A Software-Only Solution to Use Scratch Pads for Stack Data

Aviral Shrivastava, Arun Kannan, and Jongeun Lee

***Abstract*—A dynamic scratch pad memory (SPM) management scheme for program stack data with the objective of processor power reduction is presented. Basic 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. Stack frames are managed using a software SPM manager, integrated into the application binary, and shows average energy savings of 32% along with a performance improvement of 13%, on benchmarks from MiBench. SPM management can be further optimized and made pointer safe, by knowing the SPM size.**

***Index Terms*—Cache, compilers, embedded systems, scratch pad memory (SPM), static analysis.**

I. Introduction

POWER and temperature issues have been at the crux of the inevitable transition to multicores. Increasing power and temperature of processors had started to very significantly affect the performance, and multicores provided the only road ahead to still improve peak performance without much increase in the power consumption. While okay for few-core architectures, cache coherency protocols do not scale with the number of cores on a chip. Consequently, modern many-core processors are being designed with a distributed memory model, in which there is no hardware mechanism for memory virtualization. For example, the experimental Intel 80-core processor [21], network processors, like the Intel IXP1200 [17], and the IBM Cell processor that is used in the Sony Playstation 3 [18] feature local memories in each core. The data transfers between the main memory and the core have to be explicitly specified in the application by the programmer or by the compiler. Such memories are called scratch pad memories (SPMs) by the compiler community. Another equally important reason for using SPMs instead of caches is to avoid the very significant power overhead of the hardware memory management in caches [1], [3].

We observe that 10%–90%, averaging to 64.29%, of all accesses in multimedia applications [15] are to stack variables. 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. However, it is not always possible and difficult at other times to predict the data access patterns at compile time. Stack data are one such unpredictable data. Not only is the amount of stack data required by the application nondeterminable at compile time, the function call pattern is often data dependent, and cannot be determined at compile time. Early techniques proposed static data mapping of stack variables onto SPM [9], [10]. However, in static mapping techniques, the data mapping does not change with time, hence static mapping techniques 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 [4], [6], [8], [10]. 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 requirements 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 [5] uses static analysis to minimize data transfers between SPM and external memory, but they concentrate on only array data structures and increase reuse in SPM using source transformations. This approach, although effective, works well only in well-structured kernels of code. Work in [2], [6], [7] requires hardware support, which, in turn, reduces its applicability.

In this paper, we propose a complete software solution for dynamic management of SPM without requiring profile information. Unlike previous approaches, except for [2], [11], 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 (SPMM) 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 management overhead can be significant in some cases due to the SPMM 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%; however, it needs to know the SPM size at compile time. In addition, there can be problems in applications that have pointer-to-stack

Manuscript received November 27, 2008; revised June 11, 2009 and July 23, 2009. Current version published October 21, 2009. This paper was recommended by Associate Editor L. Benini.

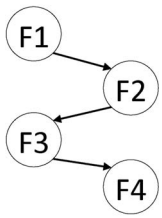
A. Shrivastava is with the Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: Aviral.Shrivastava@asu.edu).

A. Kannan is with the Advanced Visual Computing Group, Intel Corporation, Santa Clara, CA 95054-1549 USA.

J. Lee is with Ulsan National Institute of Science and Technology, Ulsan 689-805, Korea.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2030592



Function Name	Stack Frame Size (bytes)
F1	28
F2	40
F3	60
F4	54

Fig. 1. Sample program call graph.

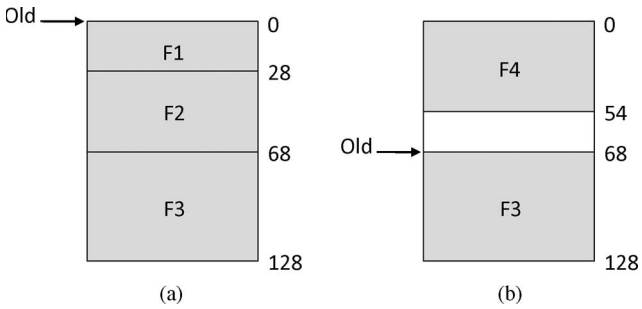


Fig. 2. Stack state for sample program before and after eviction. (a) Stack before eviction. (b) Stack after eviction.

variables of a different function frame. We further propose an extension to the circular stack management scheme that ensures correctness of execution. Our efficient pointer management scheme only incurs slight energy and performance overhead.

II. 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 B. Fig. 1 shows the call graph of the program and the stack frame sizes for all the functions. Assuming an upward growing (in address) stack, the stack state after the call to F3 is shown in Fig. 2(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. Fig. 2(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 direct memory access.

The decision to remove the oldest frame is facilitated by the natural growth order of call stack. We could evict only the number of bytes required to accommodate the new frame. However, 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 SPMM

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

1) Check for available stack space before a function call.
2) Maintain the exact map of all stack frames residing in SPM/SDRAM.
3) Keep track of the oldest frame in SPM at all times.
4) 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.

1) **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 metadata required for the pointer extension. This is discussed in more detail in Section IV.
2) **SPM State List**—This structure is populated and managed at run-time by the SPMM. 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 Section IV. Another important piece of information is the number of "Evicted Bytes." A nonzero value in this field indicates that the function before this node is present in SDRAM. Thus, before returning to that function, SPMM 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 Application Programming Interface functions (APIs) to carry out manager functions. They are briefly described as follows.

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

```
<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. 3. Library calls inserted in application.

2) **spmm_check_in()**—This API function is used to notify a function call invocation to the SPMM. The SPMM 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 SPMM 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.
3) **spmm_check_out()**—This API function 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 frames from SDRAM before returning.

Since the SPMM 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 recompilation. The SPMM library calls are inserted by the compiler in pairs, before and after each function call, as shown in Fig. 3.

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 Fig. 2(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 Fig. 3, to switch the stack pointer between the "prog" and "mgr" stack areas between these calls.

III. SPMM Call Consolidation

The previous section describes the core functionality of the SPMM in maintaining the active stack of an application on SPM. The SPMM data are 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 Fig. 3.

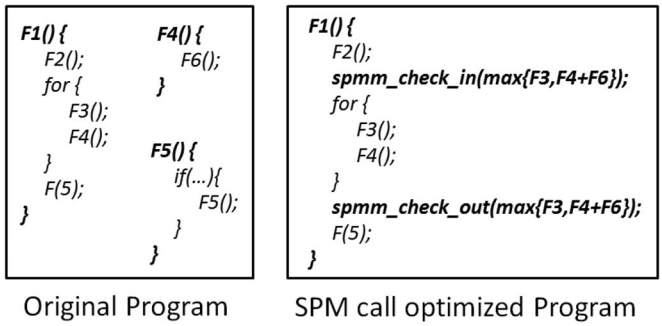


Fig. 4. Program before and after call consolidation.

However, there are opportunities to reduce these overheads by examining the call and control flow of the application.

A. Call Reduction Opportunities

SPMM technique requires an SPMM call pair to be inserted around each function call. In order to reduce the SPM management overhead, we aim to reduce the total number of manager calls by consolidating them for a group of functions. There are at least three situations in which the SPM management call can be consolidated. If two functions will be called sequentially, then the SPM management for both the functions can be done in one call. However, the manager must request space equal to the sum of stack sizes of the two functions. For example, in Fig. 4, functions $F3$ and $F4$ are called in sequence. SPMM calls can be consolidated by performing only one SPMM call before $F3$ request space for maximum of stack sizes of functions $F3$ and $F4$. Another consolidation opportunity is when two functions are called in a nested fashion. For example, in Fig. 4, function $F4$ always calls function $F6$. Separate SPMM calls for the two functions can be consolidated into one call, before $F4$, by a manager call to request space equal to the sum of the stack sizes of $F4$ and $F6$. 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. The final optimized version of the instrumented code is shown on the right side in Fig. 4.

B. Manager Call Consolidation

Now that we have identified the circumstances in which optimization is possible, we outline an algorithm which will systematically insert the manager calls only where absolutely necessary. We use global call control flow graph (GCCFG) [25] to perform this analysis. 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. However, the optimization inside a particular function will be performed once, when the graph first explores the function and will remain unchanged thereafter.

TABLE I
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

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

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 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: $Classify(V_i)$

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, $V_i \epsilon$ 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_f) = frameSize(V_f) + stackReq
13: **for all** F/L-Nodes, $V_i \epsilon$ 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.*

Algorithm 3 ComputeStackReq (V_f)
1: stackReq $= 0$
2: **for all** F/L-Nodes, $V_i \epsilon V_f$ **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**

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 Fig. 4. 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 unoptimized case, as we call the manager immediately after 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 SPMM 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 unoptimized.

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 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 Section II.

IV. EXTENSION FOR POINTER SUPPORT

The stack management technique described in Section II manages the active portion of application stack on SPM. Correctness can be a concern with SPMM in presence of pointers, since the stack data are managed in a circular fashion and stack frames change their locations during execution. This may cause certain pointers to stack data to become invalid.

TABLE II
STACK FRAME SIZES FOR THE SAMPLE POINTER 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. 5. Sample pointer program.

A. Pointer Problem

In order to succinctly explain the problem, we construct a toy program which is recursive in nature. However, the extension proposed will also work with nonrecursive programs. Table II gives the stack frame sizes of the functions in the sample program. Let the SPM size be 256 B.

In Fig. 5, 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 II and the SPM size of 256 B, the stack depth is $depth = framesize(main) + k * framesize(ptrRecursion)$. In this example, for $k = 7$, the value of $depth = 236$ B and will not cause a stack overflow. However, if $k = 8$, the $depth$ value is greater than 256 B 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 Fig. 6. The new frame, i.e., *ptrRecursion* with $k = 1$ receives the address of $var1$ and $ptrVar2$ of *main* in its arguments. However, 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 Fig. 6.

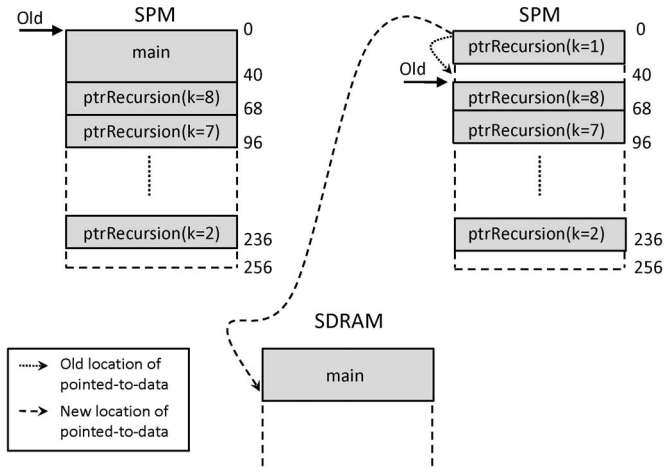


Fig. 6. Stack state of the sample pointer program.

B. Pointer Resolution

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. The extension comprises of two components.

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

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 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 II-B. The following section describes the run-time component which uses this function signature information.

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 II-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,

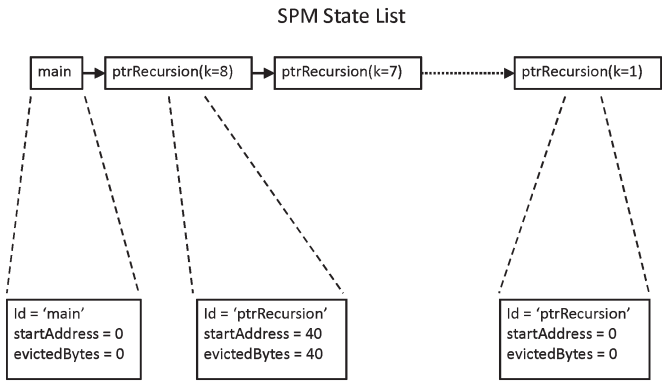


Fig. 7. State list in the SPM.

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.

For the correct execution of application in Fig. 5, it is necessary for stack data references shown by the dotted line in Fig. 6 be redirected 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 II-B to achieve this. The SPM State List holds the entire list of functions in the active call path traversed, as shown in Fig. 7. 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 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 shown in Fig. 5, if $k > 8$, then too, the pointers are updated when the pointed-to-data changes its address. However, the pointers do not need to be updated for every subsequent frame, since the new argument values are being propagated after the update.

C. Assumptions and Limitations

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. Our extension works under the following assumptions.

1) Functions will access data from other stack frames only through use of direct pointers passed as arguments to it. For single-level pointers like the second argument $(int * ptrVar1)$ of *ptrRecursion*, this will suffice. However, for multilevel pointers like the third argument $(int * *p\_ptrVar2)$, SPMM needs to update the address at each level of dereference.
2) 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.
3) 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.

TABLE III
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

V. EXPERIMENTS

A. Experimental Setup

To demonstrate the applicability and usefulness of our scheme, we perform our experiments on an ARMv5TE instruction set architecture [12] modeled using cycle-accurate SimpleScalar simulator [16]. The static analysis algorithm is implemented as a pass during the compilation using the Gnu C Compiler ported for ARM. We use the MiBench suite [15] of embedded applications to demonstrate the effectiveness of our technique. Table III shows the maximum stack depth and SPM size used for different benchmarks.

B. Energy Models

We use the CACTI tool [13] 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 0.33 and 0.13 nJ, respectively. It should be noted that the per access energy increases with SPM size. The external memory energy model is for a 64-MB Samsung K4X51163PC SDRAM [14]. The energy per read burst$(E_{SDR/RD})$ for the SDRAM is 3.3 nJ, whereas a write burst$(E_{SDR/WR})$ is 1.69 nJ. The following equations are used to calculate energy consumed:

$$\begin{aligned} E_{TOTAL} &= E_{SPM-TOTAL} + E_{SDR-TOTAL} \\ E_{SPM-TOTAL} &= (N_{RD} * E_{SPM/RD}) + (N_{WR} * E_{SPM/WR}) \\ E_{SDR-TOTAL} &= (N_{SDR-RD} * E_{SDR/RD}) \\ &\quad + (N_{SDR-WR} * E_{SDR/WR}). \end{aligned}$$

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 the following.

1) **Baseline** System with only $1K$ cache.
2) **SPMM** System with $1K$ cache and an SPM. SPM managed using SPMM described in Section II.
3) **GCCFG** System with $1K$ cache and an SPM. SPMM with call consolidation described in Section III.
4) **SPMM-Pointer** System with $1K$ cache and an SPM. SPM managed using circular stack management and pointer extension described in Section IV.

In the baseline setup, all the accesses go through the cache. In the rest of the setups, only the stack accesses go through the SPM, and rest all go through the $1k$ cache. The SPM sizes are chosen such that they are at least as much as the largest function stack frame in the benchmark. Fig. 8 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 [8] 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 32% energy savings by simply managing the entire stack from SPM seamlessly for recursive and nonrecursive functions without the time-consuming profiling process. In addition, our solution does not require the SPM size until runtime making it binary compatible.

While the *SPMM_check_in* and *SPMM_check_out* functions are relatively large, about 667 lines of assembly code, and 176 lines, respectively, for each invocation, only about 74 and 35 lines are executed. It should be noted that we have not aggressively optimized the SPMM functions for code size or performance. In addition, while the number of SPMM calls may be as small as 6 for *susan_smoothing* and as high as 60 364 for *dijkstra*, only 266 in *dijkstra* and 4 calls in *susan_smoothing* actually cause data transfers of $0xcd80$ and $0x460$ B, respectively.

We further improve upon our results by reducing the SPMM calls using the consolidation algorithm. While this optimization needs to know the SPM size at compile time, 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 SPMM 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 deallocate stack space for groups of functions rather than individual functions. The SPMM function table is accessed from SDRAM whereas only a limited set of manager data objects are kept in SPM. This is done to keep a minimal space overhead in SPM. The overhead of the SPMM is well compensated for by the reduction in total number of SDRAM accesses.

The performance trends shown in Fig. 9 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 [2] achieves a similar performance improvement. However, 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. However, the manager call

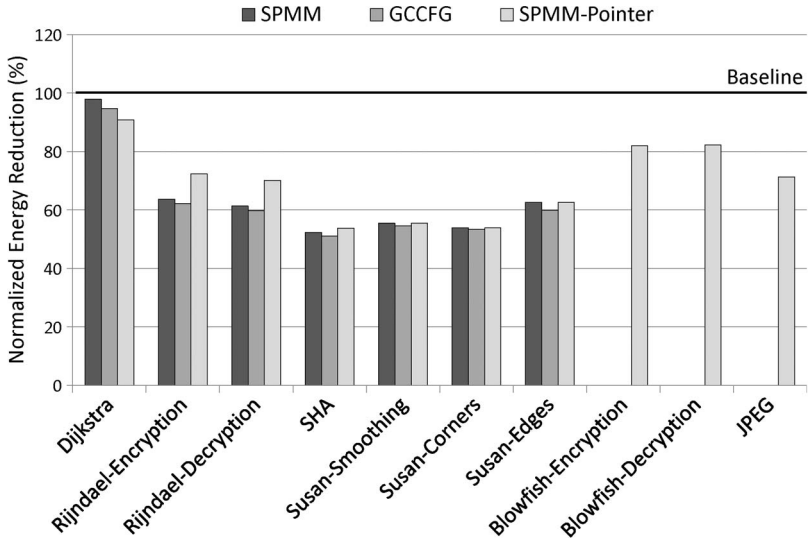


Fig. 8. Normalized energy consumption.

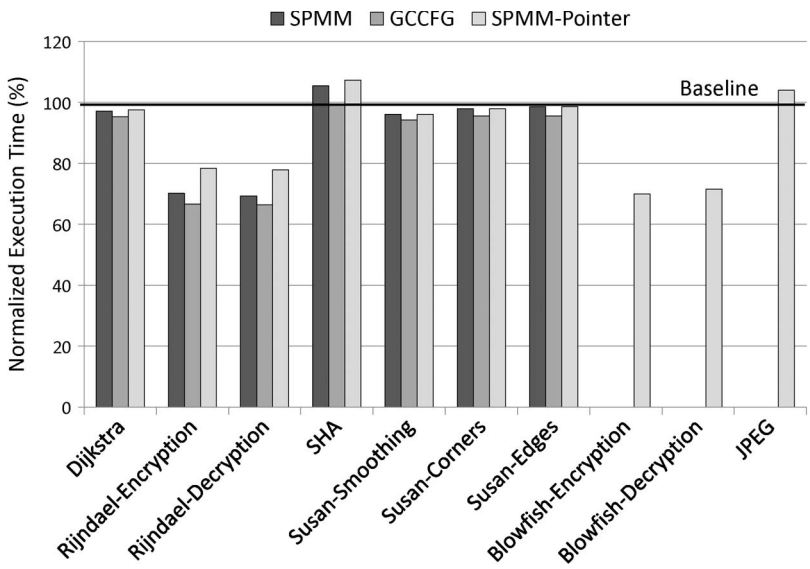


Fig. 9. Normalized run-time.

consolidation algorithm completely eliminates this degradation and results in further average performance improvement of 5%.

While almost all benchmarks use pointers, the first seven benchmarks execute correctly when the SPM size is chosen to be larger than the largest function stack. However, the last three do not. Pointer extension is necessary for all the benchmarks to be sure about the correctness of execution, but is unavoidable for the last three benchmarks. 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.

VI. Conclusion

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. We also demonstrated that if we can know the SPM size at the compile time, the overheads of SPM management can be further reduced by call consolidation. Furthermore, we also proposed pointer extension for applications in which access to stack variables of other functions can happen through pointers. To work seamlessly with interrupts, either interrupts should not use SPM or save and restore the SPM. Same is the case for multithreading. The overhead of save and restore of SPM will be small in

comparison to the software multithreading overhead since we envision the SPM size to be small.

Future work is needed in two main directions: 1) more sophisticated pointer analysis to further reduce programming restrictions and 2) break a function stack into multiple stacks to avoid the minimum SPM size constraint.

ACKNOWLEDGMENT

This work was partially funded by grants from NSF (0916652), Raytheon and Startdust Foundation. The authors would like to thank all the members of the Compiler Microarchitecture Laboratory for their valuable support in this work. J. Lee was the corresponding author for this work.

REFERENCES

[1] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems," in *Proc. 10th Int. Symp. Hardw./Softw. Codesign CODES*, 2002, pp. 73–78.

[2] S. Park, H. Park, and S. Ha, "A novel technique to use scratch-pad memory for stack management," in *Proc. Conf. DATE*, 2007, pp. 1478–1483.

[3] Y. Meng, T. Sherwood, and R. Kastner, "Exploring the limits of leakage power reduction in caches," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 3, pp. 221–246, Sep. 2005.

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

[5] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *Proc. 38th DAC*, 2001, pp. 690–695.

[6] H. Cho, B. Egger, J. Lee, and H. Shin, "Dynamic data scratchpad memory management for a memory subsystem with an MMU," *ACM SIGPLAN Not.*, vol. 42, no. 7, pp. 195–206, Jul. 2007.

[7] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. Min, "A dynamic code placement technique for scratchpad memory using postpass optimization," in *Proc. Int. Conf. CASES*, 2006, pp. 223–233.

[8] A. Dominguez, N. Nguyen, and R. K. Barua, "Recursive function data allocation to scratch-pad memory," in *Proc. Int. Conf. CASES*, 2007, pp. 65–74.

[9] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 1, no. 1, pp. 6–26, Nov. 2002.

[10] M. Verma, S. Steinke, and P. Marwedel, "Data partitioning for maximal scratchpad usage," in *Proc. Conf. ASPDAC*, 2003, pp. 77–83.

[11] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," in *Proc. Int. Conf. CASES*, 2005, pp. 115–125.

[12] ARM. (2008, Jun.). ARM926EJ-S technical reference manual. Cambridge, U.K. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198d/DDI0198_926_TRM.pdf

[13] P. Shivakumar and N. P. Jouppi, *CACTI 3.2*. [Online]. Available: http://www.hpl.hp.com/research/cacti/

[14] *Samsung K4X51163PC Mobile DDR Synchronous DRAM*. [Online]. Available: http://www.samsung.com/products/semiconductor/MobileSDRAM/2005

[15] *MiBench Suite*. [Online]. Available: http://www.eecs.umich.edu/mibench/

[16] T. Austin, *SimpleScalar LLC*. [Online]. Available: http://www.simplescalar.com/

[17] E. J. Johnson and A. R. Kunze, *IXP1200 Programming*. Santa Clara, CA: Intel Press, Feb. 2002.

[18] *The Cell project at IBM Research*. [Online]. Available: http://www.research.ibm.com/cell/

[19] *Intel Core2 Duo*. [Online]. Available: http://www.intel.com/products/processor/core2duo/index.htm

[20] *Intel Penryn 45 nm*. [Online]. Available: http://www.techwarelabs.com/reviews/processors/penryn-preview/

[21] *Intel Tera-Scale 80-Core Processor*. [Online]. Available: http://techresearch.intel.com/articles/Tera-Scale/1421.htm

[22] *IBM PowerPC*. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-powarch/

[23] *Apple iPhone*. [Online]. Available: http://www.apple.com/iphone/

[24] *Sony PlayStation3*. [Online]. Available: http://www.us.playstation.com/PS3

[25] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee, "SDRM: Simultaneous determination of regions and function-to-region Mapping for scratch-pad memories," in *Proc. Int. Conf. HIPC*, 2008, pp. 569–582.

Aviral Shrivastava received the B.S. degree in computer science and engineering from Indian Institute of Technology, Delhi, India, and the M.S. and Ph.D. degrees in computer science and engineering from the University of California, Irvine.

He is an Assistant Professor with the Department of Computer Science and Engineering, Arizona State University, Tempe. His areas of interest lie at the intersection of compilers, computer architecture, and very large scale integration computer-aided design, with a particular focus on compiler, microarchitectural, and compiler–microarchitecture hybrid techniques for improving power, performance, temperature, codesize, reliability, and robustness of embedded and multicore processor systems.

Arun Kannan received the B.S. degree in electronics engineering from the University of Pune, Pune, India, in 2004 and the M.S. degree in computer science from Arizona State University, Tempe, in 2008.

He was working on various firmware development projects with Forbes Marshall and Conexant Systems, Inc. He is currently with the Advanced Visual Computing Group, Intel Corporation, Santa Clara, CA, working on the Larrabee software platform. His research interests are in power-efficient compilation techniques and many-core processor architectures.

Jongeun Lee received the B.S. and M.S. degrees in electrical engineering and the Ph.D. degree in electrical engineering and computer science (EECS) from Seoul National University, Seoul, Korea.

He is an Assistant Professor of EECS with Ulsan National Institute of Science and Technology, Ulsan, Korea. His research interests include configurable/reconfigurable processor architecture design, compilation for energy and reliability, and compilation for multicore processors and graphical processing units.