A Software Solution for Dynamic Stack Management on Scratch Pad Memory [*]

Arun Kannan, Aviral Shrivastava, Amit Pabalkar and Jong-eun Lee
Department of Computer Science and Engineering
Arizona State University, Tempe, AZ 85281, USA
{arun.kannan, aviral.shrivastava, amit.pabalkar, jongeun.lee}@asu.edu

Abstract— In an effort to make processors more power efficient scratch pad memory (SPM) have been proposed instead of caches, which can consume majority of processor power. However, application mapping on SPMs remain a challenge. We propose a dynamic SPM management scheme for program stack data for processor power reduction. As opposed to previous efforts, our solution does not mandate any hardware changes, does not need profile information, and SPM size at compile-time, and seamlessly integrates support for recursive functions. Our technique manages stack frames on SPM using a scratch pad memory manager (SPMM), integrated into the application binary by the compiler. Our experiments on benchmarks from MiBench [15] show average energy savings of 37% along with a performance improvement of 18%.

I. Introduction

Power consumption is a serious concern in all computing systems ranging from embedded systems to large server farms. Battery operated embedded devices need to be extremely power efficient for longer operation times, whereas server farms need to reduce their cooling costs. Caches may consume very significant portion of the processor power. Even the StrongARM 110 consumes about 45% of the processor power [1]. More disturbing is the trend of rapidly increasing leakage power of the cache [3].

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. Unlike the cache, SPMs do not have tag array and comparator logic, and are consequently extremely power efficient. Banakar, Steinke, Lee, Balakrishnan and Marwedel [1] observed that SPMs consume about 40% less power, and occupy 34% less area as compared to cache of similar capacity. The latest multi-core Cell architecture from IBM employs SPM as the working memory for its low-power synergistic processor units. Network Processors like the Intel IXP1200 also rely on SPMs in their micro-engines for power-efficient processing [17].

However, the advantages of using SPM come with 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 to the SPM. Stack data has been identified as one of the most promising candidate to map to the SPM. Stack data enjoys an average of 64.29% of total data accesses for the embedded applications in the Mibench [15] suite.

Mapping data onto SPM is known to be NP-complete. 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, 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 [4, 10, 6, 8]. 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 re-use in SPM using source transformations. This approach, though effective works well only in well structured kernels of code. Work in [2] requires hardware support, which in turn again reduces their 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 paper, we propose a complete software solution, for dynamic management of SPM, and does not require 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 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 before 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%.

[*]This work is partially supported by grants from Microsoft, Raytheon and Stardust Foundation.

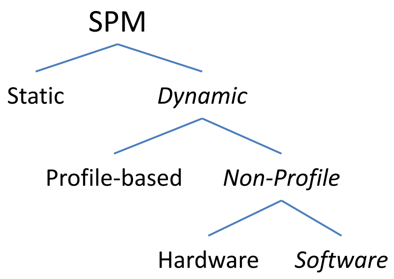


Fig. 1. Classification of Previous Work

II. Related Work

Banakar, Steinke, Lee, Balakrishnan and Marwedel [1] 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. Fig. 1 shows a taxonomy of the SPM mapping techniques.

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 [4, 5, 6, 8] outperform the static techniques in [9, 10].

We can further classify the dynamic methods into profile-based and non-profile methods. Stack management has been studied in few works [4, 8] using profile based techniques. Udayakumaran and Barua [4] 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 as well as a heuristic solution. However, the 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. Thus, there is a need for profile-independent techniques to 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*. Moreover, all the works using profiling information, except [11] 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*. Also, these methods are unable to handle recursive functions and are forced to spill them to the off-chip memory. This can be a deterrent in target architectures similar to the Cell SPE [18], 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.*

Work in [2, 6, 7] perform SPM mapping in systems with hardware support from MMU. Our work is inspired from the approach in [2], 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 [6], 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 Management of Stack

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 toy 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 Fig. 2. Assuming an upward growing (in address) stack, the stack state after the call to F3() is depicted in Fig. 3(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 in the SPM to the SDRAM. This decision 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. Thus, we choose to perform eviction at frame level and keep the management overhead to a minimum. The evicted frames are kept in stack order in a designated SDRAM location. Fig. 3(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.

The management of stack is performed by software SPM Manager (SPMM). The main function of SPMM is to monitor possible overflow and accommodate new frames in the given SPM. It is essential for the SPMM to keep track of the oldest frame at any point in time inside the SPM (shown by *old*

TABLE I
STACK FRAME SIZES FOR SAMPLE PROGRAM

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

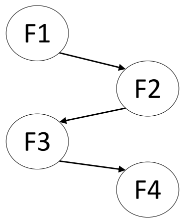


Fig. 2. Sample program call graph

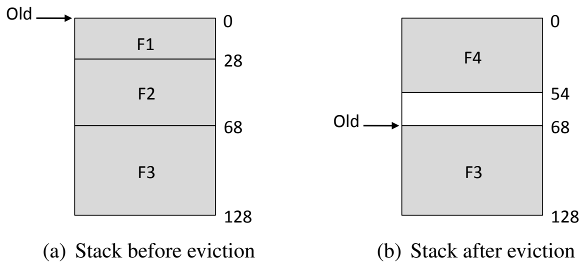
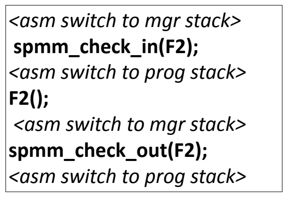


Fig. 3. Stack state for sample program



```
<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. 4. Library calls inserted in application

pointer) to start eviction from that frame. This software library is highly optimized and is linked with the application.

The SPMM uses a function table containing function addresses and their stack frame sizes generated at compile time. The SPMM can obtain the SPM size from the target system at run-time. The application can thus be supported by the SPMM on any SPM size without the need of re-compilation. The SPM manager library calls are inserted by the compiler in pairs, before and after each function call as shown in Fig. 4.

The SPMM call to *spmm_check_in(F2)* 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. 3(b), if F3() simply returns, the stack pointer will point to corrupt data. Thus, a check is made inside the *spmm_check_out(F2)* 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 to switch the stack pointer between the *prog* and *mgr* stack areas between these calls.

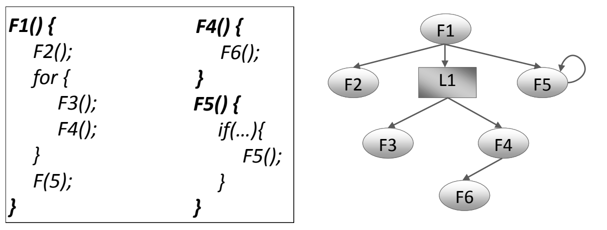
IV. Reduction of SPM Manager Call Overhead

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 Fig. 4. But, there are opportunities to reduce these overheads by examining the call and control flow of the application.

A. Opportunities

We use the sample program shown in Fig. 5(a) to elucidate the optimization. Using the SPMM technique requires a 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. For example, if we consider F4, it has a nested call to F6. Here, it is possible to avoid inserting a separate manager call around F6, 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). 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, we can insert a single call pair around F3 and F4 together, requesting for a stack space of max(F3, F4). 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. We introduce a novel data structure called Global Call Control Flow Graph 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. For conditional statements, it is assumed that both paths will be executed. Recursion can be detected while constructing the graph and is indicated by a self-loop on the F-node. Finally, the CFGs are combined to form a single graph representing both, the call and control flow information of the application. Fig. 5(b) shows a GCCFG for the sample program.



```
F1() {                F4() {
   F2();                 F6();
   for {              }
      F3();           F5() {
      F4();              if(...){
   }                        F5();
   F(5);                 }
}                     }
```

(a) Sample program (b) GCCFG for sample program

Fig. 5. Global Call Control Flow Graph (GCCFG)

B. 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. 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.

In order to perform the manager call consolidation, we explore the GCCFG in a depth-first fashion (done by the routine *Consolidate*). 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. But, the optimization inside a particular function will be performed once, when the graph first explores the function and will remain constant thereafter. Once the GCCFG exploration is complete, the *requestSize* field of each node indicates the action to be taken as shown 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 the emphmain function of the application. The algorithm for the manager call consolidation is shown in Algorithm 1, 2 and 3.

The routine *ComputeStackReq* computes the maximum stack space required by children of a node. This information is used by the routine *Classify* to check if the given size of SPM can hold both, the node and its children's stack in the available

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

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($V_i$)
```

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. 5(a). 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 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.

V. EXPERIMENTS

A. Experimental Setup

We use the cycle-accurate SimpleScalar simulator [16] to model an architecture without a cache, but consisting of an SPM and an external SDRAM memory. The processor uses the ARMV5TE ISA [12]. The static analysis algorithm is implemented as a pass during the compilation. We use the MiBench suite [15] of embedded applications to demonstrate the effectiveness of our technique.

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
```

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 0.33nJ 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 [14]. 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:

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

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

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

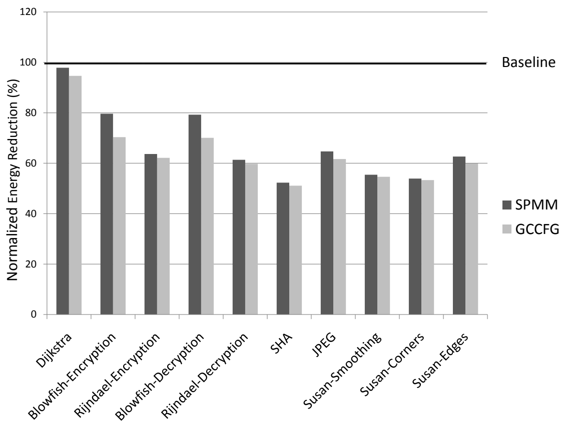


Fig. 6. Normalized energy reduction

VI. 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 (i) system with only SDRAM, 1k cache (Baseline) (ii) system with SPM and circular stack management (SPMM) and (iii) system with optimized circular stack management (GCCFG). The SPM sizes for each benchmark are chosen such that they are at least as much as the largest function stack frame in the benchmark. Since we manage stack data at the frame granularity, our technique is unable to handle stack frames of sizes greater than given SPM size. This limitation will be addressed in the future. Fig. 6 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 non-recursive functions without the time-consuming profiling process. Also, our solution does not require the SPM size knowledge till runtime making it binary compatible.

We improve upon our results further by reducing the SPM manager calls using the consolidation algorithm. We observe a further energy reduction of a maximum 9% for Blowfish. The Blowfish 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 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

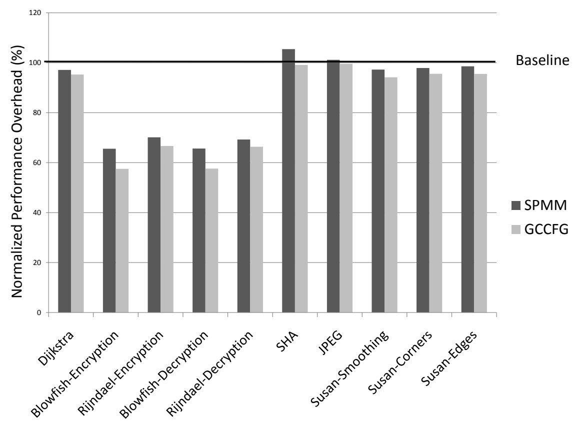


Fig. 7. Normalized performance overhead

the reduction in total number of SDRAM accesses.

The performance trends shown in Figure 7 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. 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%.

Most existing techniques [4, 9, 6] examine the profile information of stack data and formulate an ILP solution to essentially search the huge space of all possible data mappings and schedules to find out the best data mapping and schedule for SPM. This is an extremely complex problem. In contrast, our technique simply manages the SPM to follow the natural access pattern of stack data. Even after losing all the flexibility of data mapping, our solution achieves similar energy reduction.

VII. Conclusion

We proposed a simple, yet effective dynamic circular stack management scheme which does not require system SPM size at compile-time. We also proposed a static analysis method to reduce the software overhead and gained 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.

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. There are many interesting dimensions to extend this method. 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. We will investigate approaches to break the function stack and bring it into the SPM in parts. In addition, stack accesses to data in another function's stack frames (typically using pointers) is an important problem to tackle.

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," *CODES '02: Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78, 2002.

[2] S. Park, H. 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.

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

[4] 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.

[5] 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.

[6] H. Cho, B. Egger, J. Lee and H. Shin, "Dynamic data scratchpad memory management for a memory subsystem with an MMU," *SIGPLAN Not.*, pages 195–206, 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," *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 223–233, 2006.

[8] 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.

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

[10] 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.

[11] 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.

[12] ARM, "ARM926EJ-S Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198d/DDI0198_926_TRM.pdf.

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

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

[15] MiBench Suite, http://www.eecs.umich.edu/mibench/.

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

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

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