SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)

Jing Lu, Ke Bai[*] and Aviral Shrivastava
Compiler Microarchitecture Laboratory
Arizona State University, Tempe, Arizona 85287, USA
{Jing_Lu, Ke.Bai, Aviral.Shrivastava}@asu.edu

ABSTRACT

Software Managed Multicore (SMM) architectures have been proposed as a solution for scaling the memory architecture. In an SMM architecture, there are no caches, and each core has only a local scratchpad memory. If all the code and data of the task to be executed on an SMM core cannot fit on the local memory, then data must be managed explicitly in the program through DMA instructions. While all code and data need to be managed, an efficient technique to manage stack data is of utmost importance since an average of 64% of all accesses may be to stack variables [16]. In this paper, we formulate the problem of stack data management optimization on an SMM core. We then develop both an ILP and a heuristic - SSDM (Smart Stack Data Management) to find out where to insert stack data management calls in the program. Experimental results demonstrate SSDM can reduce the overhead by 13X over the state-of-the-art stack data management technique [10].

Categories and Subject Descriptors

D.3.4 [**Software**]: Processors—*Code generation, Compilers, Optimization*

General Terms

Algorithm, Design, Experimentation, Performance

Keywords

Stack data, local memory, scratchpad memory, SPM, embedded systems, multi-core processor

1. INTRODUCTION

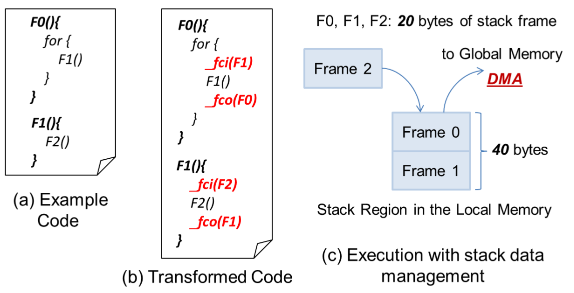
As we scale the number of cores in a processor, scaling the memory hierarchy is a major challenge. Several computer architects believe that completely cache coherent architectures will not scale when there are hundreds and thousands of cores. Recently, Intel manufactured a 48-core non-cache-coherent architecture, called Single-chip Cloud Computer or SCC [3]. However, caches still consume large amounts of power and die area. A promising option for a more power-efficient and scalable memory hierarchy is to have only scratchpad memory (SPM) in the cores. Since scratchpads consume 30% less area and power than a direct mapped cache of the same effective capacity [11], Software Managed Multicore (SMM) architectures can be extremely power-efficient. A very good example of SMM memory architecture is the Cell processor that is used in the Sony Playstation 3. Its power efficiency is around 5 GFlops per watt [14], while the power efficiency of Intel i7 4-core Bloomfield 965 XE is only 0.5 GFlops per watt [1,2].

Software Managed Multicore (SMM) architecture is a truly "distributed memory architecture on-a-chip." Therefore, applications on it require programmers to write several interacting tasks. The tasks are then mapped to the cores of the SMM architecture. Conventionally, *main task* executes on main core and creates *execution tasks*, which are then distributed and executed on execution cores. Main core has a large global or main memory, but execution cores have only a small local memory (the scratchpad memory). The execution cores can directly access only their local memory. To access other memories, including the global memory, explicit DMA instructions are needed in the application. In such architectures, the local memory is shared among code, and all data (stack, global and heap) of the task executing on the core. If the task can fit into the local memory, then extremely power-efficient execution can be achieved – and this is indeed the promise of SMM architectures.

However, for the general case, when all the code and data of the task do not fit in the local memory, explicit data management must be done to enable its execution. The programmer can do this, by bringing in the data/code before they need it, and evicting it back to the global memory after it is no longer needed. This is very difficult, since the programmer must now not only be aware of the local memory available in the architecture, but also be cognizant of the memory requirement of the task at every point in the execution of the program. Estimating the memory requirement is difficult for C/C++ programs, since stack and heap sizes may be variable and input data dependent. This difficulty of programming these SMM architectures has been the biggest roadblock in the success of extremely power-efficient SMM architectures.

To enable execution on the core of an SMM architecture, all code and data must be managed on the local scratchpad. We have started to develop techniques to manage code [18],

[*]This author contributed equally to this work.



```
F0(){
  for {
    F1()
  }
}
F1(){
  F2()
}
```

```
F0(){
  for {
    _fci(F1)
    F1()
    _fco(F0)
  }
}
F1(){
  _fci(F2)
  F2()
  _fco(F1)
}
```

(a) Example Code
(b) Transformed Code
F0, F1, F2: 20 bytes of stack frame
Frame 2
to Global Memory
DMA
Frame 0
Frame 1
40 bytes
Stack Region in the Local Memory
(c) Execution with stack data management

Figure 1: Function-level Stack Management - *(a) an example code, (b) the same code with function stubs* ***_fci*** *and* ***_fco*** *inserted before and after each function call. (c) when the program executes,* ***_fci()*** *may evict existing function frames to the global memory to make space for the incoming function frame, and* ***_fco()*** *may bring back the calling function.*

stack data [10, 29] and heap data ([6, 8, 9] for its form in C, [7] for its form in C++) on the cores with only scratchpad memories. Of these techniques, developing efficient approaches to manage stack data is especially important, since an average of 64% of all accesses in embedded applications may be to stack variables [16].

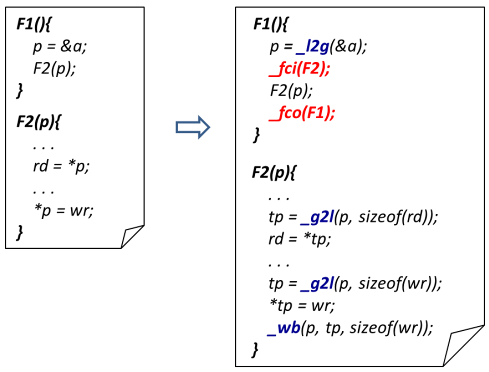
While the state-of-the-art stack data management scheme [10] enables managing stack data of any task on any SPM size (as long as the SPM size is larger than the size of the largest stack frame), there is a lot of room for improving the efficiency of stack data management. The opportunities lie in i) increasing the granularity of management, ii) not performing management when not absolutely needed, iii) performing minimal work each time management is performed, i.e., low instruction overhead of management library. To perform these optimizations, this paper makes two contributions:

- **Problem Formulation:** We formulate the optimization problem of where to insert the management functions so as to minimize the management overhead. We show that the function placement problem can be described as that of finding an optimal cutting of a weighted call graph (WCG). We believe problem definition is very important, and think that lack of formal problem definition is the reason behind high overheads of previous approaches to stack data management.
- **Efficient Heuristic:** Insights from the problem formulation enable us to design an effective heuristic, which we name SSDM. SSDM takes the WCG of the program, and then generates an efficient function placement of data management functions that satisfies the memory constraint on the local memory, while minimizing the management overhead.

Experimental results on several benchmarks from MiBench demonstrate SSDM can reduce the overhead by 13X over the current state-of-the-art stack management technique [10].

2. BACKGROUND AND STATE-OF-THE-ART

Scratchpad memories have been used in embedded systems for a long time, since they may be faster, and lower-power than caches [11]. However, unlike caches (in which the data management is in hardware and software is completely oblivious of it), the data management must be done explicitly in the software in order to use them. As a result, techniques have been developed to manage code [5,13,17,30],



```
F1(){
  p = &a;
  F2(p);
}
F2(p){
  . . .
  rd = *p;
  . . .
  *p = wr;
}
```

```
F1(){
  p = _l2g(&a);
  _fci(F2);
  F2(p);
  _fco(F1);
}
F2(p){
  . . .
  tp = _g2l(p, sizeof(rd));
  rd = *tp;
  . . .
  tp = _g2l(p, sizeof(wr));
  *tp = wr;
  _wb(p, tp, sizeof(wr));
}
```

Figure 2: Pointer Management - *Function F2 accesses the pointer p, which points to a local variable 'a' of function F1. Since 'a' is a local variable on the stack of F1, it has a local address. When F2 is called, if F1 is evicted from the local memory, then the pointer p will point to a wrong value. This is fixed by assigning a global address to the pointer when it is created (through* ***_l2g****), and then when needed, it is accessed through* ***_g2l****. Finally it is written back using* ***_wb****.*

global variables [19, 20, 26, 30], stack data [15, 22, 23, 25, 27, 28, 30] and heap data [12, 24, 28] on scratchpad memories. However, these solutions are not applicable for SMM cores because of the difference in memory hierarchy of SMM cores and the traditional embedded cores. In typical embedded cores, the scratchpad memory is in addition to the regular cache hierarchy. This implies that applications can execute on embedded cores without using the scratchpad. However, frequently needed data can be mapped to the scratchpad memory to improve performance and power. On the other hand, the scratchpad is the only memory in the core of SMM architecture. Therefore everything must be accessed through the scratchpad, the only question is how to perform the management correctly and efficiently.

This paper focuses on stack data management, since an average of 64% of all accesses in embedded applications may be to stack variables [16]. Previous stack data management techniques (both [10, 29]) propose to manage stack data at function level granularity. This is done through code transformations shown in Figure 1. Figure 1 (a) shows an example original code, and (b) shows the transformed code. The *_fci()* and *_fco()* calls are inserted before and after each function call. The function stub *_fci()* makes space for the about-to-be-called function (by removing previous function frames). The function stub *_fco()* brings back the frame of the calling function, in case it was evicted. The execution of the transformed program is depicted in (c), which shows that if the space for stack was 40 bytes, and each function frame was 20 bytes, then when function *F2* is called, there is no more space for it. The *_fci()* will evict the frame of *F0* out of the local memory to make space for the stack frame of *F2*. The *_fco()* at return from function *F1*, will bring the function frame of *F0* back in the local memory.

If a function accesses stack variables of another (ancestor) function through pointers (that may be passed to it as function parameters, or in other data structures), then there may be a problem. The problem, as shown in Figure 2 is that the pointer to a stack variable will be to a local address, since the stack is created in the scratchpad. However, when the pointer to a stack variable of an ancestor func-

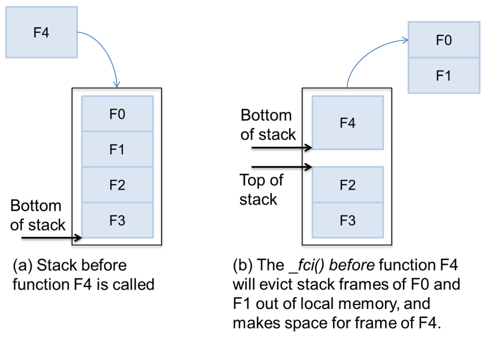

F4
F0
F1
F2
F3
Bottom of stack
(a) Stack before function F4 is called
F0
F1
Bottom of stack
F4
Top of stack
F2
F3
(b) The _fci() before function F4 will evict stack frames of F0 and F1 out of local memory, and makes space for frame of F4.

Figure 3: Circular Stack Management

tion is accessed, that function stack frame may have been evicted by the stack data management. Then the pointer will point to a wrong value. Bai et al. [10] extend the stack management approach to handle pointers correctly. To resolve pointers, they convert the local addresses of the pointers to their global addresses at the time of their definition (through the use of *_l2g* function stub), and at the time of pointer access, the data pointed to is brought into the local memory (through the use of *_g2l* function stub), and after the program is done accessing, it is finally written back to the global memory (through the use of *_wb* function stub). In this paper, we adopted the stack pointer management scheme in [10].

3. MOTIVATION

The state-of-the-art stack data management scheme [10] enables managing stack data of any task on any amount of space on the scratchpad and manages all stack pointers correctly. However, the management overhead is high, and the management is not optimized. The objective of this paper is to optimize stack data management, and reduce its overhead. Optimization opportunities lie in:

Opt1 - Increasing the granularity of management: Not only in SMM architectures, but in all multicore architectures, as the number of cores increases, the memory latency of a task will be very strongly dependent on the number of memory requests. This is because memory pipelines are becoming longer, and a large part of latency is the waiting time to get the chance to access memory. Therefore, it will be better to make small number of large requests, than large number of small memory requests. So the question is: how to increase the granularity of stack data management, even beyond function stack frames.

Opt2 - Not performing management when not absolutely needed: In existing approaches, the function *_fci()* and *_fco()* are inserted before and after each function call. Many times, these functions will not result in any data movement. For example, if there is space for the stack frame of the to-be-called function, then no DMA is required, only some book keeping happens. Much of the overhead is due to calling these functions, even though they are not needed. So, the question is: how do we not insert *_fci()* and *_fco()* functions when not needed.

Opt3 - Performing minimal work each time management is performed: In the existing approach, circular stack management, the older function frames are evicted from the top, and new frames can be instantiated as soon as enough space is available. Figure 3 shows that although this results in a judicious usage of local memory space for stack management, it makes the book-keeping of the space extremely complicated. As different functions may have different stack frame sizes, the stack space will get fragmented after some time. To be able to track the status of stack space, a data structure is required. It needs to reserve stack size of each function, where the frame is stored in the global memory, what the starting address and the end address of the free slots in the scratchpad memory are, etc. In the library, we need to check these variables and update them accordingly, which therefore slows down the application.

Table 1: Library on stack data and stack pointers

Library	Functionality
_sstore()	uses DMA to evict all stack frame(s) from local memory to global memory
_sload()	uses DMA to get all stack frame(s) in previous stack state back to local memory
_g2l(ga,size)	converts global address to a local address; gets the value from global mem. if misses
_l2g(la)	converts local address to a global address
_wb(ga,la,size)	updates data to ancestor frame

4. OVERVIEW OF OUR APPROACH

To optimize the stack data management, we propose to perform stack data management (i.e., transfer stack data between scratchpad and global memory) at the whole stack space granularity. In other words, we keep on instantiating stack frames in the local memory until the management point. At the time of management, the whole stack space is written out to the global memory. When returning from the last frame in the local memory, the whole stack state is copied from the memory to the scratchpad. Since this is no longer at function level, we rename the management functions to *_sstore*, and *_sload*. This approach of performing management at stack space level granularity has several advantages: First is that the granularity of stack data management is much coarser (than function level), and therefore there will be fewer DMA calls (Opt1). Second is that the management library (*_sstore* and *_sload*) becomes simpler, since now the scratchpad is managed as a linear queue, rather than circular queue (Opt3). Table 1 shows our runtime stack management functions and their functionalities.

A problem that can happen in this scheme is that of thrashing. This happens when the stack space is full just before entering a loop with high execution count in which another function is called. Then every time the function is called, the stack state will be written back to the global memory, and reloaded on return. However, this can be avoided by carefully placing the scratchpad functions *_sstore*, and *_sload* in the program. In the next section we formulate the problem of optimal placement of these stack data management functions. We show that the management function placement problem can be described as that of finding an optimal cutting of a weighted call graph (WCG). We formulate an Integer Linear Program solution to the problem (explained in the Appendix, section A), and then propose a heuristic (SSDM) to solve this problem efficiently.

5. PROBLEM FORMULATION

A *weighted call graph* (V, E, W, T) contains a function node set V and a directed edge set E. Each node represents a

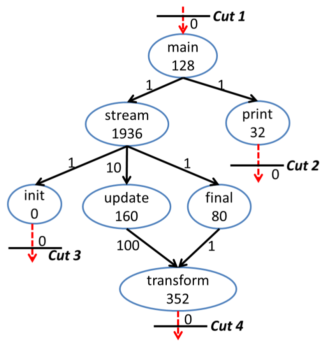


Figure 4: *WCG with cuts of benchmark SHA. The edge with dashed red arrow represents an artificial edge for root node or leaf node.*

function, and each directed edge pointing from the caller to the callee represents the calling relationship between two functions. Weight set $W = \{w_{f_1}, w_{f_2}, ...\}$ represents *stack sizes* of function nodes. Value on each edge e_{ij} ($e_{ij} \in E$) from the value set $T = \{t_1, t_2, ...\}$ corresponds to the number of times function node v_i calls v_j. Figure 4 shows the Weighted Call Graph (WCG) of the benchmark SHA.

A *root node* is the node with no in-coming edges. There is only one root node in the weighted call graph, which is usually the "main" function in a program. A *leaf node* is the node that has no out-going edges. Those are functions that do not call any other functions. However, for the convenience of our problem formulation, we add an *artificial* in-coming edge to the root node with value 0, and an *artificial* out-going edge to the leaf node with value 0. A *root-leaf path* is a sequence of nodes and edges from the root to any leaf node. For example, *main-stream-init* is a *root-leaf path* in Figure 4.

A *cutting of the graph* is defined as a set of cuts on graph edges. A *cut* on an edge e_{ij} ($e_{ij} \in E$) corresponds to a pair of function *_sstore* and *_sload* inserted respectively before and after function v_i calls function v_j. As shown in Figure 4, a set of cuts have been added on *artificial edges* in advance.

We use a list to represent the collection of nodes on a root-leaf path between two cuts. We call such a list of nodes as a *segment*. In Figure 4, the segment between cut 1 and cut 2 is <*main*, *print*>. A node can belong to multiple segments, e.g., node *stream* can be in both segment <*main*, *stream*, *init*> and <*main*, *stream*, *update*, *transform*>. As the total function frame sizes in the local scratchpad memory cannot exceed the size limit of stack space, a positive weight (the size of stack space) constraint $\mathbb{W}$ is imposed on each segment so that the total weight (stack sizes) of functions in a segment will not exceed $\mathbb{W}$. Therefore, given a segment $s = \{f_1, f_2, ...\}$ with function weights $\{w_{f_1}, w_{f_2}, ...\}$, the total weight must satisfy the weight constraint

$$\sum_{f_i \in s} w_{f_i} \leq \mathbb{W} \tag{1}$$

The cost of stack data management for each segment s comprises of two components: i) the running time spent on extra instructions caused by *_sstore* and *_sload* function calls, and ii) the time spent on data movement between the global memory and the local scratchpad memory. Let us assume a segment $s = \{f_1, f_2, ...\}$ is formed with two cuts on edges e_{start} and e_{end}, the functions in this segment have weights $\{w_{f_1}, w_{f_2}, ...\}$, and the two edges have values t_{start} and t_{end} (the number of function calls), the first part of the cost can be represented as

$$cost_1 = t_{end} \times \tau_0 \tag{2}$$

where τ_0 is a constant which represents the average execution time for extra instructions in run-time library (in both *_sstore* and *_sload* function). The time spent on data movement is linearly correlated to the size of DMA, which equals to the total function stack sizes in a segment. As a result, the second cost can be represented as

$$cost_2 = t_{end} \times 2(\tau_{base} + \tau_{slope} \times \sum_{f_i \in s} w_{f_i}) \tag{3}$$

where τ_{base} is the base latency for any DMA transfer, τ_{slope} is the additional latency increasing rate with data size, and 2 shows the consideration for DMA data transfer *in* and *out*.

Therefore, the total cost for each segment s is

$$cost_s = cost_1 + cost_2 \tag{4}$$

For a set of cuts on a Weighted Call Graph (WCG) that forms a set of segments $S = \{s_1, s_2, ...\}$, the total cost can be represented as

$$cost_{WCG} = \sum_{s_i \in S} cost_{s_i} \tag{5}$$

It should be noted that we treat each recursive function as a single segment and always assign a cut to it to ensure a pair of *_sstore* and *_sload* is placed right before and after recursive function calls. The detailed handling could be found in both ILP (Appendix, section A) and SSDM heuristic (Appendix, section B).

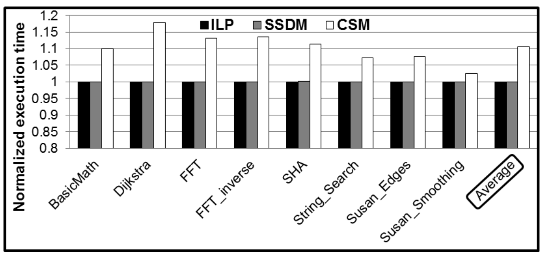
Definition 1. ***(Optimal Cutting of a Weighted Call Graph)*** *An optimal cutting of a weighted call graph G contains a set of cuts that forms a set of segments, where each segment satisfies the weight constraint and the total cost of the segments is minimal.*

6. OUR HEURISTIC: SSDM

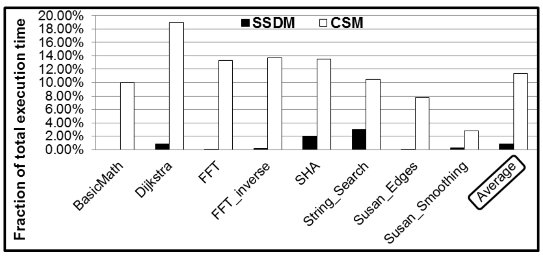
SSDM initially cuts all edges, and then checks all edges to see whether there is a cut on the edge. When a cut is found, our algorithm searches upward and downward through each *root-leaf path* to get its nearest neighboring cuts. Next we form all segments related to this cut by extracting all function nodes between the cut and its neighboring cuts. Thereafter, the total cost of those segments is calculated with Equation 2-5. Now we can assume this cut is removed, and construct new segments by combing upward segment and downward segment in the same *root-leaf path*. If none of these new segments violates the memory constraint of stack space, we can again calculate the new total cost. Otherwise, this cut could not be removed. By subtracting the newer one from the older one, we can get the *removing benefit* of this cut. We can calculate the *removing benefit* of other cuts through the same method. When all calculations are done, SSDM picks the largest one and indeed removes the cut associated with it. It keeps removing the cuts on WCG until no more cuts can be eliminated. The complete algorithm is presented in the Appendix, section B.

7. EXPERIMENTAL RESULTS

In this section we evaluate the efficiency of our SSDM technique by comparing it against the ILP (details are presented in Appendix, section A) and previous CSM heuristic approaches [10]. We have implemented our heuristic in



(a) SSDM against ILP and CSM.



(b) Overhead comparison between SSDM and CSM.

Figure 5: *SSDM reduces the data management overhead and improves performance.*

the GCC 4.1.1 cross compiler for the Cell SPE (Synergistic Processing Element). We consider eight applications from MiBench suite [16]. The other applications in MiBench suite cannot be executed on SPEs because, to some extent, they lack standard library support, or they have large application code size. The eight applications are modified to be multi-threaded by keeping all I/O functionality of the benchmark in the main thread on Power Processor Element (PPE) and the core functionality is executed on the Synergistic Processing Elements (SPEs) [14]. τ_{base} and τ_{slope} used in Equation 3 are 2.1μs and 0.075μs/KB respectively [21]. Table 2 shows detailed information of all benchmarks.

We first utilized PPE and 1 SPE available in the IBM Cell BE and compared our SSDM performance against the results from ILP and CSM [10]. The number of function calls used in Weighted Call Graph (WCG) is estimated from profile information. As observed from Figure 5(a), our SSDM shows very similar performance to ILP approach. This means our heuristic approaches the optimal solution when the benchmark has a small call graph. Compared the CSM scheme, our SSDM demonstrates up to 19% and average 11% performance improvement. The overhead of the management comprises of i) time for data transfer, ii) execution of the instructions in the management library functions. Figure 5(b) compares the execution time overhead of CSM and the proposed SSDM. Results show that when using CSM, an average 11.3% of the execution time was spent on stack data management. With our new approach SSDM, the overhead is reduced to a mere 0.8% – a reduction of 13X. Next we break down the overhead and explain the effect of our techniques on its different components:

Opt1 - Increase in the granularity of management: Due to our stack space level granularity of management, the number of DMA calls have been reduced. Table 3 shows the number of stack data management DMAs executed when we use CSM, vs. the new technique SSDM. Note that there are no DMAs required for *Basicmath.* This is because the whole stack fits into the stack space allowed for this benchmark. Our technique performs well for all benchmarks, except for *Disjkstra.* This is because of the recursive function *print_path* in *Dijkstra.* CSM will perform a DMA only when the stack space is full of recursive function instantiations, while we have to evict recursive functions every time with unused stack space. As a result, our technique does not perform very well on recursive programs. However, since many embedded programs are non-recursive, we have left the problem of optimizing for recursive functions as a future work.

Opt2 - Not performing management when not absolutely needed: Our SSDM scheme reduces the number of library function calls because of our compile-time analysis. In Table 4, we compare the number of *_sstore* and *_sload* function calls when using SSDM, vs. *_fci* and *_fco* calls when using CSM. We can observe that our scheme has much less number of library function calls. The main reason is that our SSDM considers the thrashing effect discussed in Section 4. Our approach tries to avoid (if possible) placing *_sstore* and *_sload* around a function call that executes many times, for example, within a loop. However, CSM always inserts management functions at all function call sites.

Opt3 - Performing minimal work each time management is performed: Our management library is simpler, since we only need to maintain a linear queue, as compared to a circular queue in CSM. Table 5 shows the amount of local memory required by SSDM and CSM, where we can find our runtime library has much less footprint than CSM does. It is very important for improving the performance, since stack frames will obtain less space in the local memory if the library occupies more space. The reason for larger footprint of CSM is that it needs to handle memory fragmentation, while our SSDM doesn't have this circumstance.

Table 6 shows the cost of extra instructions per library function call. We ran all benchmarks with both schemes and approximately calculated the average additional instructions incurred by each library call. As demonstrated in Table 6, our SSDM performs much better than CSM. There is no cost in SSDM when the stack region is sufficient to hold the incoming frames. However, CSM still needs extra instructions, since it checks the status of the stack region at runtime. *hit* for *_g2l* and *_wb* means the accessing stack data is residing in the local memory when the function is called, while *miss* denotes stack data is not in the local memory.

Table 2: Benchmarks, the number of nodes and edges in their WCG, their stack sizes, and the scratchpad space we manage them on.

Benchmark	Nodes	Edges	Stack Size (B)	Scratchpad Size (B)
BasicMath	7	6	400	512
Dijkstra	11	12	1712	1024
FFT	22	21	656	512
FFT_inverse	22	21	656	512
SHA	13	12	2512	2048
String_Search	11	10	992	768
Susan_Edges	8	7	832	768
Susan_Smoothing	7	6	448	256

Table 3: Comparison of number of DMAs

Benchmark	CSM	SSDM
BasicMath	0	0
Dijkstra	108	364
FFT	26	14
FFT_inverse	26	14
SHA	10	4
String_Search	380	342
Susan_Edges	8	2
Susan_Smoothing	12	4

Table 4: Number of *_sstore/_fci* and *_sload/_fco* calls

Benchmark	**_sstore/_fci**		**_sload/_fco**	
	CSM	SSDM	CSM	SSDM
BasicMath	40012	0	40012	0
Dijkstra	60365	202	60365	202
FFT	7190	8	7190	8
FFT_inverse	7190	8	7190	8
SHA	57	2	57	2
String_Search	503	143	503	143
Susan_Edges	776	1	776	1
Susan_Smoothing	112	2	112	2

Table 5: Code size of stack manager (in bytes)

	_sstore/_fci	**_sload/_fco**	**_l2g**	**_g2l**	**_wb**
CSM	2404	1900	96	1024	1112
SSDM	184	176	24	120	80

In CSM approach, more instructions are needed for the *hit* case than the *miss* case in the function *_wb*. It is because the library directly writes back the data to the global memory when *miss*, but looking up the management table is required to translate the address. More importantly, as the table itself occupies space and therefore needs to be managed, CSM may need additional instructions to transfer table entries.

Besides comparing results between SSDM and CSM, we also examined the impact of stack space size and the scalability of our heuristic. We found that i) performance improves as we increase the space for stack data (Appendix, section C), ii) our SSDM scales well with different number of cores (Appendix, section D).

8. SUMMARY AND FUTURE WORK

This paper focuses on managing stack data, since the majority of the accesses in embedded applications may be to stack variables. We formulated the problem of efficiently placing library functions at the call sites. In addition, we proposed a heuristic algorithm called SSDM to generate the efficient function placement. Our experimental results show that SSDM generates function placement which leads to significant performance improvement compared to CSM.

Our optimization works under the assumption that Weighted Call Graph (WCG) could be constructed. However, future work could be devising a scheme to handle function pointers in the construction of WCG. In addition, the number of function calls are profile-based. A static estimation method should be proposed to get those values. Finally, previous scheme for pointers to stack data is directly adopted, but a proper scheme might be developed to further reduce the stack pointer management cost.

9. ACKNOWLEDGMENT

This research was partially funded by grants from National Science Foundation CCF-0916652, IIP-0856090, and NSF I/UCRC for Embedded Systems.

10. REFERENCES

[1] Intel Core i7 Processor Extreme Edition and Intel Core i7 Processor Datasheet, Volume 1. In *White paper*. Intel.

[2] Raw Performance: SiSoftware Sandra 2010 Pro (GFLOPS).

[3] The SCC Programmer's Guide. Technical report.

[4] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[5] F. Angiolini et al. A Post-Compiler Approach to Scratchpad Mapping of Code. In *Proc. CASES*, pages 259–267, 2004.

[6] K. Bai, and A. Shrivastava. A Software-Only Scheme for Managing Heap Data on Limited Local Memory (LLM) Multi-core Processors. *ACM TECS*, 2013.

Table 6: Dynamic instructions per function

	_sstore/_fci		**_sload/_fco**		**_l2g**	**_g2l**		**_wb**	
	F	NF	F	NF		H	M	H	M
CSM	180	100	148	95	24	45	76	60	34
SSDM	46	0	44	0	6	11	30	4	20

* F: stack region is full when function is called; NF: stack region is enough for the incoming function frame; H: hit of stack data; M: miss of stack data.

[7] K. Bai, D. Lu, and A. Shrivastava. Vector Class on Limited Local Memory (LLM) Multi-core Processors. In *Proc. of CASES*, 2011.

[8] K. Bai and A. Shrivastava. Heap Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proc. CODES+ISSS*, 2010.

[9] K. Bai and A. Shrivastava. Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures. In *Proc. of DATE*, 2013.

[10] K. Bai, A. Shrivastava, and S. Kudchadker. Stack Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proc. ASAP*, pages 231–234, 2011.

[11] R. Banakar et al. Scratchpad Memory: Design Alternative for Cache on-chip Memory in Embedded Systems. In *Proc. CODES+ISSS*, pages 73–78, 2002.

[12] 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.

[13] B. Egger et al. A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization. In *Proc. CASES*, pages 223–233, 2006.

[14] B. Flachs et al. The Microarchitecture of the Synergistic Processor for A Cell Processor. *IEEE Solid-state circuits*, 41(1):63–70, 2006.

[15] L. Gauthier and T. Ishihara. Implementation of Stack Data Placement and Run Time Management Using a Scratch-Pad Memory for Energy Consumption Reduction of Embedded Applications. *IEICE*, 94-A(12):2597–2608, 2011.

[16] M. R. Guthaus et al. Mibench: A Free, Commercially Representative Embedded Benchmark Suite. *Proc. Workload Characterization*, pages 3–14, 2001.

[17] A. Janapsatya et al. A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric. In *Proc. ASP-DAC*, pages 612–617, 2006.

[18] S. C. Jung, A. Shrivastava, and K. Bai. Dynamic Code Mapping for Limited Local Memory Systems. In *Proc. ASAP*, pages 13–20, 2010.

[19] M. Kandemir and A. Choudhary. Compiler-directed Scratch pad Memory Hierarchy Design and Management. In *Proc. DAC*, pages 628–633, 2002.

[20] M. Kandemir et al. Dynamic Management of Scratch-pad Memory Space. In *Proc. DAC*, pages 690–695, 2001.

[21] M. Kistler et al. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, May 2006.

[22] L. Li, L. Gao, and J. Xue. Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In *Proc. PACT*, pages 329–338, 2005.

[23] M. Mamidipaka and N. Dutt. On-chip Stack Based Memory Organization for Low Power Embedded Architectures. In *Proc. DATE*, pages 1082–1087, 2003.

[24] R. McIlroy et al. Efficient Dynamic Heap Allocation of Scratch-pad Memory. In *ISMM*, pages 31–40, 2008.

[25] N. Nguyen, A. Dominguez, and R. Barua. Memory Allocation for Embedded Systems with A Compile-time-unknown Scratch-pad Size. In *Proc. CASES*, pages 115–125, 2005.

[26] P. Panda et al. On-chip vs. Off-chip Memory: the Data Partitioning Problem in Embedded Processor-based Systems. In *ACM TODAES*, pages 682–704, 2000.

[27] S. Park et al. A Novel Technique to Use Scratch-pad Memory for Stack Management. In *Proc. DATE*, pages 1478–1483, 2007.

[28] F. Poletti et al. An Integrated Hardware/Software Approach for Run-time Scratchpad Management. In *Proc. DAC*, pages 238–243, 2004.

[29] A. Shrivastava et al. A Software-only Solution to Use Scratch Pads for Stack Data. *IEEE TCAD*, 28(11):1719–1728, 2009.

[30] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions. *ACM TECS*, 5(2):472–511, 2006.

APPENDIX

A. INTEGER LINEAR PROGRAMMING FORMULATION

In this section, we present our Integer Linear Programming (ILP) formulation for placing *_sstore* and *_sload* functions. For a given segment, the cost and total weight can be calculated with Equation 1-5. Given a graph G, all the possible segments can be found out in advance by randomly picking two edges from the graph and putting two cuts on them respectively. Therefore, the optimal *_sstore* and *_sload* placement problem can be transformed as to pick out a set of segments from all the possible segments whose total cost is minimal, and they also satisfy the following two conditions: i) the set of segments can make up the complete weighted call graph G, and ii) each segment satisfies the weight constraint.

The weight constraint can be checked with Equation 1, while checking the first constraint is more complicated. For a graph, we can cut each edge of the graph, and define a smallest segment as an *element*, which contains exactly one node and two edges. In the example shown in Figure 6, the graph is composed of five elements, namely, $<e_0$-F_0-$e_{01}>$, $<e_{01}$-F_1-$e_{13}>$, $<e_{13}$-F_3-$e_3>$, $<e_0$-F_0-$e_{02}>$ and $<e_{02}$-F_2-$e_2>$. Similarly, any segment S in a graph can be represented as a set of elements $S = \{el_1, el_2, ...\}$. In the previous example, the segment formed by the cuts on e_0 and e_{13} contains two elements, which are $<e_0$-F_0-$e_{01}>$ and $<e_{01}$-F_1-$e_{13}>$. For a segment S and a root-leaf path P, if all nodes in elements that belong to S are also contained in P, we say $S \subseteq P$, and we define the segment S as a *subset-segment* of P. For example, in Figure 6, the segment $<F_0, F_1>$ is a *subset-segment* of path F_0-F_1-F_3. Apparently, each segment must be a *subset-segment* to at least one root-leaf path. Now we can check if a set of picked segments can make up the complete weight call graph G. If each element in the path P_i is contained in one and only one subset-segment of P_i, then we can claim that the picked segments can cover path P_i. If the picked segments can cover all paths in G, then we can claim that the picked segments $\mathcal{S}$ can make up the complete graph G.

Eventually, the problem can be presented as follows:
Input:

- $\mathbb{W}$: total weight constraint, it is the size of local scratchpad memory

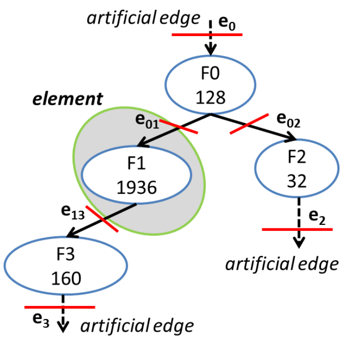


Figure 6: *WCG has many elements, which is composed of 1 node and 2 edges between cuts.*

- E: a set of elements
- S: a set of segments
- P: a set of root-leaf paths
- $cost_s$: cost of each segment s, where $s \in S$
- *weight(s)*: total weight of each segment s, where $s \in S$
- *In(e, s)*: binary value. For any segment s and element e, it is one if $e \in s$, zero if otherwise.
- *subset(s, p)*: binary value. For any segment s and root-leaf path p ($p \in P$), it is one if $s \subseteq p$, zero if otherwise.
- *E(p)*=$\{e_1, e_2, ...\}$: a set of elements such that $e_i \in p$, $p \in P$.

Variable:

$$x_s = \begin{cases} 1 & \text{if segment } s \text{ is picked} \\ 0 & \text{otherwise} \end{cases}$$

Objective Function:

$$\text{minimize} \quad \sum_{s \in S} cost_s \times x_s$$

Constraints:

$$weight(s) \times x_s \leq \mathbb{W}, \ \ for \ s \in S$$

$$\sum_{s \in S} subset(s, p) \times In(e, s) \times x_s = 1, \ \forall \ p \in P, \ and \ \forall \ e \in E(p)$$

The first constraint is the weight constraint, and the second constraint guarantees that the picked segments can make up the complete graph. It should be noted that we must treat each recursive function as a single segment, and add one more constraint for each as follows:

$$x_s = 1, \forall s \ that \ indicates \ a \ recursive \ function$$

It ensures a pair of *_sstore* and *_sload* is placed right before and after recursive function calls.

B. SSDM HEURISTIC

In this section, we present the complete SSDM heuristic for placing *_sstore* and *_sload* library functions. As observed from Algorithm 1, Line 1 preprocesses all recursive edges by placing a cut on them. Since *_sstore* and *_sload* are statically placed at compile time and recursive function calls itself, we must put a cut on the recursive edge to eliminate the nondeterminacy of recursive functions. In line 8-10, we first find out the segments that are associated with each cut x_{ij} on edge e_{ij} ($e_{ij} \in E$). To do this, we need to find out all root-leaf path P_i, where $e_{ij} \in P_i$. Then we search upward through each P_i, until we meet a cut x_{up}. Similarly, we search downward through each root-leaf path P_i, until we meet a cut x_{down}. The segment between x_{ij} and x_{up} or x_{down} is defined as associated with x_{ij}. For example, in Figure 6, the segments that are associated with cut on e_{02} is the segment $<F_0>$ and the segment $<F_2>$. Then we calculate the cost of each segment with Equation 2-5, and the total cost by summing up the cost of all the associated segments. In Line 11-19, we assume the cut is removed, and we can get a new set of associated segments. Those segments are formed by merging the segment between x_{ij} and x_{up} with the segment between x_{ij} and x_{down} on each root-leaf path P_i. As an edge might belong to several root-leaf paths, there might be many x_{up} and x_{down} accordingly. In Figure 6, after removing the cut on e_{02}, the two associated segments are merged into one segment, which is $<F_0$, $F_2>$. Similarly, we can calculate the cost of each new segment

Algorithm 1: SSDM(WCG(V,E))

```
1  Place cuts on recursive edges, if there are recursive
   functions.
2  Define vector $\mathcal{C}$, in which $x_{ij}$ indicates if a cut should be
   placed on edge $e_{ij}$ ($e_{ij} \in E \setminus E_{recursive}$). set all $x_{ij}$=1.
3  while true do
4      Define vector $\mathcal{B}$ to store removing benefit of each cut.
5      foreach $x_{ij} == 1$ do
6          Set boolean violate to false, it shows if removing
           this cut would violate the weight constraint.
7          Define total cost $Cost_{before} = 0$.
8          foreach segment $s\_old_i$ that are associated with
           $x_{ij}$ do
9              Calculate cost $cost\_old_i$ with Equation 2-5.
10             $Cost_{before} += cost\_old_i$
11         Assume the cut of $x_{ij}$ is removed, and get a new
           set of associated segments.
12         Define total cost $Cost_{after} = 0$.
13         foreach new associated segment $s\_new_i$ do
14             Check weight constraint with Equation 1.
15             if weight constraint is violated then
16                 violate = true
17                 break
18             Calculate cost $cost\_new_i$ with Equation 2-5.
19             $Cost_{after} += cost\_new_i$
20         if violate then
21             continue
22         Calculate the benefit of removing the cut as
           $B_{ij} = Cost_{before} - Cost_{after}$.
23         if $B_{ij} > 0$ then
24             Store $B_{ij}$ into vector $\mathcal{B}$.
25     if $\mathcal{B}$ contains no element then
26         break
27     Find out the largest benefit value $B_{max}$ from $\mathcal{B}$, and
       set the corresponding cut $x_{max} = 0$.
28 foreach $x_{ij}$==1 do
29     Place a cut on edge $e_{ij}$, i.e., the compiler places
       _sstore and _sload right before and after the call
       instruction respectively.
```

with Equation 2-5, and the total cost of all associated segments after removing the cut. Line 14-17 check if weight constraint is satisfied by removing this cut. If the constraint is violated, this cut will not be considered to be removed (line 20-21). Line 27 removes the cut with largest positive benefit among all the cuts whose removal will not violate the weight constraint. Line 25-26 is the exit condition of the WHILE loop. The procedure stops until no more cut can be removed from the graph. At this point of time, the rest cuts either have negative removing benefit, or cannot be removed due to weight constraint. The last two lines in the algorithm shows the operations that need to be made in our modified compiler.

C. IMPACT OF STACK SPACE

The experiment for each application in Section 7 was conducted under the scratchpad size specified in Table 2. Next we constructed another set of experiments that evaluates our SSDM technique under tight size constraints. The benchmark *Dijkstra* contains many nested function calls within loop structures, making it a good candidate for showing the impact of different stack region sizes. We expanded the region size from 160 bytes to 416 bytes with the step size of 32 bytes. The resulted performances are demonstrated in Figure 7, where the execution time with different stack region sizes were normalized to the smallest one. The execution time decreases when we increase stack region size. When the size reaches 384 bytes, the performance hardly improves. The primary reason is that we conservatively manage the recursive function by always placing a pair of library function around all its call sites. Therefore, although the region size is large enough, no more benefit can be obtained as only the insertion for recursive function *print_path* is left.

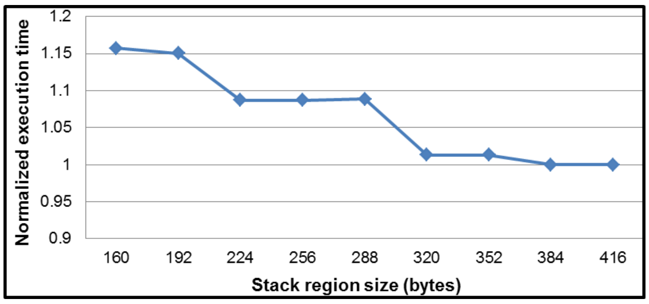


Figure 7: *Performance - different stack region sizes.*

D. SCALABILITY OF SSDM

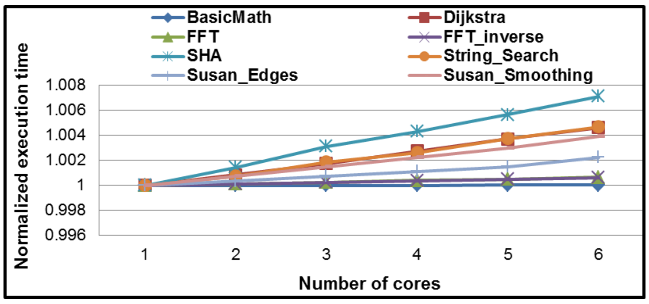


Figure 8: *Performance - different number of cores.*

Figure 8 shows the results we examined the scalability of our SSDM heuristic. We normalized the execution time of each benchmark with number of SPEs to its execution time with only one SPE, and show them on y-axis. In this experiment, we executed the same application on different number of cores. This is very aggressive, since DMA transfers occur almost at the same time when stack frames need to be moved between the global memory and the local memory. This will lead to the competition of DMA requests. As shown in Figure 8, the execution time increases gradually as we scale the number of cores, but no more than 1%. Benchmark *SHA* increases most steeply, as there are many stack pointer accesses in this program. Because of this, more data transfers are conducted for objects pointed by those stack pointers.