p2* is only used in the same function the variable *b* is defined. Meanwhile, although the calls to *g2l* in line 6 is necessary, it can be promoted to be outside of the loop to avoid repeated computations. Notice the calls to *g2l* in line 6 can not be eliminated or reduced by standard compiler optimizations such as common subexpression elimination or loop invariant code motion. This is because *g2l* function needs to access some global states (implemented as global/static variables), such as the current values of the stack pointers, which could be changed by function calls and stack frame management between any two consecutive *g2l* calls. These interactions with global states prevent standard compiler optimizations from removing or relocating *g2l* calls, since the compiler cannot guarantee the changes will not cause any unexpected side effect to the semantic of programs.

This work aims to reduce these overheads based on two key ideas: i) we only manage pointers when they are used as call arguments instead of each of the uses, so that we only need to translate once at the caller and the called function respectively. ii) if the stack frame of a function is definitely in the SPM, then any accesses via pointers to the local variables in the

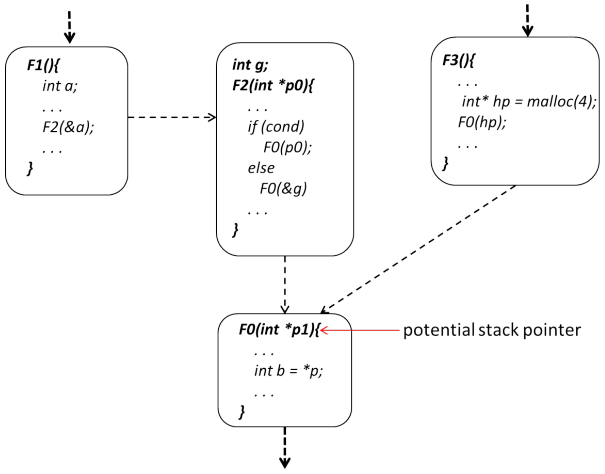
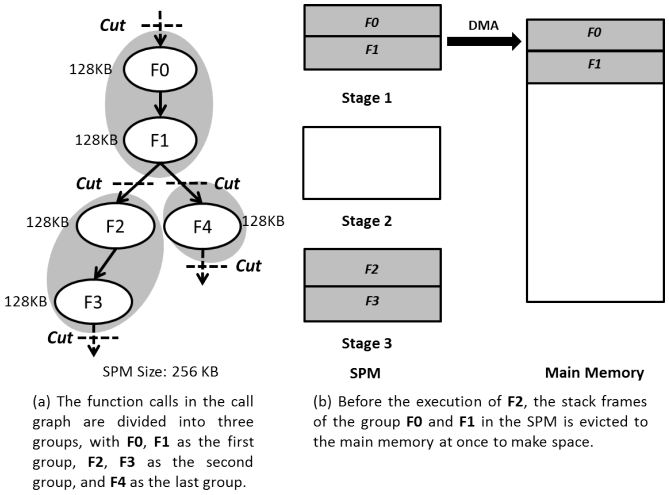


Fig. 4: Identification of the potential pointer to stack.



(a) The function calls in the call graph are divided into three groups, with **F0**, **F1** as the first group, **F2**, **F3** as the second group, and **F4** as the last group.

(b) Before the execution of **F2**, the stack frames of the group **F0** and **F1** in the SPM is evicted to the main memory at once to make space.

Fig. 5: The analysis to find out at which stack frames will exist in SPM at the same time.

stack frame do not need management. Figure 3c demonstrates the first idea. Instead of calling *l2g* on every definition of pointers (line 2 and 4 in Figure 3b), we only call *l2g* once in *F1* when it calls *F2* and passes a pointer-type argument (line 2 in Figure 3c). Also, we only call *g2l* function once at the beginning of *F2* and reuse the result (line 5, 7 and 8 in Figure 3c), instead of calling it for every use (line 6 and 7 in Figure 3b). The *g2l* function is called outside the for loop to reduce overhead. Figure 3d shows the further optimized code with the second idea. If we know for sure the stack frames of *F1* is in the SPM when *F2* is called, then no stack frame management is needed, neither the pointer management—the references to the array *a* of *F1* through pointer *p* in *F2* will be guaranteed to access the correct locations, as the stack frame of *F1* is not moved. The code in Figure 3d becomes exactly the same as the original code. In other words, it eliminates the overhead of stack management—both stack frame management and pointer management.

V. Details of Our Approach

A. Steps of Our Approach

To achieve the efficient pointer management, our approach takes three steps. First, we need to decide if any pointer-type arguments of a function can be potential references to stack variables. Second, we will run the analysis to divide function calls in the call graph into groups, so that all the stack frames of each group can fit into the SPM at once. In the last step, we insert pointer management functions properly based on the previous analyses.

1) Identifying Stack Data Pointers at Function Calls: First of all, we perform an inter-procedural analysis to find out if any pointer-type arguments at function calls are potentially referring to stack variables. A function may be called at multiple locations during the execution of a program, therefore the same parameter may refer to multiple arguments that can be stack, heap or global variables, depending on the control flow at run-time. Consider the call graph in Figure 4. When *F0* is called, the type of the argument that is referred by the parameter *p1* may have three different types. If the control flow comes from *F2*, and *cond* is evaluated to be *true* in *F2*, then the argument passed to *F0* is a pointer to stack variable *a*, which is defined in *F1* and passed to *F2* when it is called; Otherwise, if the *cond* is evaluated to be *false*, then the argument is a pointer to the global variable *g*. If the control flow comes from *F3* to *F0*, then *p1* in *F0* refers to the heap data referred by *hp* in *F3*. Since the actual control flow is not known at compile-time, we have to conservatively assume *p1* refers to a stack variable.

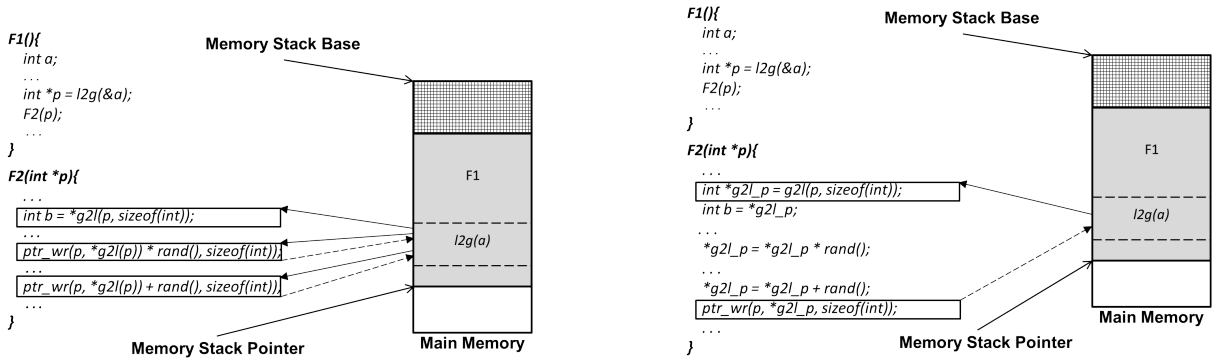
To accomplish such analysis, we first go though all the functions and identify all the pointer-type formal parameters. Once we find such a parameter, we check all the call sites of the function, and find out all the possible arguments. Any of these arguments that refers to stack data needs to be managed.

Several cases pose challenges for this analysis. When a pointer to stack data is passed as an argument to a recursive function, then we need to call *l2g* on the pointer, and call *g2l* on the result of *l2g* function that is passed to the called function in the called function. This is because we do not know how many times the recursion will happen at compile time, so we need to conservatively assume the stack frame of the caller is evicted when the called function is executed. When the type of the variable a pointer refers to cannot be identified, we conservatively assume it is a stack pointer and manage it.

2) Identifying Coexistent Stack Frames: This step decides which stack frames can exist in SPM at the same time. We use the same analysis from Smart Stack Data Management heuristic (SSDM)[8]. Here we just explain the high-level idea. Details of the algorithm can be found in [8].

The general idea of SSDM is that instead of managing stack frames one at a time at every function call, we can manage multiple stack frames of consecutive function calls along any path of a call graph at the same time. Instead of evicting the stack frame of the caller function from the SPM to main memory to make space for the called function whenever a function call happens, we can keep allocating SPM space for stack frames of function calls, and perform the stack frame management all at once only when there is not enough SPM space.

In the given call graph in Figure 5, the size of the stack



(a) Previous pointer management calls *g2l* and *ptr_wr* functions on every read and write to stack variables.

(b) Our pointer management calls the *g2l* function before the first read and *ptr_wr* after the last write, and reuse the local buffer for other accesses to the same memory location.

Fig. 6: Compared to previous pointer management, our approach reuses the local buffer created by g2l function and saves management overhead.

frame of each function is 128 KB, and the size of the available SPM space is 256 KB. Therefore, the available SPM space can hold two stack frames at once. Assume the SPM is empty at first, then in the given call graph, we know the stack frame management are only necessary when *F1* calls *F2*, or when *F1* calls *F4*. Any other function calls do not need such management. For example, right before *F0* calls *F1*, the SPM only keeps the stack frame of *F0*, which takes up 128 KB space. The spare space in the SPM is large enough to hold the stack frame of *F1*. Therefore, no stack frame management is needed when this call happens. As a result, we can divide the call graph into three groups, $\{F0,F1\}$, $\{F2,F3\}$, and $\{F4\}$. The three groups are separated from each other by the dash lines in the figure, or what are termed *cuts*. Each cut between two adjacent groups indicates the need for inserting stack frame management functions. When any function call crosses a cut, the stack frames of the group the caller is in (currently in the SPM) are moved to the SPM, and the stack frames of the group the called function is in are brought from the main memory to the SPM. For instance, before *F1* calls *F2* in Figure 5, the SPM holds the stack frames of *F0* and *F1*. When *F1* calls *F2*, the stack frames of *F0* and *F1* are evicted to the main memory, and the stack frames of *F2* and *F3* are brought to the SPM.

Once we divide the call graph into different groups, we know the stack frame management is only needed for function calls that crosses any two different groups. This is true for pointer management as well, since pointer management is necessary only if stack frames are moved due to stack frame management.

Our analysis initially places a *cut* on each edge of the call graph, which specifies the need of stack management (both of stack frame management and pointer management) for the call represented by the edge, and then greedily remove the cut which will result in the greatest reduction of stack management overhead—for instance, inserting management functions within a loop should be avoided as far as possible—while not violating the constraint that the sizes of stack frames between any two cuts should not be greater than the available SPM space, until it can not find any such cut.

3) Inserting Pointer Management Functions: Once we have the necessary information ready, we can decide where to insert pointer management functions. We first go through the call graph and check (1) if any function call passes any pointers that may refer to stack variables (from the analysis done in V-A1), and (2) if the stack frames that enclose these stack variables are in the SPM when the pointers are accessed in the called function, or in other words, if the called function that accesses the pointers belongs to the same group of the function that defines the stack variables (from the analysis done in V-A2). Upon the confirmation of both conditions, we need to call *l2g* function on these pointers; otherwise, if any such pointer is not referring to a stack variable, or the stack variable the pointer refers to is in the SPM, then no pointer management is required for this pointer.

If we call *l2g* on a pointer-type argument on any call site of the called function, we need to call *g2l* and *ptr_wr* function in the called function for reads and writes to the pointer respectively, since we do not know which call site will the control flow comes at compile-time. While this may cause unnecessary calls to *g2l* and *ptr_wr* functions, we have explained that extra pointer management functions will not affect the correctness of programs. On the other hand, if we do not conservatively insert these pointer management functions in the called function, the correctness of execution will not be guaranteed, since there is a chance that the control flow may come from the caller function with *l2g* function calls at run-time.

As an optimization, we reuse the local buffer created by *g2l* function, in contrast to creating new buffer and destroying it every time by the previous approach. Figure 6 shows an example. The previous pointer management [7] will call *g2l* and *ptr_wr* function on each read and write to stack data pointer *p* in *F2* respectively, even if these memory accesses are to the same memory location. On the other hand, our approach only inserts *g2l* before the first read and *ptr_wr* after the last write of *p*, and redirect the other memory request to the local buffer *g2l_p* created by the *g2l* function call. With such a

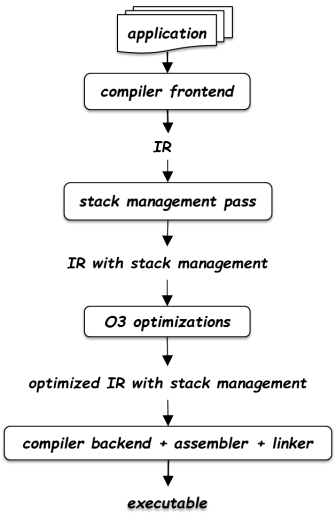


Fig. 7: The compilation process of benchmarks used in experiments.

policy, we can avoid redundant memory allocation and DMA requests.

When *g2l* or *ptr_wr* function is called, the compiler needs to pass the size of the stack variable in case of triggering DMA transfers. When there are multiple possible sizes, we need to use the maximal possibility. For example, a function may be called at two different cites with two array-type arguments of different sizes. In this case, since we do not know from which call site control will flow at run-time, we have to use the size of the larger array. While this approach may transfer more than necessary data if at run-time the control flow comes from the function with the smaller array, using the maximum size will not affect the correctness of the program being executed.

VI. Experiments

A. Improvement Over The State of The Art

We compare our pointer management with the state-of-the-art pointer management proposed by [7], on top of the latest stack frame management technique Smart Stack Data Management (SSDM) [8]. We implement the two approaches of stack management as passes in LLVM compiler infrastructure [22]. We compile benchmark applications from Mibench benchmark suite [6] with each of the two LLVM passes, then run and collect performance statistics of the execution of generated binaries in the Gem5 CPU simulator [23].

The compilation process of benchmarks is shown in Figure 7. All the compilations in our experiments are done with O3 optimization on. The LLVM passes are implemented at the Intermediate Representation (IR), a transitional stage between the translation from source code to machine language. In other words, our passes are independent of the Instruction Set Architecture (ISA) used, and should work with different compiler back ends for code generation.

We build the SPM aside the main memory, and implement a DMA instruction for data transfers between them in the Gem5 simulator. The DMA cost in our experiments consists of the start-up cost and the transfer time. The start up cost includes all the time spent setting up the DMA transfer, and the transfer time is the time spent on transferring the requested data, which can be calculated by dividing the size of the data by the bandwidth. The CPU frequency is set to 3.2 GHz in the Gem5 simulator. The start up cost is set to 91 nanoseconds (about 291 CPU cycles), and the data transfer rate is set to 0.075 nanoseconds (0.24 CPU cycles) per byte of data [24] (4 bytes/cycle). These numbers are consistent with the parameters used in [8].

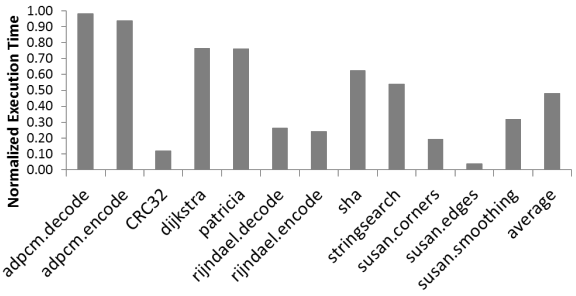


Fig. 8: Normalized execution time of benchmarks with stack management using our pointer management compared to using the previous pointer management. Our pointer management reduces the execution time by 52% on average.

Table I shows the number of pointer management function calls introduced by the state-of-the-art pointer management and our approach (the first three columns under *Previous Pointer Management* and *Our Pointer Management* categories respectively). The numbers show that our approach almost completely eliminates calls to the pointer management functions, i.e. *l2g*, *g2l* and *ptr_wr*. For example, for *rijndael.encode*, the numbers of calls of *l2g*, *g2l*, and *ptr_wr* are reduced from 155940, 1442301, 28 to 1, 1, and 0 respectively. These results are for experiments on SMM architecture with the SPM size equal to the average of the minimum and maximum stack size for each application. We will explain our choice of the SPM size later.

The reduction of pointer management consequently reduces the execution time of applications. Figure 8 shows the normalized execution time of benchmarks using our approach over the previous approach. Our approach achieves on average 52% reduction of execution time. Notice that even for the benchmarks in which we do not achieve significant performance improvement, for instance, *adpcm.decode*, and *adpcm.encode* our pointer management still reduces the pointer management overhead. We get less improvement because the time spent on pointer management is insignificant compared to the execution time in these applications.

B. Comparable Performance Compared to Caches

On top of the comparison with the state-of-the-art stack management techniques for SMM architectures, we also compare the performance of our technique with hardware caching. We perform a conservative comparison with cache-based architectures. The cache-based system is configured to have a 4-way L1 data cache which only caches the stack data. All the other memory accesses are considered as cache hits. The

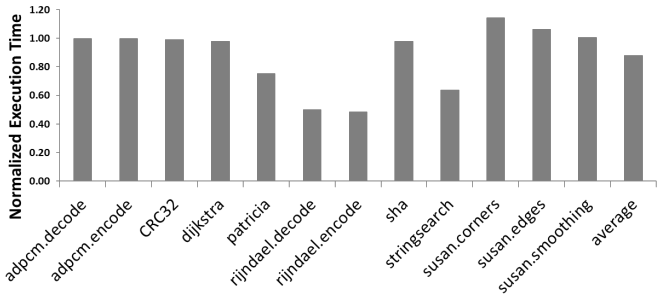


Fig. 9: Normalized execution time of the stack management with our pointer management on a SMM architecture to hardware caching. Our approach achieves 12% reduction of execution time on average, even with conservative estimates.

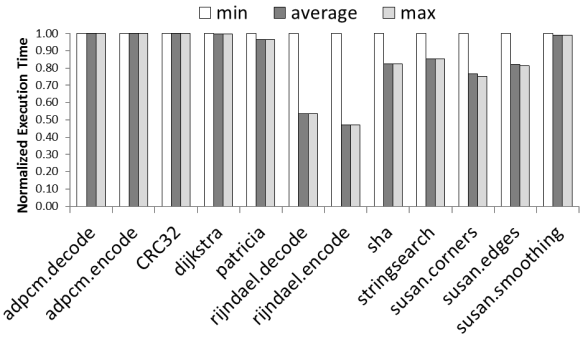


Fig. 10: Execution time of benchmarks with our approach using three SPM sizes, all normalized to the execution time that uses the minimum SPM size.

size of the cache is configured to be the smallest power of two greater than the SPM size. Also, we set the cache miss penalty to be the same as the DMA start-up cost. The overhead in a cache-based architecture is equal to the number of cache misses times the cache miss penalty. Meanwhile, the overhead of stack management in a SPM-based architecture includes both the time for executing the extra management instructions, and the time for DMA operations to move data. For each application, the SPM size is the average of the minimum and maximum stack size of the application (again the reason will be explained later).

Table I shows the stack management overhead caused by our approach on an SMM architecture (the fourth to sixth columns under *Our Pointer Management* category) versus that caused by hardware caching. The DMA transfers for benchmarks without pointer management are triggered by stack frame management. When the benchmark $rijndael.encode$ is executed on the SMM architecture, our approach requires 87 management instructions and 2 DMA calls which transfers 2336 bytes of data. When this benchmark is executed on a cache-based system with the stack data being managed on a cache slightly larger than the SPM, it incurs 244983 misses. Therefore, even with the extra instructions, SPM management is still more efficient. Figure 9 plots the execution time of a benchmark on the SMM architecture, normalized to the execution time of the same application run on the cache-based system. The plot shows that our approach reduces the execution time by 12% on average.

C. Choice of SPM Stack Size

Figure 10 shows the execution time of the benchmarks with three SPM sizes: the minimum required stack size (size of the largest stack frame in the application), the maximum possible stack size (sum of the sizes of all the stack frames), and their average. All of them are normalized to the execution time when using the minimum size. As the figure shows, while using the minimum size may cause longer execution, using the average size or using the maximum size are not much different.

This is because in all the benchmarks we used, the size of the largest stack frame is much larger than the others, so even if we only use the average SPM size (greater than the size of the largest stack frame), it is large enough to hold multiple small stack frames, which is able to eliminate pointer management for calls between these functions with small stack frames. However, if we only allocate the minimum required size (exactly equal to the size of the largest sack frame), and the function with largest stack frame happens to be called in the loop, there will be no other choices but to evict stack frames within the loop every time the function is called, which

TABLE I: Overhead of pointer management

	Previous Pointer Management			Our Pointer Management						Hardware Caching
benchmark	#l2g	#g2l	#ptr_wr	#l2g	#g2l	#ptr_wr	#DMA	Overall DMA Size	#Management Instructions	#L1D Misses
adpcm.decode	3428	2740	1370	0	0	0	0	0	0	30
adpcm.encode	3427	2740	1370	0	0	0	0	0	0	63
CRC	1368874	2737731	1368866	2	2	2	2	160	156	5361
dijkstra	90548	45309	44925	0	0	0	30758	1477664	784309	51964
patricia	104017	52763	3801	0	0	0	4902	436896	124551	274607
rijndael.decode	136447	1422796	11	1	1	0	2	2336	87	244983
rijndael.encode	155940	1442301	28	1	1	0	2	2336	87	244983
sha	5041	19827	12270	0	0	0	2	608	50	1578
stringsearch	606	798	57	0	0	0	0	0	0	756
susan.corners	50	242719	103	5	4	3	44	1703104	1331	36
susan.edges	50	550091	2716	5	4	3	44	1703104	1331	37
susan.smoothing	48	1630815	1535	7	6	4	46	2423392	1459	29

causes much higher overhead, such as $rijndael.encode$ and $rijndael.decode$. Therefore, the average size is chosen to balance the execution time and SPM space used.

VII. Conclusion

In this paper we proposed an approach of pointer management on stack data for Software Managed Multicore (SMM) architectures. Our approach divides function calls of a program into groups based on the call graph and inserts pointer management functions only if a pointer to stack data is defined and used in two different groups. The experimental results demonstrate that our approach not only significantly improves overall performance compared to the state-of-the-art pointer management in stack management, but also delivers comparable performance over using the cache for stack data management.

Acknowledgment

This work was partially supported by funding from National Science Foundation grants CCF 1055094 (CAREER), and CNS 1525855.

References

[1] G. Bournoutian and A. Orailoglu, "Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive Mobile Processors," in *Proc. of CODES+ISSS*, 2011, pp. 89–98.

[2] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *Proc. of PACT*, 2011, pp. 155–166.

[3] A. Garcia-Guirado, R. Fernandez-Pascual, A. Ros, and J. Garcia, "Energy-Efficient Cache Coherence Protocols in Chip-Multiprocessors for Server Consolidation," in *Proc. of ICPP*, 2011, pp. 51–62.

[4] Y. Xu, Y. Du, Y. Zhang, and J. Yang, "A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs," in *Proc. of ICS*, 2011, pp. 285–294.

[5] B.-S. L. M. B. R. Banakar, S. Steinke and P. Marwedel, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems," in *Proc. of CODES*, 2002, pp. 73 – 78.

[6] 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," *Proc. of Workload Characterization*, pp. 3–14, 2001.

[7] K. Bai, A. Shrivastava, and S. Kudchadker, "Stack Data Management for Limited Local Memory (LLM) Multi-core Processors," in *Proceedings of the International Conference on Application Specific Systems, Architectures and Processors (ASAP)*, 2011, pp. 231–234.

[8] J. Lu, K. Bai, and A. Shrivastava, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)," in *Proceedings of the 50th Design Automation Conference (DAC)*, 2013.

[9] O. Avissar, R. Barua, and D. Stewart, "An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems," *ACM TECS*, vol. 1, no. 1, pp. 6–26, 2002.

[10] M. Verma, S. Steinke, and P. Marwedel, "Data Partitioning for Maximal Scratchpad Usage," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '03. New York, NY, USA: ACM, 2003, pp. 77–83. [Online]. Available: http://doi.acm.org/10.1145/1119772.1119788

[11] N. Nguyen, A. Dominguez, and R. Barua, "Memory Allocation for Embedded Systems with A Compile-time-unknown Scratch-pad Size," in *Proc. of CASES*, 2005, pp. 115–125.

[12] M. Mamidipaka and N. Dutt, "On-chip Stack Based Memory Organization for Low Power Embedded Architectures," in *Proc. of DATE*, 2003, pp. 1082–1087.

[13] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, "An Integrated Hardware/Software Approach for Run-time Scratchpad Management," in *Proc. of DAC*, 2004, pp. 238–243.

[14] S. Park, H.-w. Park, and S. Ha, "A Novel Technique to Use Scratch-pad Memory for Stack Management," in *Proc. of DATE*, 2007, pp. 1478–1483.

[15] H. Cho, B. Egger, J. Lee, and H. Shin, "Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU," in *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '07. New York, NY, USA: ACM, 2007, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1254766.1254804

[16] A. Dominguez, N. Nguyen, and R. K. Barua, "Recursive Function Data Allocation to Scratch-pad Memory," in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '07. New York, NY, USA: ACM, 2007, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/1289881.1289897

[17] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic Management of Scratch-Pad Memory Space," in *Proc. of DAC*, 2001, pp. 690–695.

[18] L. Li, L. Gao, and J. Xue, "Memory Coloring: A Compiler Approach for Scratchpad Memory Management," in *Proc. of PACT*, 2005, pp. 329–338.

[19] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions," *ACM TECS*, vol. 5, no. 2, pp. 472–511, 2006.

[20] L. Gauthier and T. Ishihara, "Implementation of Stack Data Placement and Run Time Management Using a Scratch-Pad Memory for Energy Consumption Reduction of Embedded Applications," *IEICE*, vol. 94-A, no. 12, pp. 2597–2608, 2011.

[21] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee, "A Software Solution for Dynamic Stack Management on Scratch Pad Memory," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 612–617. [Online]. Available: http://dl.acm.org/citation.cfm?id=1509633.1509775

[22] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[23] 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, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[24] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, pp. 10–23, 2006.