A SOFTWARE-ONLY SOLUTION FOR STACK MANAGEMENT ON SYSTEMS WITH SCRATCH PAD MEMORY

by

Arun Kannan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2008

A SOFTWARE-ONLY SOLUTION FOR STACK MANAGEMENT ON SYSTEMS WITH SCRATCH PAD MEMORY

by

Arun Kannan

has been approved

October 2008

Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Charles Colbourn
Rida Bazzi

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

The pursuit for higher performance and higher power-efficiency in computing has led to the evolution of multi-core processor architectures. Early multi-core processors primarily used the shared memory multi-processing paradigm. However, the conventional shared memory architecture, due to its limited scalability becomes a performance bottleneck. Newer architectures like the IBM Cell with 10 cores have adopted new memory architectures to truly enable the peak computing performance available. In order to achieve higher performance, it is necessary to re-design not only the bus topology, but also the memory hierarchy. The distributed memory model used in non-uniform memory access (NUMA) architectures is becoming popular in these modern processors.

Conventional on-chip memory like caches have been replaced by a low power, low area alternative called scratch pad memory (SPM). Caches perform the data and code transfers in hardware in an automated fashion. Unlike caches, the transfers in SPM need to be explicitly managed by the compiler. In order to achieve a power-efficient operation, it is important to map the most frequently used objects onto the SPM. In this thesis, a dynamic scratch pad memory management scheme is proposed for program stack data with the objective of processor power reduction. As opposed to previous efforts, this technique does not need the SPM size at compile-time, does not mandate any hardware changes, does not need profile information and seamlessly integrates support for recursive functions. This solution manages stack frames on SPM using a software scratch pad memory manager (SPMM), integrated into the application binary by the compiler. The experiments on benchmarks from MiBench suite show average energy savings of 37% along with a performance improvement of 18%.

To

my parents,

Shruti

&

Khush

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Aviral Shrivastava for giving me the opportunity to work in the Compiler and Micro-architecture Lab on a wonderful research project. This work would not have been possible without his invaluable guidance. I would like to thank him for teaching me to think about a problem from a broader research perspective which helped me every step along the way. His vision for this research and his relentless efforts has been a great source of inspiration and helped me take this work a baby-step towards the bigger picture. His emphasis on crisp technical writing and presentation enabled me to hone my writing and presentation skills.

I would also like to thank Dr. Charles Colbourn and Dr. Rida Bazzi for the invaluable guidance, participation and support. I thank my lab colleagues Jong-eun Lee, Sai Mylavarapu, and Reiley Jeyapaul for the research discussions. I thank my lab colleague and dear friend Amit for all the technical help with this research. Khushboo, Amit, Rooju and Vivek thanks for making my Masters a fun-filled experience. I will always cherish the endless discussions and laughter filling that 'fun-aisle' of the lab. Rooju, thanks so much for being such a sport and lending yourself to be the victim of our jokes. You have been a great project partner and a wonderful friend. A special mention to Shankar a.k.a 4s, for always being there to help me, right from my first day in this country.

I would like to acknowledge the Computer Science and Engineering department for providing me teaching assistantship, and I am eternally thankful to my academic advisor, Martha Vander Berg for all her help during the course of my program.

I am very grateful to my family. Without their encouragement it would have been impossible for me to finish this work. Last but definitely not the least, I would like to

extend my special thanks to Khushboo, for the unconditional support she has given me in every aspect of my life. Her enthusiasm and smile always keeps me going.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. Introduction

Over the past two decades, computers have evolved from simple number crunching machines in the size of a boiler room to the extremely high performance, general purpose, compact devices like the iPhone [3]. During the course of this evolution, the improvements in processor architecture have been mainly performance-centric. The processor became increasingly complex with out-of-order execution, instruction level parallelism, involved branch prediction mechanisms just to name a few. The process technology, following the Moore's law has played a very important factor in enabling this complexity. Intel is already producing chips with 45nm technology (code name 'Penryn') which allows packing 410 million transistors on a single die [18]. But, it was not until the turn of the millennium, that other significant challenges in design related to leakage power, thermal effects became dominant. These factors not only increase power consumption and cooling costs, but can severely hamper performance.

Given these challenges, it has become imperative to design processors with a good power-performance ratio. Over the recent years, multi-core processor designs have become increasingly popular. These architectures have two or more processors on the same die. Each of the cores are simpler and operate on a lower frequency as compared to their uni-processor ancestors. The hardware support for parallelism available in multi-core processors can be utilized by the application to achieve higher performance and lower power. Intel Core2 Duo [17] with two cores were among the first multi-core processors to be used in consumer markets followed by the Intel Quad-Core [19] with four cores launched in the server market. These processors use the shared memory architecture meaning all the cores share a common bus to the main memory and have their own caches. This introduces cache coherency problems and reduced aggregate bandwidth available to each core, thus

limiting their full potential. Such a design, although feasible with lesser number of cores, is not a scalable solution for future processors with 100's of cores. The shift from multi-core to many-core design has already begun with newer architectures like the IBM Cell [16] and Intel Tera-Scale [20]. IBM Cell [16] processor used in the Sony Playstation3 [31] gaming console has two master PowerPC [15] processors and eight synergistic processing units (SPU). Intel's Tera-Scale computing is building a scalable multi-core architecture and currently has an $80-core$ prototype. These modern processors are using new memory architectures and bus topologies to achieve the goals of high bandwidth, low power and scalability. The IBM Cell [16] processor has a cache-less NUMA architecture; each of the eight SPUs have fast access to their own local memory (called *Local Store*).

Power consumption is an important concern in all computing systems ranging from embedded devices to large server systems. Battery operated embedded devices need to be extremely power efficient for longer operation times, whereas server farms need to reduce their cooling costs. The cache has been identified as the single most power hungry component in the processor [8]. It is also worth noting that the cache also occupies a majority of the silicon real estate. The experiments conducted by Banakar et. al [8] show that the cache in StrongARM 110 consumes about 45% of the total dynamic processor power [8]. More disturbing is the trend of rapidly increasing leakage power of the cache [23].

Due to the aforementioned problems, the use of alternative low-latency, low-power, on-chip memory known as Scratch Pad Memory (SPM) has become very popular. Banakar et al. [8] observed that SPMs consume about 40% less power, and occupy 34% less area as compared to a cache of similar capacity. The IBM Cell processor uses SPM for the *Local Store* memory for its low-power synergistic processor units. Network Processors like the

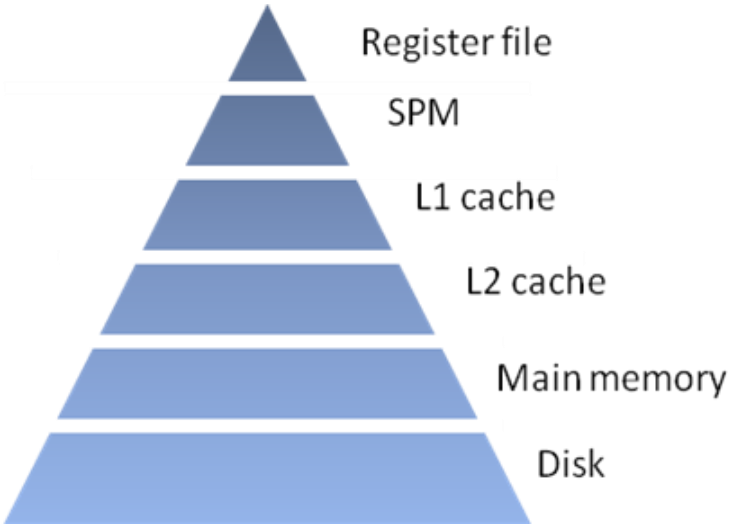


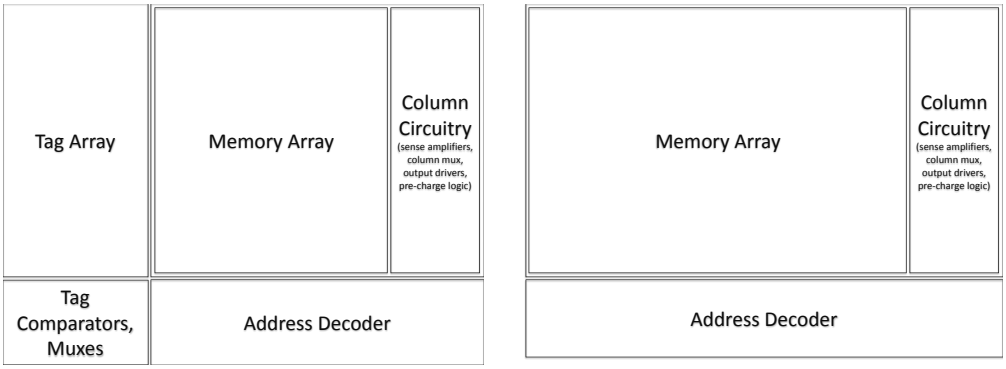
Fig. 1. Energy per access pyramid

Intel IXP1200 [1] also rely on SPMs in their micro-engines for power-efficient processing. The pyramid shown in Figure 1 shows the energy per access trend for all types of memory. It can be seen that the energy per access increases as we move further away from the processor. It is important to note here that although, SPM is shown at a level lower than the L1 cache in terms of energy, it is still at the same level as the L1 cache in terms of memory hierarchy.

A. *Cache vs SPM*

Caches are used to achieve better performance by exploiting the temporal and spatial locality of accesses in a program. Figures 2(a) and 2(b) show the basic components of a cache and SPM respectively.

As seen from the Figure 2(a), the cache is primarily an SRAM array. The tag array and comparator logic in hardware is used to perform the checks to locate a recently used datum. During the execution of the program, this logic is active for each memory access



(a) Cache architecture (b) SPM architecture

Fig. 2. Cache and SPM architecture

made by the program. Thus, it is no surprise that the cache consumes a lot of power while making those checks.

The SPM, on the other hand, does not use any extra logic. It is simply an SRAM array supplemented by the required address decoding logic as shown in Figure 2(b). This explains the lower area and lower power ratings for the SPM. In case of a cache based system, the application is oblivious to the existence of an underlying cache. This enables binary compatibility and portability of applications. The SPM, however, is not transparent to the application as it is an explicitly addressed region of memory in the processor's address space. Thus, the application or the compiler needs to be SPM aware in order to fully utilize this memory. Caches are not preferred in real-time systems due to the tighter timing guarantees demanded by tasks. SPMs are very suitable in these systems as the data present in SPM is always known to the application and can be accessed in a fixed number of cycles. Overall, the compiler plays an important role in the efficient management of SPM.

B. *Challenges in the Compiler*

The advantages of using SPM come with a new set of challenges. In SPMs, data transfers to/from memory have to be explicitly managed by the application. This is challenging, as it may not be possible to predict the data access pattern at compile-time due to the inherently dynamic nature of programs. In order to maximize the power gains by using SPM, it is essential to map data objects that are most frequently referenced onto the SPM.

Each of the types of application data i.e. global, stack and heap have different characteristics and need to be treated separately when mapping to SPM. Global data is 'live' throughout the program execution and is the easiest to map onto SPM. Stack data is more involved as the 'liveness' changes for data objects depending on the call-path traversed. The address of stack objects may not be constant. One can also have multiple instances of the same object (recursive functions). Heap data is the most challenging as the size of allocated object is unknown at compile time.

In this thesis, we propose a solution for managing stack data of an application on SPM. Our experiments in Figure 3 show that stack data enjoys an average of 64.29% of total data accesses for the embedded applications in the Mibench [5] suite.

Mapping data onto SPM is known to be NP-complete. Early techniques proposed static data mapping of stack variables onto SPM [7, 33]. However, in static mapping techniques, the data mapping does not change with time, and hence they are unable to exploit the dynamically changing data access pattern of program. Consequently, dynamic mapping techniques were proposed. However, most dynamic mapping techniques are profile-based [9, 10, 32, 33]. The use of profile limits their scope of application, not only because of the difficulty in obtaining reasonable profiles, but also due to high space and time re-

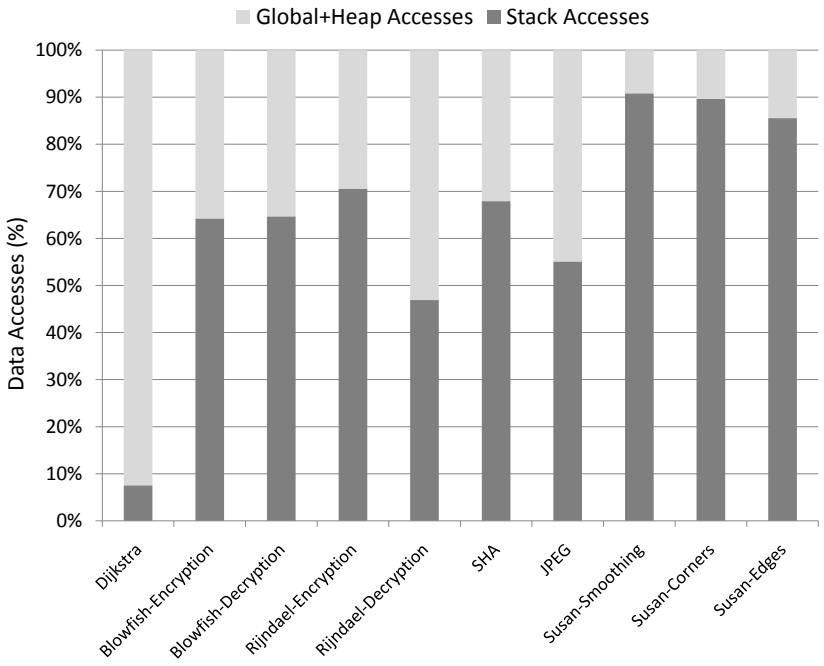


Fig. 3. Stack accesses in MiBench

quirements to generate a profile. Techniques that do not require profile information are preferred; however, there are only a few profile-independent dynamic mapping techniques for SPM. One of them [22] uses static analysis to minimize data transfers between SPM and external memory, but they concentrate on only array data structures and increase re-use in SPM using source transformations. This approach, though effective, works well only in well structured kernels of code. Work in [26] requires hardware support, which in turn reduces its applicability.

While static analysis based, profile-independent dynamic mapping techniques for SPMs are desirable, the challenge is to achieve significant power and performance improvements using them. *In this thesis, we propose a complete software solution for dynamic management of SPM without requiring profile information.* Unlike previous approaches, except for [24, 26], our solution does not require the SPM size until run-time, thus giving

the advantage of binary compatibility. Our approach to map stack data on the SPM is to manage the active stack frames in a circular fashion. The application is enhanced with a software SPM manager at compile-time. When the SPM is filled and unable to accommodate the stack frame for a new function call, a software manager makes space by evicting the oldest frame at the beginning of the SPM to off-chip memory. We achieve an average of 32% reduction in energy with this technique with an average performance improvement of 13%.

Although effective, the SPM manager overhead can be significant in some cases due to the SPM manager calls around each function invocation. We use static analysis to reduce these calls by grouping them. This optimization reduces the software overhead and achieves an average energy reduction of 37% with an average performance improvement of 18%. Applications which have pointers can pose problems when used with circular stack management. It is worth noting that only the pointers to stack variables get affected by circular stack management. In order to ensure correctness of execution in the presence of pointers, we propose an extension to the circular stack management scheme in chapter V.

II. Related work

Banakar et. al [8] provide a comprehensive comparison between caches and scratch pad memories. They demonstrate SPM as an energy efficient alternative to cache and report a performance improvement of 18% with a 34% reduction in area. Figure 4 shows a taxonomy of the SPM mapping techniques.

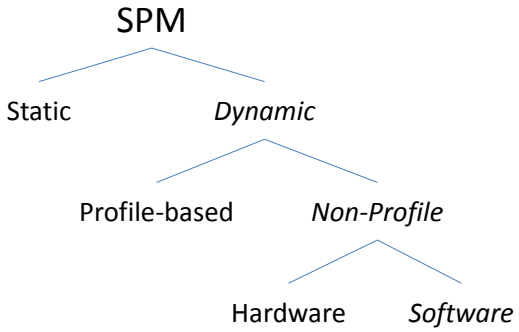


Fig. 4. Taxonomy

All the existing work on SPM mapping can be classified as static and dynamic techniques. Static techniques map certain data objects to the SPM and the contents of SPM remain constant throughout the execution of the program. Dynamic techniques, however, adapt themselves to the changing data access patterns of a program and can change the contents of SPM depending upon the point of execution. It is no surprise that the dynamic techniques in [9, 10, 22, 32] outperform the static technique in [7, 33].

We can further classify the dynamic methods into profile-based and non-profile methods. Stack management has been studied in few works [10, 32] using profile based techniques. Udayakumaran et. al [32] propose a dynamic technique to map global and stack data to SPM. They perform a profile analysis on the application and use it to propose an ILP solution. But, due to the limited scalability of the ILP, they propose a greedy heuristic which achieves near-optimal results in their benchmarks. However, the basis of

both the solutions is in the profile data. An application profile may heavily depend on the input data. The profile based techniques discussed above are proposed to be built into the compiler. It should be noted that getting profile data before every compilation will not only increase complexity of compilation, but also may be infeasible in terms of time. These methods are unable to handle recursive functions and are forced to spill them to the off-chip memory. This inherent limitation is evident in the fact that there is a separate work [10] which maps recursive stack data on SPM. The authors propose to identify the levels of recursion which use a lot of stack data by profiling and map only those stack frames onto SPM. These approaches are a deterrent in target architectures similar to the Cell SPE, TI MSP430 which requires the code or data to be brought into the SPM before accessing it. *Our SPM management technique works well even on this architecture and seamlessly handles recursive as well as non-recursive functions on SPM.* Moreover, all the works using profiling information, except [24] need to know the SPM size at compile-time restricting their binary compatibility. On the other hand, *our technique does not need the SPM size information until run-time.* Thus, there is a need for profile-independent techniques for a scalable and feasible SPM mapping solution. *Our technique does not depend on profiling and thus can scale well for any size of application.*

Work in [9, 12, 26] perform SPM mapping in systems with hardware support from the memory management unit (MMU). Our work is inspired from the approach in [26], where the authors modify the MMU permission fault handler to perform circular stack management. But, the hardware approach works only on systems with MMU limiting its flexibility. This method uses the access permission bits of a page to perform its management. This raises security concerns as any malicious application can take undue advantage

of this management technique to get access to other processes. In [9], stack pages are managed based on profile information and modifying the page fault handler of MMU to bring pages to SPM on demand. Though the hardware techniques show promise, they lack the flexibility and ease of implementation, since they need architectural modification.

Thus, there is a need for developing SPM mapping techniques which are dynamic, profile-independent and built in software. Our technique is a dynamic, profile-independent, pure-software technique which ensures a feasible and scalable solution to the problem of stack data mapping. We present our approach in the next section followed by analysis and experimental results.

III. Circular Stack Management

A. *Active Stack Management*

Our focus is to keep the active stack data on the SPM. In order to keep the book-keeping overhead to a minimum, we consider stack data at the granularity of function stack frames. At first, this may seem too coarse, but we demonstrate significant power-performance savings even at this level.

We will consider a small sample program to explain our technique. Let us consider an SPM of size 128 bytes. Table I shows the stack frame sizes for all the functions in the program and the call graph is shown in Figure 5. Assuming an upward growing (in address) stack, the stack state after the call to F3 is depicted in Figure 6(a). It can be seen that in this example, the stack space requirement of our toy program is much larger than the available SPM size. In order to make room for the stack frame for function F4, we evict the oldest frame(s) in the SPM to the SDRAM. The evicted frames are kept in stack order in a designated SDRAM location. Figure 6(b) shows the state of stack after eviction of the older frames. Similarly, on the return path, when F3 returns, the evicted frames i.e. F1 and F2 need to be brought back from SDRAM into the SPM at their previous location. The movement of data between the SDRAM and SPM is performed in software. Future implementations will incorporate the transfer by means of DMA.

TABLE I
Stack frame sizes for sample program

Function	Frame Size(Bytes)
F1	28
F2	40
F3	60
F4	54

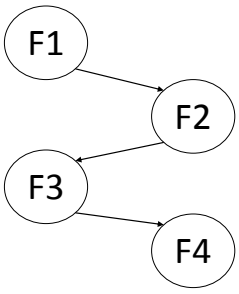
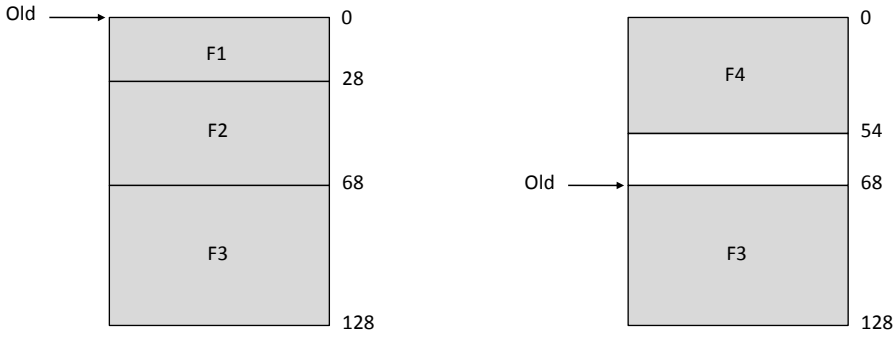


Fig. 5. Sample program call graph



(a) Stack before eviction (b) Stack after eviction

Fig. 6. Stack state for sample program

The decision to remove the oldest frame is facilitated by the natural growth order of call stack. We can evict only the number of bytes required to accommodate the new frame. But, then we have to keep track of all such partial frames. It can also happen that there is insufficient space at the bottom to accommodate a new frame. In such an event, one would think of allocating the frame partially in the remaining space at the bottom and place the rest from the top of SPM. However, this is not possible as the stack management is done at run-time whereas the code has already resolved references to the stack objects with respect to the frame pointer at compile-time. Thus, we choose to perform eviction at frame level and keep the management overhead to a minimum.

B. *Software SPM Manager*

Our technique is pure-software and hence, the management of stack is performed by a software SPM Manager (SPMM). The key functions of SPMM can be summarized as follows.

- Check for available stack space before a function call
- Maintain the exact map of all stack frames residing in SPM/SDRAM
- Keep track of the oldest frame in SPM at all times
- Transfer frames to/from the SPM to SDRAM

The SPMM needs a few important data structures to perform its functions. Some of these structures are populated by the compiler by static analysis whereas others are used by the manager to maintain run-time state.

- **Function Table** - The function table, as the name suggests, holds statically generated information for each function in the application. For each function, the table holds the function's stack frame size and some meta-data required for the pointer extension discussed in chapter V.
- **SPM State List** - This structure is populated and managed at run-time by the SPM Manager. It is the list of all the active stack frames currently residing in the SPM/SDRAM. For each node in the 'SPM State List', we maintain the function Id, its start address in SPM, and linear address. The 'Start Address' is nothing but an address in the range of the SPM and multiple active frames can have overlapping addresses due to circular management. 'Linear Address', on the other hand, is the

start address of frame assuming an infinite SPM size. Thus, the addresses of different active stack frames never overlap. This field is used in pointer extension discussed in chapter V. Another important piece of information is the number of 'Evicted Bytes'. A non-zero value in this field indicates that the function before this node is present in SDRAM. Thus, before returning to that function, SPM Manager needs to fetch this frame from SDRAM to SPM.

The SPMM is implemented as a highly optimized library to be linked with the application. The library provides three basic APIs to carry out manager functions. They are briefly described below:

- **spmm_init()** - This API is used to initialize the SPM Manager structures and generally inserted in the 'main' function of the application. This is also the place where the SPM Manager can query a system register to obtain the SPM size on the target.

- **spmm_check_in()** - This API is used to notify a function call invocation to the SPMM. The SPM Manager uses the function Id to look up the frame size in the 'Function Table'. It uses inline assembly statements to read current values of processor registers SP and registers containing function arguments. After estimating the space available for the next function call, the SPM Manager may evict certain number of frames to SDRAM in order to accommodate the new frame. This manager call needs to be inserted before a user function call in the application.

- **spmm_check_out()** - This API is inserted after each user function call in the application. It essentially updates the SPM State List to indicate successful return from a user function. At the same time, it inspects the 'Evicted Bytes' field and may fetch

```
<asm switch to mgr stack>
 spmm_check_in(F2);
<asm switch to prog stack>
F2();
 <asm switch to mgr stack>
spmm_check_out(F2);
<asm switch to prog stack>
```

Fig. 7. Library calls inserted in application

frames from SDRAM before returning.

Since the SPM Manager queries for the SPM size at run-time, this gives us the advantage of working with an unknown SPM size at compile time making our software portable and binary compatible. The application can thus be supported by the SPMM on any SPM size without the need for re-compilation. The SPM manager library calls are inserted by the compiler in pairs, before and after each function call as shown in Figure 7.

The SPMM call to spmm_check_in() is necessary to check if there is space available for F2 and handle a possible overflow required to accommodate it. When F2 returns, it is necessary for the SPMM to verify that the call returns to a valid stack frame. For example, if we consider the SPM state shown in Figure 6(b), if F3 simply returns, the stack pointer will point to corrupt data. Thus, a check is made inside the spmm_check_out() to detect this situation and fetch the old stack frames from external memory.

The SPMM functions need stack space for their own execution. This is allocated in a reserved area of the SPM. The manager is carefully implemented without using any standard library calls to ensure minimal stack space overhead. Assembly code is inserted as shown in Figure 7 to switch the stack pointer between the 'prog' and 'mgr' stack areas between these calls.

IV. Optimizations

The previous section describes the core functionality of the SPM manager in maintaining the active stack of an application on SPM. The SPM Manager data is mapped permanently to a reserved portion of the SPM to reduce performance overhead. Even so, using the circular management of stack may lead to a performance overhead due to the extra manager library calls before and after each user function call as shown in Figure 7. But, there are opportunities to reduce these overheads by examining the call and control flow of the application.

A. *Call Reduction Opportunities*

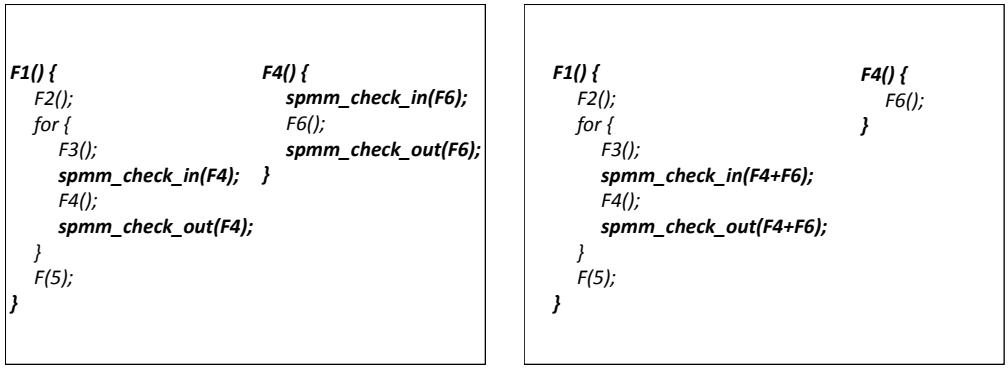
We use the sample program shown in Figure 8 to illustrate the opportunities for call reduction. Using the SPMM technique requires a SPM manager call pair to be inserted around each function call. We would like to reduce the total number of manager calls by consolidating them for a group of functions.

```
F1() {
    F2();
    for {
        F3();
        F4();
    }
    F(5);
}
```

```
F4() {
    F6();
}
F5() {
    if(...){
        F5();
    }
}
```

Fig. 8. Sample program

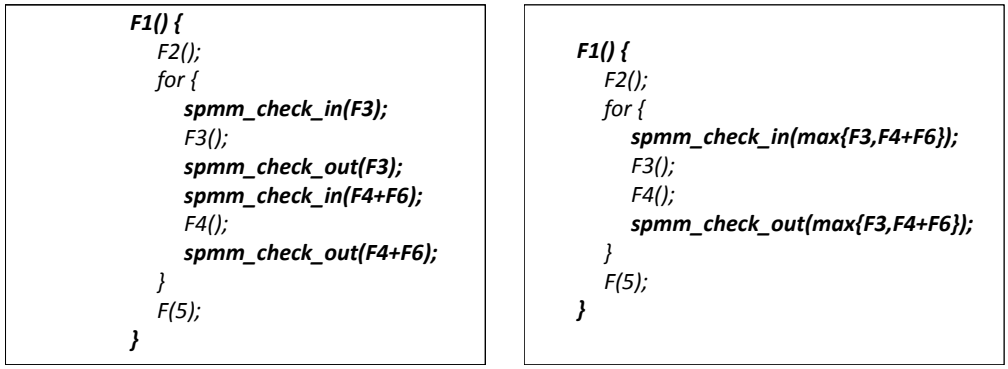
For the example in Figure 8, if we consider F4, it has a nested call to F6. Here, it is possible to avoid inserting a separate manager call around F6 as seen in Figure 9(a), if we can request the space for F6 in the manager call for F4. In this case, the requested stack space will be equal to the requirement of (F4+F6) as shown in Figure 9(b).



(a) Un-optimized code (b) Optimized code

Fig. 9. Opportunity in nested function calls

Another opportunity can be seen in F1, where F3 and F4 are always called in sequence. Here, instead of making multiple manager calls for F3 and F4 as seen in Figure 10(a), we can insert a single call pair around F3 and F4 together, requesting for a stack space of max(F3, F4) as shown in Figure 10(b).



(a) Un-optimized code (b) Optimized code

Fig. 10. Opportunity in sequential function calls

Loops in the program also give an opportunity to avoid repeated manager calls. Since F3 and F4 are executed in a loop, it is possible to make the manager call outside the loop construct. It can be seen how it is possible to chain these individual optimizations

to considerably reduce the manager call overhead. The final optimized version of the instrumented code is shown in Figure 11.

```
F1() {
   F2();
   spmm_check_in(max{F3,F4+F6});
   for {
      F3();
      F4();
   }
   spmm_check_out(max{F3,F4+F6});
   F(5);
}
```

Fig. 11. Final optimized code

B. *Global Call Control Flow Graph*

We introduce a novel data structure called Global Call Control Flow Graph (GCCFG) to perform this analysis. The GCCFG is an extension to the standard control flow graph (CFG). It is a directed graph $G = (V, U, E)$, where an F-node $v \epsilon V$ represents a function, an L-node $u \epsilon U$ represents a loop and a directed edge $e \epsilon E$ in $V \bigcup U$ represents a function call or a nested loop. It is constructed in two simple steps. Firstly, a CFG is constructed for each function. Then, the loop headers and loop branches are identified to form the $L - Nodes$ in the graph. If a function is called inside a loop, the corresponding F-node is joined to the loop header L-node with an edge. Further, all L-nodes representing nested loops, if any, are also joined. F-nodes not inside any loop are joined to the first node of the CFG. For conditional statements, it is assumed that both paths will be executed. The first node, F-nodes , L-nodes and corresponding edges are retained, while all other nodes and edges are removed. This step trims the CFG and retains the control flow and call flow information.

In the next step, all CFGs are merged by combining each F-node with the first node of the corresponding CFG. Recursion can be detected while constructing the graph and is indicated by a self-loop on the F-node. The merge ensures that strict ordering is maintained between the CFGs, i.e. if two functions are called one after another, the first function is a left child and the other function is a right child of the caller function. The GCCFG is an approximate representation of the runtime execution flow of the program. The GCCFG for the sample program in Figure 8 is shown in Figure 12.

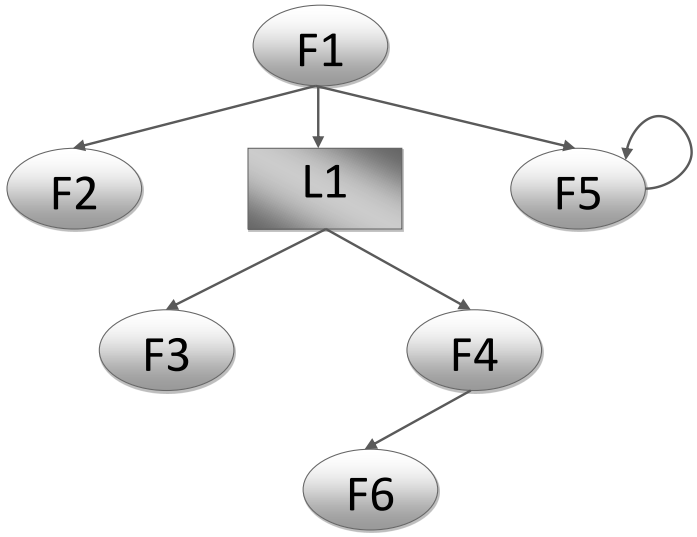


Fig. 12. GCCFG for sample program

Each node $v \epsilon V$, $u \epsilon U$ stores some information required to statically analyze the code and traverse the GCCFG. The members comprising each node are explained below:

- *nodeType* - This field indicates if the node is an F-node or an L-node.
- *frameSize* - This gives the stack frame size for F-nodes. This field has a value of '0' for L-node.
- *requestSize* - This field is populated by the call reduction analysis. It holds the stack

size to be requested from SPMM in the presence of optimizations.

- *parent* - This field holds a pointer to the parent F-node. In case of nested loops, the parent will be the containing function.
- *recursive* - This flag is set if the F-node is recursive. It is ignored for L-nodes.
- *start* - This field holds the start address for F-nodes during static analysis.
- *children* - This field holds the list of children F-nodes/L-nodes in call order.

C. *Manager Call Consolidation*

Now that we have identified the circumstances in which optimization is possible, we outline an algorithm which will systematically explore the GCCFG and insert the manager calls only where absolutely necessary. In order to perform the manager call consolidation, we explore the GCCFG in a depth-first fashion (done by the routine *Consolidate* shown in Algorithm 1). Starting from the leaf functions, we check to see if any of the aforementioned optimizations are possible and if so, fill the *requestSize* field. It must be noted that each call instance of the same function may be optimized differently depending upon its parent and siblings i.e. the call path traversed. But, the optimization inside a particular function will be performed once, when the graph first explores the function and will remain unchanged thereafter.

Once the GCCFG exploration is complete, the *requestSize* field of each node indicates the action to be taken as given in Table II.

We can now insert the appropriate manager calls by exploring each of the GCCFG nodes for the *requestSize* field. The exploration always starts at the F-node representing

TABLE II
GCCFG REQUEST SIZE FIELD

Value of *requestSize*	Action
$= 0$	Insert manager call before this node using *frameSize* value (applies only to F-nodes)
> 0	Insert manager call before this node using *requestSize* value
$= -1$	Do not insert manager call

the 'main' function of the application. The algorithm for the manager call consolidation follows:

Algorithm 1 Consolidate (V_f)

1: **for all** children, $V_i \epsilon$ children(V_f) **do**
2: *Consolidate*(V_i)
3: **end for**
4: *Classify*(V_i)

The routine 'ComputeStackReq' in Algorithm 3 computes the maximum stack space required by children of a node. This routine also detects recursion and marks a flag in the GCCFG node. This information is used by the routine 'Classify' in Algorithm 2 to check if the given size of SPM can hold both, the node and its children's stack in the available space. The conditional statement in step 4 and step 11 of 'Classify' checks to see if there is enough space either before or after the parent function. This is important as any optimization should not end up requiring eviction of its immediate parent's stack frame.

To understand this, let us go back to the sample program in Figure 8. Consider that there are a few statements between F3 and F4 which access F1's stack frame. Now, if the consolidation of manager calls to F3 and F4 lead to eviction of the stack frame of F1, the program will access corrupt stack data when executing the statements between F3 and F4. This does not happen in the un-optimized case, as we call the manager immediately after

Algorithm 2 Classify (V_f)

```
1: ComputeStackReq(V_f)
2: if (nodeType(V_f) is L-Node then
3:    Find parent F-node V_p for V_f
4:    if (condition)† then
5:       requestSize(V_f) = stackReq
6:       for all F/L-Nodes, V_i ϵ children(V_f) do
7:          requestSize(V_i) = −1
8:       end for
9:    end if
10: else if (nodeType(V_f) is F-Node then
11:    if (condition)† then
12:       requestSize(V_i) = frameSize(V_i) + stackReq
13:       for all F/L-Nodes, V_i ϵ children(V_f) do
14:          requestSize(V_i) = −1
15:       end for
16:    end if
17: end if
```

†*Check to see if there is enough space for the children function(s) either before or after the parent function in SPM.*

returning from F3. Here, if the stack frame of F1 was evicted, the manager would fetch it from external memory before proceeding ahead.

In the event that the maximum program stack requirement is less than the SPM size, the algorithm would suggest insertion of only one consolidated manager call at the 'main' function. Thus, for such cases, the SPM manager overhead is at its minimum. Since this analysis is carried out at compile time, it is not possible to optimize for recursive functions as the depth of recursion may vary with program input. We therefore leave the recursive functions un-optimized.

This optimization is implemented as a compiler pass which scans the source files to generate the GCCFG. Once the GCCFG is generated, for a given SPM size, the manager call consolidation algorithm is applied to insert the SPMM calls. The compiler then generates the Function Table for the application which is embedded into the binary. The Function

Algorithm 3 ComputeStackReq (V_f)

```
1: stackReq = 0
2: for all F/L-Nodes, V_i ϵ V_f do
3:    if recursive(V_i) is TRUE then
4:       stackReq = SPMSize
5:       break
6:    end if
7:    if requestSize(V_i) > 0 then
8:       size = requestSize(V_i)
9:    else
10:      size = frameSize(V_i)
11:   end if
12:   stackReq = max{stackReq,size}
13: end for
14: if stackReq = 0 then
15:   stackReq = SPMSize
16: end if
```

Table generated here has entries for each function as well as entries for certain consolidated blocks (loops, groups of functions).

It is also important to note that the target SPM size is required at compile-time to perform this optimization. Thus, the application programmer has a choice to optimize if he knows the system SPM size. If not, the programmer can always fall back on the base method described in chapter III.

V. Extension for Pointer support

The stack management technique described in chapter III manages the active portion of application stack on SPM. Applications which contain pointers can be used with this approach, but need more analysis in order to ensure correctness of execution. Correctness can be a concern with SPMM in presence of pointers, since the stack data is managed in a circular fashion and stack frames change their locations during execution. This may cause certain pointers to stack data to become invalid. In the following sections, we propose an extension to the circular stack management technique to handle pointers and illustrate using an example in section B.

A. *Run-time Pointer-to-stack Resolution*

We would like to expand the applicability of the circular stack management technique described in chapter III to applications with pointers. Our technique is for the management of stack data and hence, we consider all the pointer variables pointing to stack data. We need to analyze the source code to detect these pointers. The extension proposed in this section has a few restrictions and is unable to handle all possible cases of programmatically using stack values by pointers. We discuss these limitations and the underlying assumptions before explaining the extension.

- Functions will access data from other stack frames only through use of pointers passed as arguments to it.
- The source language is strongly typed (no type-casting). We cannot detect pointers if they are disguised as other types when passed in function arguments.
- Pointers to stack data are not passed within other structures.

The assumptions given above may seem too restrictive when applied to programs written in C, but actually, they conform well with good programming practices to be followed. The extension comprises of two components:

- Compile-time analysis to detect function signatures (function prototypes).
- Run-time analysis to resolve pointer-to-stack addresses.

A.1. *Compile-time Component*

Since we assume that the pointers-to-stack data originate and propagate only through function arguments, it is essential to know all the function signatures of the application. We obtain this information by parsing this information from the abstract syntax tree (AST) of the application. For each function, we record the type of each argument and add this information to the SPMM Function Table data structure described in section B. The following subsection describes the run-time component which uses this function signature information.

A.2. *Run-time Component*

The run-time component is activated as part of the SPMM library calls themselves. The function of this component is to validate all pointer arguments in the function signature. We use the SPM State List structure of the SPMM for this validation. The SPM State List described in section B holds vital information about the current call path executing in the program. For this discussion, the 'owner frame' is the frame to which the pointed-to-data of a pointer belongs. The validation is comprised of three simple steps:

1. We first find the owner frame for that pointer argument value. The SPM State List stores the starting address for each frame and also knows the size of all the frames

in the current call graph path. Since it is possible that multiple frames in the SPM State List may contain this address, we scan the list in the reverse direction (i.e. latest node to oldest node).

2. Once the owner frame is found, we need to check if it is present in SPM/SDRAM. This can be achieved by inspecting the 'Evicted Bytes' field of the next frame. If this field is greater than zero, it indicates that the frame is currently residing in SDRAM. If 'Evicted Bytes' is zero, it implies that the function resides in SPM and the pointer is still valid causing no further action to be taken.

3. The last step is to compute the new address and is necessary if the owner frame was detected to have moved to SDRAM. The evicted frames are stored in stack order in SDRAM in a linear fashion. The 'Linear Address' field in the SPM State List gives the start address of a frame assuming an infinite SPM starting from address 0. We can now simply add the 'Linear Address' value to the SDRAM eviction area base address to locate the owner frame in SDRAM. Now, it is possible to find the new address of the pointer since the offset within the frame remains unchanged.

B. *Illustrative example*

Let us consider the sample program in Figure 13 to understand the pointer problem. In order to succinctly explain the problem, we construct a toy program which is recursive in nature. However, it is important to note that the extension proposed can also work with non-recursive programs. The Table III gives the stack frame sizes of the functions in the sample program. Let the SPM size be 256 bytes.

TABLE III
STACK FRAME SIZES FOR POINTER SAMPLE PROGRAM

Function	Frame Size(Bytes)
main	40
ptrRecursion	28

```
int main(void) {
    int k = 8;
    int var1 = -1, var2 = -2;
    int *ptrVar2 = &var2;
    int **p_ptrVar2 = &ptrVar2;

    ptrRecursion(k,&var1,p_ptrVar2);
    printf("%d %d",var1, var2);
}
void ptrRecursion(int k, int *ptrVar1, int **p_ptrVar2) {

    if (k == 1){
        *ptrVar1 = 1000;
        **p_ptrVar2 = 2000;
        return;
    }
    ptrRecursion(--k,ptrVar1,p_ptrVar2);
}
```

Fig. 13. Pointer sample program

In Figure 13, the value of the variable k in the function *main* decides the level of recursion i.e. the stack depth. Pointers to local variables of *main* viz. $var1$ and $var2$ are passed to the function *ptrRecursion*. The pointer to $var2$ in the third argument is passed as a two-level pointer reference, whereas that of $var1$ in the second argument is a single level pointer reference. At the tail of the recursion, the values of local variables $var1$ and $var2$ are changed through their respective pointers inside *ptrRecursion*. This example uses the common programming practice of using pointers to local variables and reading/writing to them in other functions. Essentially, the function stack for the active function accesses data in other stack frames in its call path.

Given the stack frame sizes in Table III and the SPM size of 256 bytes, the stack depth is $depth = framesize(main) + k * framesize(ptrRecursion)$. In this example, for $k = 7$, the value of $depth = 236$ bytes and will not cause a stack overflow. But, if $k = 8$, the *depth*

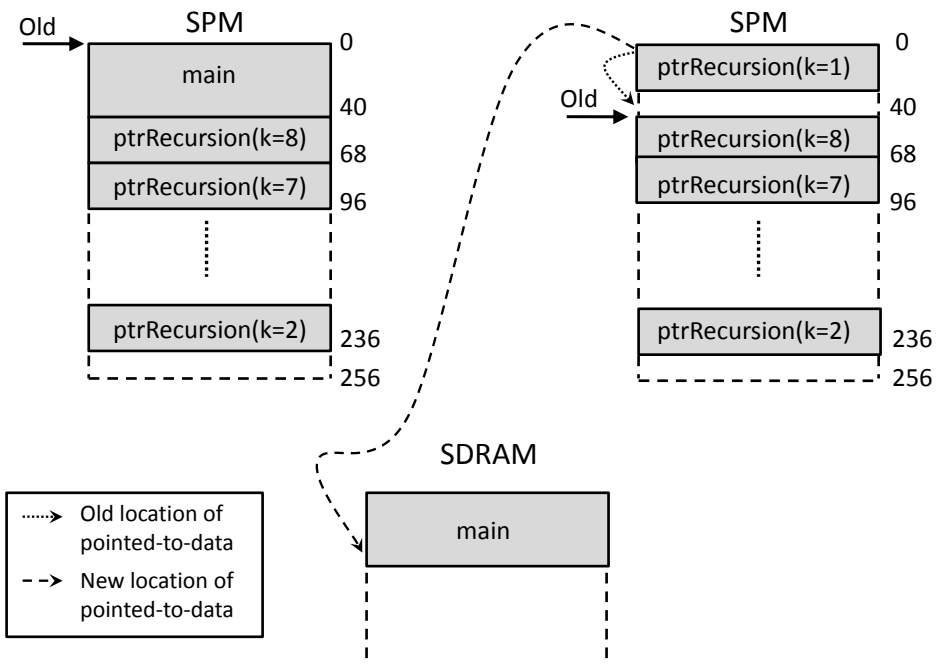


Fig. 14. Pointer sample program stack state

value is greater than 256 bytes and will try to overflow the SPM stack. The SPMM will accommodate the new stack frame by evicting the oldest frame in the SPM (i.e. *main*) as shown in Figure 14. The new frame i.e. *ptrRecursion* with $k = 1$ receives the address of $var1$ and $ptrVar2$ of *main* in its arguments. But, the SPMM has already moved the stack frame of *main* to SDRAM. In this case, any writes/reads using the pointer arguments will cause a corruption of stack or incorrect operation of the program. This is shown by the dotted lines in Figure 14.

In order to ensure correctness, it is necessary for the SPMM to update the pointer address argument such that the reads/writes will happen at the correct location in SDRAM. This is shown by the dashed lines in Figure 14. For single level pointers like the second argument ($int * ptrVar1$)of *ptrRecursion*, this will suffice. But, for multi-level pointers like the third argument ($int * *p\_ptrVar2$), SPMM needs to update the address at each level of

de-reference.

It is important to note that we are concerned with only pointers to stack data. All other pointers in the application (pointers to heap data, pointers to global data) are not a problem since the SPMM never touches those data. We already have the function signature information for each function generated at compile-time and stored in the Function Table structure. This tells us the pointer arguments present in each function which need to be validated during run-time.

For the correct execution of application, it is necessary for stack data references shown by the dotted line in Figure 14 be re-directed to the location shown by the dashed line. This implies that the SPMM should change the value of the pointer argument that is to be passed on to the Function *ptrRecursion*.

Let us assume the address values of the local variables of *main* as $address(var1) = 20$, $address(var2) = 24$ and $address(ptrVar2) = 28$. The SPMM is called (*spmm_check_in*) before the function call to *ptrRecursion (k = 1)*. Due to insufficient space, the SPMM evicts *main*, causing the local variables of *main* to now reside at an SDRAM location changing their addresses to $address(var1) = 4020$, $address(var2) = 4024$ and $address(ptrVar2) = 4028$. However, the pointer argument values for Function *ptrRecursion (k = 1)* still hold the old addresses. This is the point where the SPMM call has to update these values before letting Function *ptrRecursion (k = 1)* start its execution.

We use the SPM State List structure described in section B to achieve this. The SPM State List holds the entire list of functions in the active call path traversed as shown in Figure 15. We also need to query the Function Table to get the list of pointer arguments to be inspected. For each pointer argument in the function *ptrRecursion*, we perform three

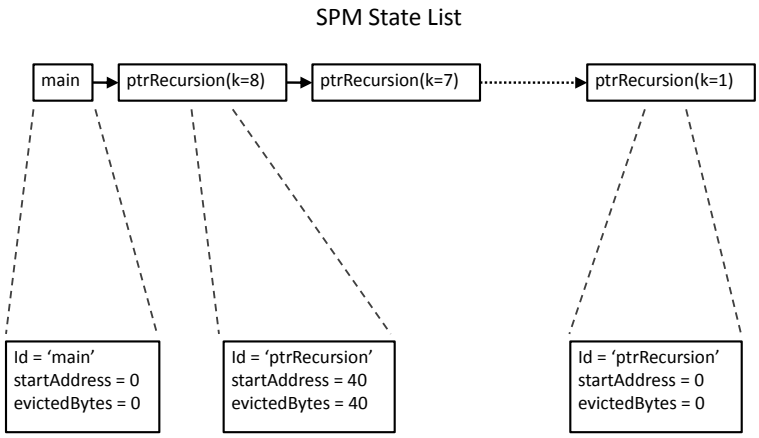


Fig. 15. SPM state list

simple steps:

1. We first want to find the owning frame for that pointed-to-data. Since, we store the starting address for each frame and also know the size of the frame, this is achievable by simply scanning the list. In the example above, the owner frame is *main*.

2. We need to check the current location of *main* i.e. if it is present in SPM/SDRAM. This can be achieved by inspecting the 'Evicted Bytes' field of the next frame i.e. *ptrRecursion(k=7)*. If this field is greater than zero, it indicates that *main* is currently residing in SDRAM. If 'Evicted Bytes' is zero, it implies that the function resides in SPM and we do not need to update the address.

3. The last step is to compute the new address. The evicted frames are stored in stack order in SDRAM in a linear fashion. The 'Linear Address' field in the SPM State List gives the start address of a frame assuming an infinite SPM starting from address 0. We can now simply add the 'Linear Address' value to the SDRAM eviction area base

address to locate *main* in SDRAM. Once found, we can find the new address of the local variables $var1$ and $ptrVar2$ and $var2$.

After finding the new address, the SPMM modifies the argument values passed to the function *ptrRecursion(k=1)*. An important aspect of this method is that the pointers need to be updated only once. In the sample program given in Figure 13, if $k > 8$, then too, the pointers are updated when the pointed-to-data changes its address. But the pointers do not need to be updated for every subsequent frame, since the new argument values are being propagated after the update.

VI. Experiments

A. *Experimental Setup*

We use the cycle-accurate SimpleScalar simulator [6] to model an architecture without a cache, but consisting of an SPM and an external SDRAM memory. The processor uses the ARMV5TE ISA [4]. The static analysis algorithm is implemented as a pass during the compilation using the GCC compiler ported for ARM. We use the MiBench suite [5] of embedded applications to demonstrate the effectiveness of our technique. Table IV shows the maximum stack depth and SPM size used for different benchmarks.

TABLE IV
Benchmarks

Name	Stack Depth(Bytes)	SPM Size(Bytes)
Dijkstra	424	256
Blowfish-Encryption	12440	8192
Rijndael-Encryption	796	1024
Blowfish-Decryption	11984	8192
Rijndael-Decryption	812	1024
SHA	2240	2048
JPEG	10570	8192
Susan-Smoothing	14380	12288
Susan-Corners	14124	12288
Susan-Edges	14960	12288

B. *Energy Models*

We use the CACTI tool [30] for the SPM energy model with 0.13μ technology. For an SPM of size 1k, the energy per read ($E_{SPM/RD}$) and write ($E_{SPM/WR}$) access are 0.33nJ and 0.13nJ respectively. It should be noted that the per access energy increases with SPM size. The external memory energy model is for a 64MB Samsung $K4X51163PC$ SDRAM [28]. The energy per read burst($E_{SDR/RD}$) for the SDRAM is 3.3nJ, whereas, a write burst($E_{SDR/WR}$) is 1.69nJ. The following equations are used to calculate energy consumed:

$$E_{TOTAL} = E_{SPM-TOTAL} + E_{SDR-TOTAL}$$

$$E_{SPM-TOTAL} = (N_{RD} * E_{SPM/RD}) + (N_{WR} * E_{SPM/WR})$$

$$E_{SDR-TOTAL} = (N_{SDR-RD} * E_{SDR/RD}) + (N_{SDR-WR} * E_{SDR/WR})$$

C. *Results and Analysis*

We evaluate the effectiveness of the circular stack management technique and the consolidation algorithm by comparing the energy consumption and performance improvement for:

1. System with only SDRAM, $1k$ cache (Baseline)
2. System with SPM and circular stack management (SPMM)
3. System with optimized circular stack management (GCCFG)
4. System with circular stack management and pointer extension (SPMM-Pointer)

The SPM sizes are chosen such that they are at least as much as the largest function stack frame in the benchmark. Figure 16 shows the normalized energy reduction obtained for the benchmarks.

The average reduction in energy using SPMM against the Baseline is 32% with a maximum energy reduction of 49% for the SHA benchmark. The dynamic profiling based technique in [10] focuses on mapping recursive stack data to SPM and achieves an average reduction in energy of 31.1%. It should be noted that the authors suggest taking multiple profiles and averaging them in order to reduce profile dependence In contrast, we achieve

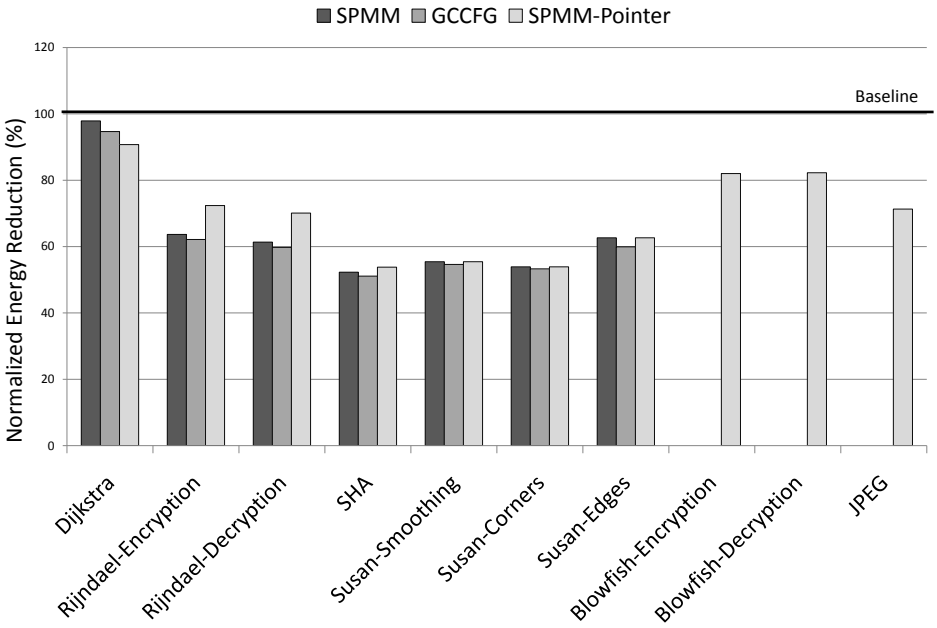


Fig. 16. Normalized energy reduction

32% energy savings by simply managing the entire stack from SPM seamlessly for recursive and non-recursive functions without the time-consuming profiling process. Also, our solution does not require the SPM size till run-time making it binary compatible.

We improve upon our results further by reducing the SPM manager calls using the consolidation algorithm. We observe a further average energy reduction of 5%. The SHA benchmark contains many nested function calls within loop structures making it a good candidate for optimization using our consolidation algorithm. It should be noted that the GCCFG consolidation reduces only the SPM manager call overheads while the data movement between SPM and SDRAM in case of overflows remains constant. Call consolidation causes evictions to occur in bigger chunks. This happens so, because the manager may allocate and de-allocate stack space for groups of functions rather than individual functions. The SPM manager function table is accessed from SDRAM whereas only a limited set of

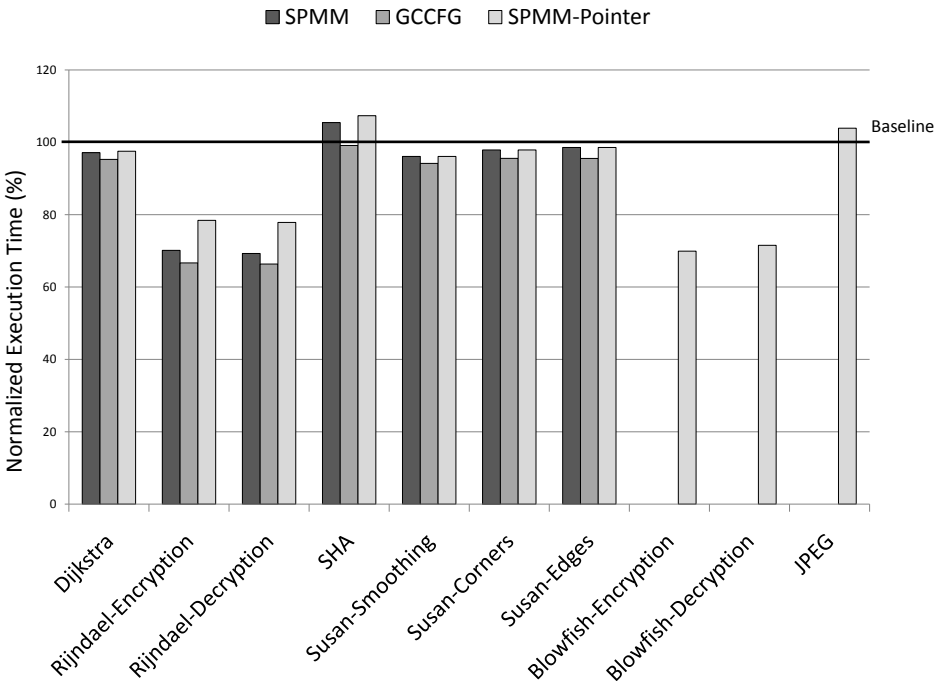


Fig. 17. Normalized performance overhead

manager data objects are kept in SPM. This is done to keep a minimal space overhead in SPM. The overhead of the SPM Manager is well compensated for by the reduction in total number of SDRAM accesses.

The performance trends shown in Figure 17 are normalized with the Baseline case. It is important to note that the performance obtained using the Baseline system is $16x$ better than a system without any on-chip memory. In case of processors which only have an SPM and no cache, our technique is extremely beneficial for performance as well as power.

We observe an average performance improvement of 13% for SPMM technique with a maximum improvement of 34% for Blowfish-Decryption. It is interesting to note that the hardware assisted circular stack management in [26] achieves a similar performance improvement. But, our solution does not require any hardware support and can be ported to any architecture using SPM. We observed performance degradation up to 6% in the

SHA and JPEG benchmarks. But, the manager call consolidation algorithm completely eliminates this degradation and results in further average performance improvement of 5%.

The pointer extension proposed in chapter V is extremely useful in situations where the application programmer is already given a fixed SPM size on the system. If the application uses pointers to stack data and wants to use the Circular Stack Management, this may lead to incorrect execution as demonstrated in section B unless the suggested extension is used with SPMM. Due to the SPMM with pointer extension (SPMM-Pointer), we are able to run more benchmarks like Blowfish and JPEG which use a lot of pointer-to-stack references.

We observe an average energy reduction of 29.6% with SPMM-Pointer. The reduced energy savings by 3.3% as compared to SPMM can be attributed to the extra instructions executed to validate pointers in the program during the SPMM calls. It is no surprise that we also see reduced performance improvement from 13% for SPMM to 10% for SPMM-Pointer. However, as pointed out before, a programmer can now run his application with pointers even when he does not have the liberty to choose the SPM size.

VII. Conclusions

A simple, yet effective, dynamic circular stack management scheme which does not require system SPM size at compile-time was proposed. The static analysis method to reduce the software overhead achieved average energy reduction of 37% with an average performance improvement of 18%. The stack management demonstrated is not restricted to cache-less architectures and can also be used in general purpose systems and scales well with application size.

There are many interesting dimensions to extend this method. The pointer extension suggested can be combined with static analysis methods to detect all types of pointers and resolve them at run-time. When the stack frame size is greater than SPM itself, the function stack cannot be brought into the stack and needs to be used from the main memory. One can investigate approaches to break the function stack and bring it into the SPM in parts.

The solution presented is promising as there is a clear need, but lack of SPM mapping techniques which are dynamic, profile-independent, pure software and binary compatible.

REFERENCES

[1] Intel Corporation. The Intel IXP1200 Family of Network Processors - Product line. http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] Apple. Apple iPhone. http://www.apple.com/iphone/.

[4] ARM. Advanced RISC Machines (ARM) ARM926EJ-S Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198d/DDI0198_926_TRM.pdf.

[5] T. Austin. SimpleScalar LLC. http://www.simplescalar.com/.

[6] T. Austin. SimpleScalar LLC. http://www.simplescalar.com/.

[7] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.

[8] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. *CODES '02: Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78, 2002.

[9] H. Cho, B. Egger, J. Lee, and H. Shin. Dynamic data scratchpad memory management for a memory subsystem with an mmu. *SIGPLAN Not.*, 42(7):195–206, 2007.

[10] A. Dominguez, N. Nguyen, and R. K. Barua. Recursive function data allocation to scratch-pad memory. *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 65–74, 2007.

[11] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, 2005.

[12] 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. *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 223–233, 2006.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[14] M. Hind. Pointer analysis: haven't we solved this problem yet? *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[15] IBM. PowerPC. http://www.ibm.com/developerworks/linux/library/l-powarch/.

[16] IBM. The Cell project at IBM Research. http://www.research.ibm.com/cell/.

[17] Intel. Core2 Duo. http://www.intel.com/products/processor/core2duo/index.htm.

[18] Intel. Intel Penryn 45nm. http://www.techwarelabs.com/reviews/processors/penryn-preview/.

[19] Intel. Intel Quad-Core. http://www.intel.com/technology/quad-core/.

[20] Intel. Intel Tera-Scale 80-core processor. http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

[21] M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. *DAC '02: Proceedings of the 39th Conference on Design Automation*, pages 219–224, 2002.

[22] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. *DAC '01: Proceedings of the 38th Conference on Design Automation*, pages 690–695, 2001.

[23] Y. Meng, T. Sherwood, and R. Kastner. Exploring the limits of leakage power reduction in caches. *ACM Trans. Archit. Code Optim.*, 2(3):221–246, 2005.

[24] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *CASES '05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 115–125, 2005.

[25] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, 2000.

[26] S. Park, H. woo Park, and S. Ha. A novel technique to use scratch-pad memory for stack management. *DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1478–1483, 2007.

[27] D. J. Pearce, P. H. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[28] Samsung. Samsung Semiconductor Memory. K4X51163PC Mobile DDR Synchronous DRAM. http://www.samsung.com/products/semiconductor/MobileSDRAM/2005.

[29] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, 2008.

[30] P. Shivakumar and N. Jouppi. Cacti 3.2. http://www.hpl.hp.com/research/cacti/.

[31] Sony. Sony PlayStation3. http://www.us.playstation.com/PS3.

[32] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 276–286, 2003.

[33] M. Verma, S. Steinke, and P. Marwedel. Data partitioning for maximal scratchpad usage. *ASPDAC: Proceedings of the 2003 Conference on Asia South Pacific Design Automation*, pages 77–83, 2003.