Managing Stack Data on Limited Local Memory Multi-core Processors

by

Saleel Kudchadker

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

ARIZONA STATE UNIVERSITY

May 2010

Managing Stack Data on Limited Local Memory Multi-core Processors

by

Saleel Kudchadker

has been approved

April 2010

ABSTRACT

Limited Local Memory (LLM) architectures are popular scalable memory multi-core architectures in which each core has a local software-controlled memory, e.g., the IBM Cell processor. While similar to the scratch pad memories (SPMs) in embedded systems, local memory architecture is different: instead of being in addition to the cached memory hierarchy, local memories are a part of the only memory hierarchy of the core. Consequently, different schemes of managing local memory are needed. The existing circular stack management is a promising approach, but is only for extremely embedded applications, with constraints on the maximum stack size, and limited use of pointers. This research presents a generalized approach to manage the stack data of any application in a constant space on the local memory. The experimental results on benchmarks from MiBench running on the IBM Cell processor in Sony Playstation 3, show gains in programmability, and generalization without much loss of performance.

To

My Family

ACKNOWLEDGMENTS

I would like to take this opportunity to express my sincere gratitude towards every individual who has contributed to this thesis, and helped me both academically and personally during my graduate student years at Arizona State University.

Firstly, I would like to thank my adviser, Dr. Aviral Shrivastava, for his priceless guidance, encouragement, and support throughout my graduate studies making it most exciting endeavor of my master's degree. It was he who motivated me into the field of compiler microarchitecure and GCC and encouraged me to think, solve, and clearly articulate my research problems.

In the same breath, I would also like to thank Dr. Evan Scannapieco and Dr. Sandeep Gupta for their patience and continuous support. Their invaluable comments and insights helped to improve the quality of my thesis. I specially thank Dr. Evan Scannapieco for being on my committee and taking interest in my work at Compiler Microarchitecure Laboratory. I also thank my colleagues at the Compiler Microarchitecture Laboratory, Ke Bai, Seung Jung Chul, Reiley Jayapaul, Fei Hong, Yooseung Kim and Chuan Huang for their interesting perspectives on my research. I am also greatful to JC and Paul for helping me with GCC.

I am very grateful to my parents whose love, support and belief in me has made life's challenges so much more easily surmountable. I am also grateful to my roomates Vithal, Siddhi, Gajanan and Atish for their support throught.

TABLE OF CONTENTS

LIST OF FIGURES

I. Introduction

Processor architectures have rapidly moved to multi-core architectures in the recent years to better deal with the challenges of power consumption, heat dissipation, and limited instruction level parallelism, at a coarser level of thread and core-level. However, as we scale the number of cores, scaling the memory hierarchy is becoming an important challenge. An unified memory architecture, using a traditional cache hierarchy, that is automatically hardware managed is becoming increasingly infeasible. This is not only because caches consume significant power, but also ensuring coherency between caches makes them slow and difficult to scale for large number of cores [7].

Limited Local Memory (LLM) architecture is a promising multi-core platform that has highly scalable memory design. LLM architectures feature a small memory on the processor cores. The core can only access it's local memory, and any transfers between the global memory and local memory have to be explicitly specified in software as direct memory access (DMA) request. In other words, LLM architectures have a software controlled memory in the cores. LLM architectures feature in popular modern and futuristic many-core architectures, e.g., the IBM Cell [11] and the experimental 80-core processor from Intel [18].

LLM processors are programmed in a multi-threaded manner, with explicit Message Passing Interface (MPI) style communication between threads. Each of the threads is then executed on a core. If the entire code and data for a thread can fit into the local memory of the core, then the application will execute extremely efficiently – and this is indeed the promise of LLM architectures. However, if that is not the case, then there are two options: i) the programmer re-partitions and re-parallelizes the application by changing the algorithm so that the code and data can fit in the local memory. This may be formidable, as changing the natural way of parallelization of an application can be counter-intuitive. ii)the other option is to manage all the thread code data on the limited local memory of the core without much change in the application. This chief attraction of the second option is that it keeps programming scalable memory architectures (LLMs) natural and easier.

This goal of this paper is to enable the second option. This requires changing the application to bring the data before it is needed, and may also include evicting the not-so-urgently needed data out of the local memory. Two important questions come up in developing a solution for this. The first is about the granularity of communication, while the second is about the nature and amount of application changes required. These two are opposing needs, and this is what makes the problem challenging. For example, one trivial way of managing application data on the achieving this is by replacing every access to a variable by a load from the global memory, use, and then store to the global memory. Although this scheme is simple and automate-able, it results in several small transactions between the local and the global memory. This is detrimental for performance, but more importantly, is in complete discordance with the trend of multi-core architectures. Looking ahead, while the bandwidth of communication between the local and global memory will increase, the memory latency will only increase with time. What is needed is a scheme that will cause small number of coarse-grain communications between the global and local memory. Also if the scheme is not automatable, it should require a small number of intuitive changes in the application.

While management is needed for all code and data, in this paper we focus only on stack data management. This is because about 64% of memory accesses in embedded applications are to stack data [13], and optimizing them is crucial for performance. Kannan et al. [13] proposed circular stack management scheme to manage the stack data of a thread in a constant space on local memory of an LLM processor. Circular stack management essentially keeps the top few frames in the local memory, and moves the older function frames to the global memory. While effective, this scheme is only applicable for extremely embedded systems in which the maximum stack size of the application may be known at compile time. Further it has limitations on the maximum stack size and only allows a limited use of pointers. Consequently, it is quite restrictive for not-so-embedded applications.

This paper presents a comprehensive scheme to manage stack data of a thread on the local memory of a core of LLM multi-core. The key contributions of this work are:

- Our technique can manage any amount of stack data in a constant space on local memory. Consequently, we can run recursive programs with any stack depth. Further, our technique requires lesser space on the local memory than the previous approach.

- Our scheme resolves all pointer-to-stack accesses. This truly widens the applicability of our technique.

- Only a small set of simple and intuitive changes to the application are needed. The API for our stack management technique consists of only 4 functions, and two of them can be inserted automatically. The other two are extremely simple to use. Finally, since our stack data management scheme is at coarse granularity, it is efficient and scalable.

We apply our stack management technique on benchmarks from MiBench Suite [9] and measure the runtime on Sony Playstation 3. Experiments demonstrate that our technique is simple to implement, allows arbitrary stack depths, and manages all pointers to stack variables without much performance loss.

II. Background

A. *Limited Local Memory Architecture*

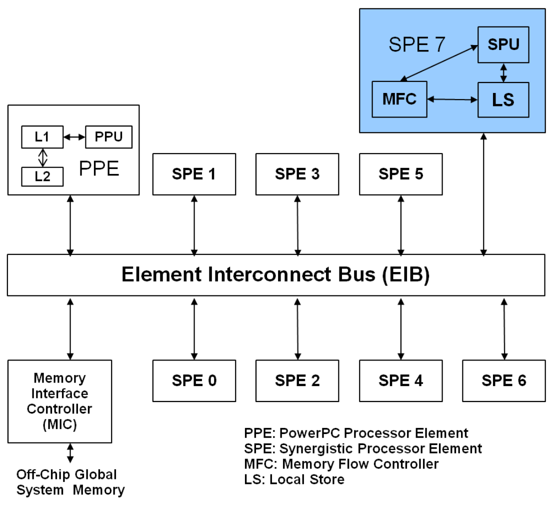
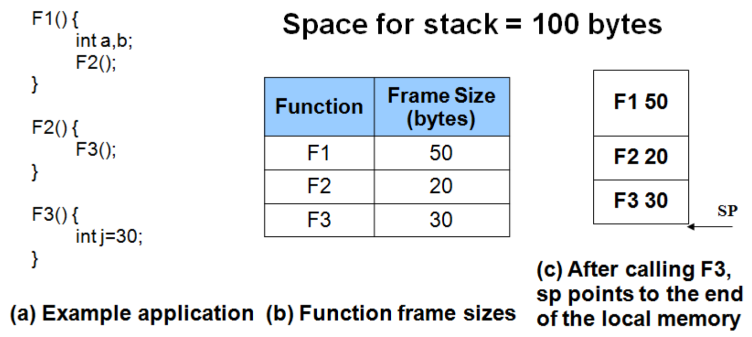

SPE 7
SPU
MFC
LS
L1
PPU
L2
PPE
SPE 1
SPE 3
SPE 5
Element Interconnect Bus (EIB)
Memory Interface Controller (MIC)
SPE 0
SPE 2
SPE 4
SPE 6
Off-Chip Global System Memory
PPE: PowerPC Processor Element
SPE: Synergistic Processor Element
MFC: Memory Flow Controller
LS: Local Store

Fig. 1. The Limited Local Memory architecture of Cell BE with 1 main core (Power Processing Element, or PPE), and 8 execution cores (Synergistic Processing Elements, or SPEs) is shown. The SPEs can only access the local memory, and transfers between the local memory and the global memory have to be explicitly specified in the application.

Limited local Memory (LLM) architecture is a distributed memory platform, in which each cores has access to only a local memory. The Cell processor from IBM [11] is a good example of a LLM multi-core system. As shown in Figure 1, it includes a on-chip, a Power Processing Element (PPE) core, and 8 Synergistic Processing Elements (SPE) cores. The Cell processor has a globally coherent direct memory access (DMA) engine for transferring data between local memories and the global memory. The SPEs can access only a local memory of size 256 KB, and do not have any direct access to global memory. They can only access it through DMA calls, explicitly inserted in the application code. The DMA calls can be initiated by the PPE or the SPEs.



```
F1() {
      int a,b;
      F2();
}

F2() {
      F3();
}

F3() {
      int j=30;
}
```

Function	Frame Size (bytes)
F1	50
F2	20
F3	30

Space for stack = 100 bytes
F1 50
F2 20
F3 30
SP
(a) Example application
(b) Function frame sizes
(c) After calling F3, sp points to the end of the local memory

Fig. 2. If there is 100 bytes of space to manage the stack data of the application, then global memory is not needed. However, if there is less space on the local memory, then some frames need to be evicted out of the local memory to the global memory. This is the problem of stack memory management.

B. *Stack Data Management*

Applications are written for LLM architectures in a multi-threaded paradigm with explicit Message Passing Interface (MPI) style communication between threads. The main thread executes on the main core (PPE in Cell), and spawns threads on the execution cores (SPEs in Cell). The objective of data management in each execution core is to efficiently execute the thread allocated to it, using the local memory present in the core. The space on the local memory is shared by code, stack, global and heap data. While management is needed for all kinds of data, it is especially important for stack data. This is because stack data is dynamic in nature and may not be known at compile time. Without management, stack data can write onto heap data, global data, or even overwrite the code. In the best case, the application will crash, but in the worst, it may just result in incorrect output.

If there is enough space on the local memory to fit the whole stack data, then no stack data management is needed. Figure 2 (a) shows an example of an application thread with three functions *F1*, *F2* and *F3*. Figure 2 (b) shows the sizes of each of the function frames.

If we have 100 bytes of space on the local memory to manage the stack data, then global memory is not needed. However, if there is less space, then some of the older function frames have to be moved to the global memory in order to execute the application – this is the problem of stack data management.

III. Related Work

Local memories in Limited Local Memory (LLM) multi-core processors are raw memories that are completely in software control. They are very similar to the Scratch Pad Memories (SPMs) in embedded systems. Banakar et al. [4] proposed the use of SPMs when he noted that majority of power in the processor was consumed by the cache hierarchy (more than 40% in StrongARM 1110). He demonstrated that this compiler controlled memory could result in performance improvement of 18% with a 34% reduction in die area. SPMs are extensively used in embedded processors, e.g., the ARM architecture [1]. Techniques have been developed to manage code [2,6,10,15–17] , global variables [3,12,14–17], stack [5,13,17] and heap data [8] on SPMs.

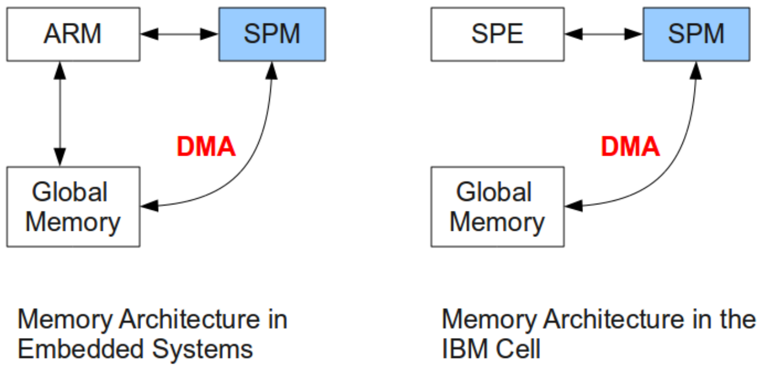

ARM
SPM
DMA
Global Memory
Memory Architecture in Embedded Systems
SPE
SPM
DMA
Global Memory
Memory Architecture in the IBM Cell

Fig. 3. In the ARM architecture, SPM is in addition to the regular memory hierarchy, while in the Cell processor, the local memory is an essential part of the memory hierarchy of the SPE.

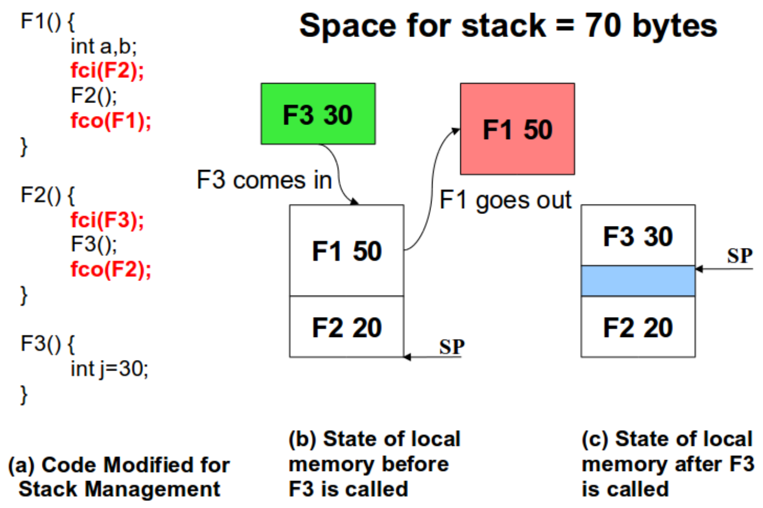
In this paper, we deal with managing stack data. Among the stack data management techniques, [17] works on mapping non-recursive function to SPM and [5] maps recursive functions, but both are profile-based, and are therefore only applicable for extremely embedded systems.

While all these works are related, they are not directly applicable for local memories in LLM architecture. This is because, as Figure 3 illustrates, SPMs are present in embedded processors in addition to the regular cache hierarchy, while the local memory is a part of the only memory hierarchy of the core of a LLM processor. Consequently, while “what to map” is an important question to use SPM in embedded systems, but it is not even an option in local memories in LLM architectures. The circular stack management scheme in [13] works for local memories, and is our main comparison point. We describe the circular stack management scheme in more detail in the next section A, and describe it’s limitations. We outline our approach in Section V, and then experimentally demonstrate the effectiveness of our technique in Section B.

IV. Stack Management

A. *Circular Stack Management*

The circular stack management operates at the level of function frames. The basic technique is to export function frames to the global memory if there is no more space on the local memory. Figure 4 illustrates the functioning of circular stack management. Consider now, that we need to manage the stack data of the application in Figure 2 in only 70 bytes of local memory. The local memory gets full when we call *F2*, there is no more space for stack of *F3*. To make space for *F3* the Circular Stack Management scheme evicts the frame of *F1* to the global memory. When *F3* returns, it frees up its space on the local memory. Finally when the execution returns to *F1*, it's stack frame is brought from the global memory to the local memory.



```
F1() {
        int a,b;
        fci(F2);
        F2();
        fco(F1);
}

F2() {
        fci(F3);
        F3();
        fco(F2);
}

F3() {
        int j=30;
}
```

Space for stack = 70 bytes
F3 30
F3 comes in
F1 50
F1 goes out
F1 50
F2 20
SP
F3 30
SP
F2 20
(a) Code Modified for Stack Management
(b) State of local memory before F3 is called
(c) State of local memory after F3 is called

Fig. 4. Circular Stack Management: The stack space can be managed in a circular fashion. If we have only 70 bytes of space on the local memory to manage stack data, then frame F1 is evicted to global memory to make space for F3. Before the execution returns to F1, it must be brought back to the local memory.

The eviction and fetch of frames is achieved by using stack management functions *fci* and *fco* before and after every function call respectively as shown in the code snippet in

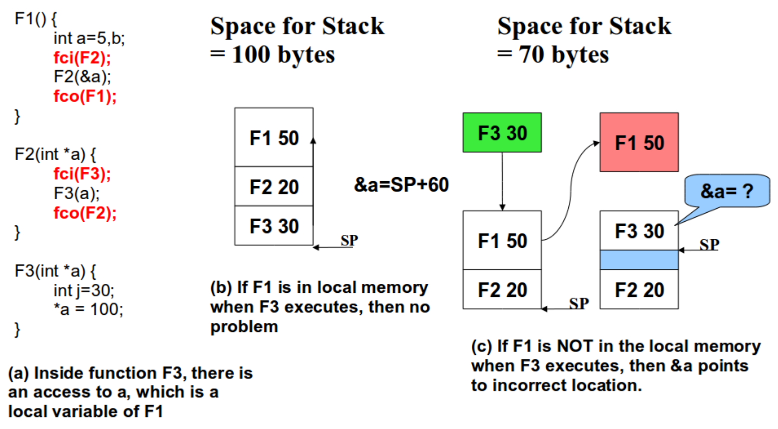

F1() {
int a=5,b;
fci(F2);
F2(&a);
fco(F1);
}
F2(int *a) {
fci(F3);
F3(a);
fco(F2);
}
F3(int *a) {
int j=30;
*a = 100;
}
(a) Inside function F3, there is an access to a, which is a local variable of F1
Space for Stack = 100 bytes
F1 50
F2 20
F3 30
SP
&a=SP+60
(b) If F1 is in local memory when F3 executes, then no problem
Space for Stack = 70 bytes
F3 30
F1 50
F2 20
SP
F1 50
&a= ?
F3 30
SP
F2 20
(c) If F1 is NOT in the local memory when F3 executes, then &a points to incorrect location.

Fig. 5. Pointer Threat: When the frame of F1 is evicted to the global memory and F3 takes its place, the pointer **a** in F3 which refers a local variable in the frame of F1 cannot be referenced as the variable does not exist in local memory. This can cause the program to crash or give wrong results.

Figure 4. *fci* makes an entry into a Stack Management Table. A function table counter keeps track of every function that is checked-in. When the function is checked-out using *fco*, the information entered for that function is discarded and the count is reduced. Next, we describe some limitations of this stack management approach.

B. *Limitations of Circular Stack Management*

B.1. *Pointer Threat*

The Circular Stack Management scheme above works efficiently for applications that do not have pointer references to any previous frames. However, if a function frame has a pointer reference to a variable in the evicted function frame, then there is a problem. Figure 5 illustrates the pointer threat in detail. Figure 5 (a) shows that **a** is a local variable in function *F1*. It is passed to function *F2*, and eventually function *F3*, where it is accessed. Figure 5 (b) shows that if we use 100 bytes in local memory to manage the stack, then when function *F3* executes, *F1* is still in the local memory, and therefore the code will execute correctly. However, if we use only 70 bytes on the local memory to manage the stack, then as Figure 5 (c) illustrates, when *F3* is executing, function *F1* may have been evicted out of the local memory, and therefore the access may be erroneous.

Note that one way to solve the pointer problem is to just increase the size of local store used to manage stack data. Clearly this does not work in general, because the call graph of an application may not be statically determinable, e.g., in the presence of function pointers. Even when this works, it may be difficult to determine the minimum space on the local memory statically due to difficulties in pointer disambiguation. In practice, pointer safe local memory size may be determined by repeated simulation, or by using local memory more than the maximum stack space. To conclude, pointers to stack variables of other functions either make it difficult to run the application, or at the least, increase the minimum space on the local memory that must be used to manage the stack data.

B.2. *Memory Overflow*

There are two aspects of memory overflow in the previous approach. One is the overflow of the stack management table. Previous technique assumes a statically determined size of the stack management table. For large depths and highly recursive applications, the number of entries in the stack management table can exceed the space allocated causing an overflow. The second aspect is the overflow of the memory declared in the global memory to manage the evicted stack data. Existing technique declares a static array in the global memory to manage all the stack data. When we need to evict/fetch a function frame to/from the global memory, we need to know the address in global memory where we will write/read this function frame. This global memory address can be maintained by just a variable in the local memory, which initially points to the start of the array in the global memory, allocated for managing the stack data, and incrementing/decrementing it by the function frame size when the function is evicted/fetched to/from the global memory. However, a statically declared space for managing stack data on the global memory works only for extremely embedded applications where the maximum stack size of the application may be known at compile-time.

V. Our Approach

This section outlines our approach to overcome all the limitations of the circular stack management outlined previously. There are three main aspects of our approach: i) manage the stack data in the global memory, ii) manage the stack management table in a constant space in the local memory, and iii) resolve any stack pointers.

A. *Dynamic Management of Main Memory*

It is clear that for the general case, the stack data in the global memory must be managed dynamically. This implies that at some point of time, the execution core must request the main core to allocate more memory. Since this cannot be done by a DMA call, and some other communication mechanism between the execution core and the main core must be used. In the Cell processor, we use the mailbox facility for this. The main core must be listening to memory requests from the execution core. For this, we implement a new thread on the main core that will continuously listen to requests from the execution core, and allocate more memory when requested. The main core allocates more memory in the global memory, and sends the start and the end addresses of the allocated space to the execution core. This is done so that in most cases, the address translation can be done in the execution core, and only a direct DMA will be needed.

On the execution core, this functionality is implemented in the *fci* function. The *fci* function first checks if there is space for the incoming function stack on the local memory. If not, then it also checks if more memory is needed in the main memory. If not, then the oldest function frames can be evicted to the global memory, otherwise, it sends a request via the mailbox to the main core to allocate more memory. The memory management thread in the main core accepts this request, allocates more memory than the request, and sends the start and end addresses of the newly allocated memory to the execution core,

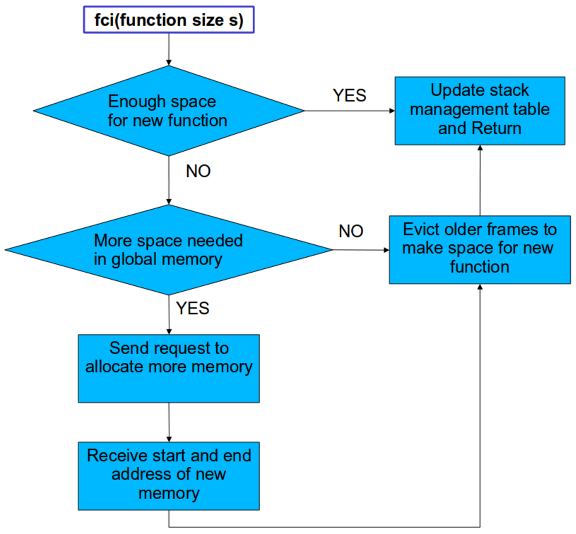

fci(function size s)
Enough space for new function
YES
Update stack management table and Return
NO
More space needed in global memory
NO
Evict older frames to make space for new function
YES
Send request to allocate more memory
Receive start and end address of new memory

Fig. 6. Function check in: *fci* may request for allocation of more memory from the main core. In order to reduce the overhead, both the start and end addresses of the newly allocated memory are send to the execution core. By doing this, the decision of whether more memory needs to be allocated can be made on the execution core itself, and most of the times, just a DMA is enough.

which can then use it for further stack management. The functionality of *fco* is very similar, except that if the all the functions from a memory block have been brought back to the local memory, then the memory block is free-ed.

Instead of adding the global memory management functionality in the existing thread of the main core, keeping this as a separate thread has several advantages. One is that the code of the main thread does not need to be modified, and the extra threads can be supplied as a part of the library, and the user just needs to compile their application with it. Finally, this separate thread solution scales with the number of cores, as just one thread will be able to manage the memory requirements of all the execution threads on the processor. Since the memory allocation is managed by the operating system on the main core, the dynamically allocated buffers never infringe each others space.

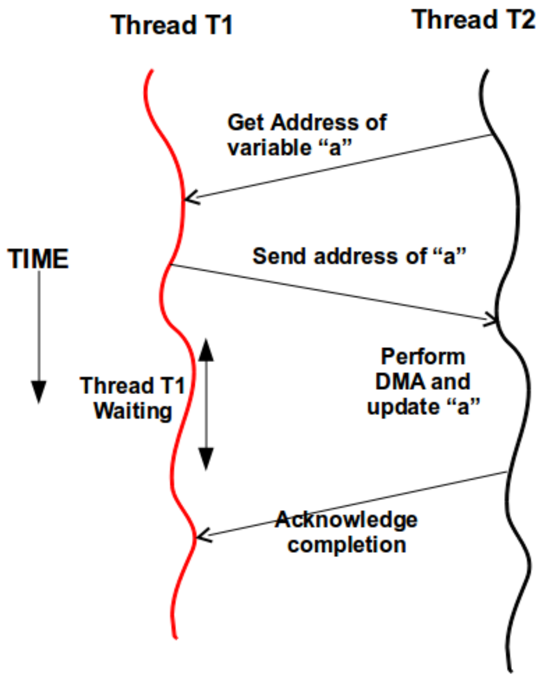
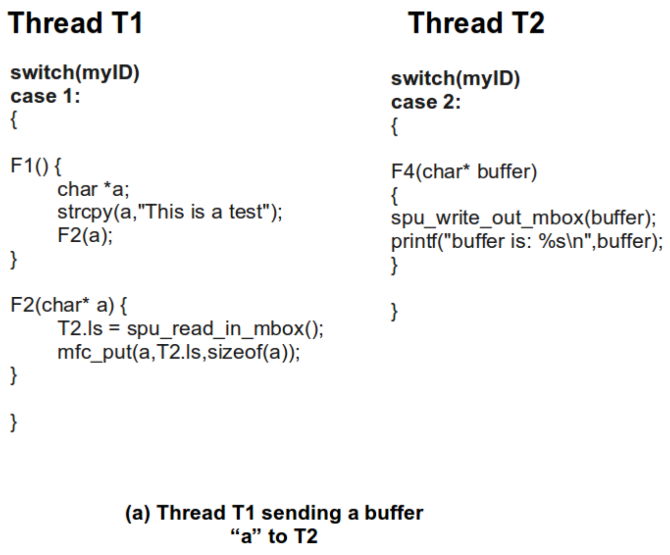


Fig. 7. Thread T2 accesses data from T1 after T1 sends the address of the variable. T1 waits until T2 acknowledges completion of the operation.

For multi-threaded applications, there may be mutual sharing of data among different threads leading to sharing of stack variables. The communication between two different threads running on distributed cores, exclusively takes place using DMA and this is handled by the programmer. Consider a case of a thread *T2* running on one local core accessing a variable of a thread *T1* running on a different local core. To perform a DMA fetch operation ,thread *T2* needs the address of the variable of *T1*. Once the variable is fetched, *T1* waits until the modified variable is written back to its stack to ensure coherency. Once it receives DMA write acknowledgement from *T2* , it continues regular execution. The code snippet of the actual code is shown in Figure 8.

With stack management, the above operation may have a different complexity as the variable accessed by a different thread may not be present in on the local memory. To ensure coherency, we must determine if the function frame that contains the variable is present in the stack. If the variable is local to a differnt function frame, we can perform *fci* operation, to ensure the the variable exists locally on the stack and then pass the address for the regular inter-tread DMA operation. Once the operation completes, *fco* brings back any frames if they were evicted due to *fci*. This is crucial to ensure coherency for any inter-thread DMA. As shown in the Figure 8, the function *F2* of the thread *T1* writes a variable


Thread T1

```
switch(myID)
case 1:
{

F1() {
        char *a;
        strcpy(a,"This is a test");
        F2(a);
}

F2(char* a) {
        T2.ls = spu_read_in_mbox();
        mfc_put(a,T2.ls,sizeof(a));
}

}
```

Thread T2

```
switch(myID)
case 2:
{

F4(char* buffer)
{
spu_write_out_mbox(buffer);
printf("buffer is: %s\n",buffer);
}

}
```

(a) Thread T1 sending a buffer "a" to T2

Fig. 8. Therad T1 is writing into the buffer of T2 in the function F2(). With management, we must ensure function frame of F1() exists in the memory to ensure coherency

a to thread *T2* in the function *F2*. To ensure that *a* is present in the local memory, we must ensure that *F1* is in the local memory *T1* as well as ensure that the desitination *buffer* exits in the local memory of *T2*. This can be accomplished by determining if source frame *F1* and the destination frame *F3* of thread *T2* exits in their respective stacks by using *fci* before the DMA is executed. The addresses can be exchanged by using mail boxes. Thus the coherency may be ensured in circular stack management.

B. *Dynamic Management of Stack Management Table*

Stack management table is extremely crucial for this dynamic management of stack data. The stack management table is needed every time a function frame is evicted/fetched from/to the local memory. It helps in finding out the global address for a local address. In addition, it also keeps track of the space left in the global memory, and if more space needs to be allocated. However stack management table occupies space on the local memory, and enabling unlimited stack depth also requires managing the contents of the stack management table. In other words, some part of the stack management table must be evicted to the global memory to make space for new entries.

Dynamic Management of stack management table is achieved by setting an initial fixed size of the table and monitoring if the table gets filled. Just like the function frames, the stack management table can also be managed in a circular fashion in the functions *fci* and *fco*. In *fci* the table size is checked, if the number of entries equal the fixed size, the entire table is exported and the table entry pointer is reset. In *fco* if the table is empty, one table-full of entries are fetched back to the local memory, and the stack management table pointer is set to the maximum size. Finally more memory in the global memory may be required to manage arbitrary depth of recursion. This can be easily implemented in the same way as we manage the function frames, using the same memory management thread on the main core.

C. *Pointer Resolution*

Pointers to stack variables of other functions either make the program unable to execute with a constant stack space on the local memory, or in the best case, the minimum amount of stack space required on the local memory may be too large. We avoid this problem by resolving the pointer address, and fetching the variable that is pointed to from the global memory if needed. Stack pointer resolution works in three steps. First we determine where the pointer points to in the main memory if the function frame has been exported. Secondly, we bring in the data to the local memory, and perform the pointer operation. Thirdly, we write it back to the global memory at the exact place from where it was fetched. This is achieved by two functions *local_address getVal(local_address a)* and *local_address putVal(global_address ga)*.

The function *getVal* converts the local address of a variable into the address in the global memory by first finding the function frame where the pointer can lie by comparing it with the start and end addresses of the frame. Then it finds the relative displacement from the end of the frame to the pointer address to get the *offset*. The address of the variable in the global memory can then be found by adding the *offset* to the start of the stack data in the global memory. If there are multiple memory blocks that have been allocated, then we must subtract block sizes from the *offset*. The unit of data transfer in the Cell processor is 16 bytes. Therefore, we must find the address of the 16 byte block containing the address, and fetch it from the global memory into a buffer in the local memory. This small buffer in the local memory is managed like a cache. The function returns the new local address of the data, and all pointer operations can now be performed using this new local address. Finally the function *putVal* writes the 16 byte block containing the address back to the global memory.

The process of address translation from local address to global address of the pointer **a** in the example code shown in Figure 5 is illustrated in the Figure 9. Like original, the stack data is maintained top down, with stack growing towards lower addresses. Suppose

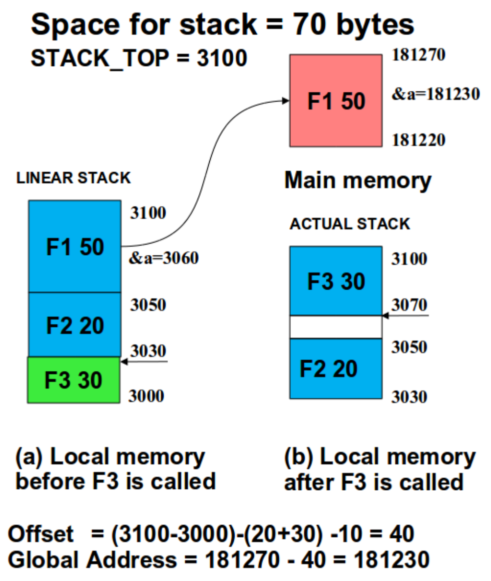
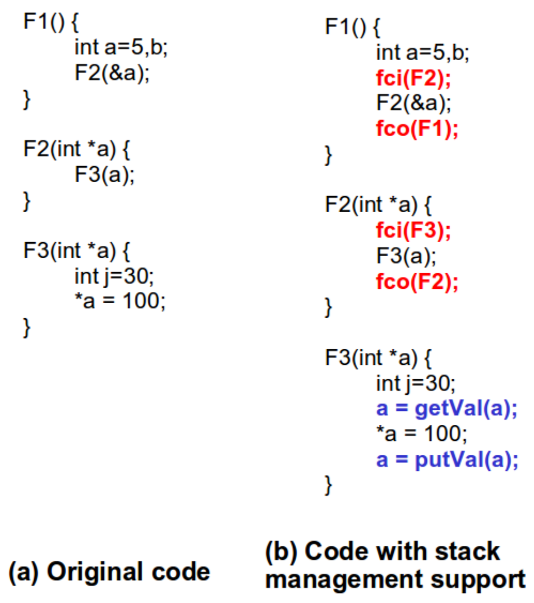

Space for stack = 70 bytes
STACK_TOP = 3100
F1 50
181270
&a=181230
181220
Main memory
LINEAR STACK
F1 50
3100
&a=3060
F2 20
3050
3030
F3 30
3000
ACTUAL STACK
F3 30
3100
3070
3050
F2 20
3030
(a) Local memory before F3 is called
(b) Local memory after F3 is called
Offset = (3100-3000)-(20+30) -10 = 40
Global Address = 181270 - 40 = 181230

Fig. 9. Pointers are resolved in the *getVal()* function. First the offset of the variable is computed using the stack start address in the local memory and the number of times the local stack has been flushed. The offset is used to move relatively in the main memory to reach the pointer location.

the stack top is at address 3100, and the function *F1* is at the top, with the variable **a** is at address 3060 in the stack frame of function *F1*. Assuming the total space on the local memory to manage stack data is 70 bytes, then when function *F3* is called, function *F1* is evicted out to the main memory. Suppose the start address of stack data in the main memory is at 181270, then the global address of variable **a** in the global memory is $181270 - 40 = 181230$. Now function *F3* is called with pointer to **a** as a parameter, it gets 3060 as the address. From this address, the function *getVal* has to generate the global address. This can be simply done by first computing the *offset*, as $3070 - 3100 + 1 * 70 = 40$, where 70 is the stack size, and it is multiplied by the number of times the local stack has been emptied. This *offset* can then be used to compute the global memory address of the variable.



```
F1() {
        int a=5,b;
        F2(&a);
}

F2(int *a) {
        F3(a);
}

F3(int *a) {
        int j=30;
        *a = 100;
}
```

(a) Original code

```
F1() {
        int a=5,b;
        fci(F2);
        F2(&a);
        fco(F1);
}

F2(int *a) {
        fci(F3);
        F3(a);
        fco(F2);
}

F3(int *a) {
        int j=30;
        a = getVal(a);
        *a = 100;
        a = putVal(a);
}
```

(b) Code with stack management support

Fig. 10. Stack Management API: the API consists of just 4 functions, and is very intuitive to use.

D. *Programming API*

The stack management Application Programming Interface (API) is very simple for the programmer's to use, and consists of just 4 functions. Figure 10 illustrates all the changes that are required in a simple application that uses pointers to stack variables of other functions. The functions *fci* and *fco* are added just before and after a function call. The functions *getVal* and *putVal* are added just before and after a stack pointer access.

VI. Experiments

A. *Setup*

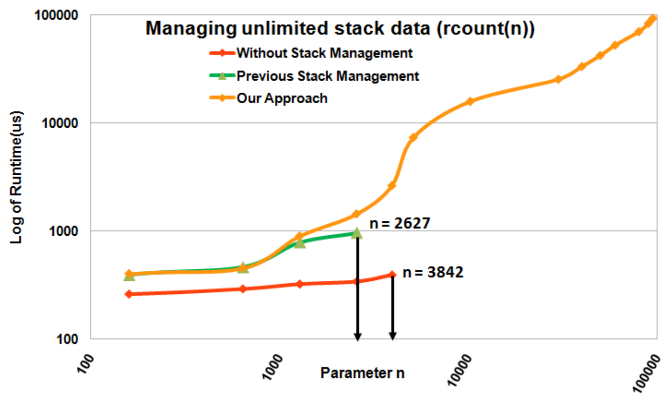
We demonstrate the need and effectiveness of our approach by experiments on Sony Playstation 3. We have installed Linux Fedora 9 on the Sony PS3, which gives access to 6 of the 8 SPEs. We have implemented our approach as a library with the GCC. We compile and run benchmarks from the MiBench suite [9]. These benchmarks are not multi-threaded; we have made them multithreaded by keeping all the input and output functionality of the benchmark in the main thread, which runs on PPE. The core functionality of the benchmark is however executed on the SPE. Thus each benchmark has two threads, one running on the PPE and the other on SPE. In our last experiment on scaling, we run multiple threads of the same functionality on the SPEs. The runtime for the SPEs was recorded using *spu_decrementer()* function and using *_mftb()* for PPE provided as libraries with IBM Cell/BE SDK 3.1. To abstract away the variability in timing due to operating system and other reasons, we run each experiment 60 times, and take the average.

B. *Results and Discussion*

B.1. *Enabling Limitless Stack Depth*

To demonstrate the need of our technique we executed a simple recursive function *rcount* on the PS3, and plot the runtimes in Figure 11. This simple application requires 8784 bytes for the code and 1200 bytes for global data, and the rest 246 KB can be used for stack. The function frame size for this application is 64 bytes, and therefore, without stack management, this application only works for $n < 3842$.

The interesting thing that Figure 11 shows is that when we apply the previous approach of stack management, the recursive function only works for $n < 2627$. This is because as n increases, the space in the local memory used up by the stack management table also



```
int rcount(int n)  {
        if (n==0) return 0;
        return rcount(n-1) + 1;
}
```

Fig. 11. Without stack management, the recursion function only works for $n < 3842$. Our technique enables limitless stack depth.

increases, and therefore less space is left for storing function frames. We would like to make clear that the observation that the previous stack management scheme works for even smaller programs than without any stack management cannot be generalized and is therefore a little misleading. This happens only for cases when the function frame is extremely small, and the stack management overhead is relatively high. In general, previous approach should/does enable executing applications with larger stack sizes and depths.

Most important observation from the Figure 11 is that our technique has no limitation on the stack depth that it can support. As compared to previous technique [13], we are not restricted by the size of the stack management table and the memory allocation in the global memory. When the number of entries in the table exceed the fixed size we mention, it is exported dynamically to the PPE. Also our scheme does not need array size on the PPE to accommodate the stack frames. This is taken care of automatically by the memory management thread in the PPE.

B.2. *Better performance at smaller space*

Another aspect of our technique is that it can manage stack data in a smaller space on the local memory. The minimum space that the previous approach [13] required on the local memory is the sum of the largest function and the size of the stack management table. The stack management table contains one entry for each function instantiation. We manage the stack management table dynamically between the local memory and the global memory, and therefore can work with just one entry in the stack management table. As a result not only we can execute applications with arbitrary stack depths (which we demonstrated in the last experiment), our technique occupies much less space on the local memory. Using less space on the local memory is extremely crucial, since the local memory is typically small, and on top of that, it is shared by global, stack, heap data and the application code. In order to maximize the flexibility of mapping it is vital to map each individual data in as little space as possible.

While the conclusion scales for all the benchmarks, due to limited space, we show graphs for two of the benchmarks, *sha* and *Dijkstra*, both from MiBench in Figure 12. For *sha*, the previous technique does not work until the space allocated for stack data on the local memory is 2976 bytes, while our technique starts working with only 2432 bytes of space. Similarly, our technique can bring down the minimum space on local memory required to perform stack management from 1344 bytes to just 672 bytes. In both cases we see that the runtime for the previous and our technique are comparable for the same stack size. The reason for this is that although our technique is more generic and incurs more management overhead, we can reduce the space in the local memory used for stack management table, and increase the number of function frames that can be kept on the local memory. This results in less DMA between the local memory and the global memory, and also less communication between the PPE thread and the SPE thread.

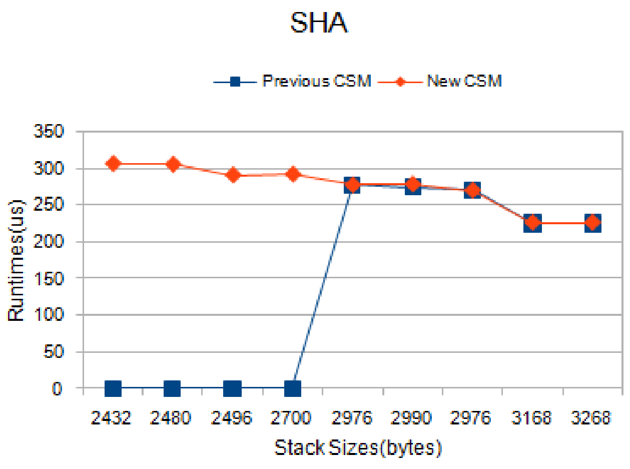
B.3. *Wider Applicability*

Our technique promises to run any application in the least amount of stack on the Limited Local Memory architectures. Table 13 shows a range of benchmarks upon which we have tested our technique. For each benchmark, we find out the size of the largest function, and also find out the maximum stack depth by running these benchmarks once. We then run these benchmarks using space on local memory equal to the size of the largest function plus the maximum size of stack management table. This minimum stack size is shown in the first column of Table 13. The second column in the Table 13 shows the runtime of the application, if it ran, *CRASHES* is printed. It can be noted that several benchmarks like *FFT* crash. Our stack management can work with lesser space on the local memory. The fourth column lists the minimum space on the local memory required by our scheme, and the fifth column lists the time required to execute the application on that minimum space.

There are two main observations here. The first is that our technique successfully resolves the pointer, and therefore works for a wide range of benchmarks. The second observation is that, as compared to the cases when the previous approach also worked, our technique may have slightly more runtime, e.g., the runtime for *dijkstra_small* using previous technique was $882\mu s$, but with our management is $890\mu s$. But this is because we are using lesser space, and managing using only one entry in the stack management table. For a fair comparison, we increase the total space occupied by stack data and stack data management data structures to that of the second column, we achieve at-par performance.

B.4. *Scalability of our technique*

Till now all our experiments were on PPE and one SPE. Our approach adds a memory management thread to the PPE that would service the memory requests from all the cores. To illustrate the scalability of our approach, we execute identical benchmark on every core. We run the benchmarks at the least stack and table size possible. This ensures that we do maximum transfers of the of stack management table and stack frames to the main memory. Figure 14 plots the runtimes of the application (as measured on the PPE), of the benchmarks as the number of cores/threads scale. In the case of FFT, the memory traffic is lesser as there is no recursion and it gives nearly similar runtimes as we scale the number of SPE's. However, for other benchmarks, the runtime gradually increases as we scale the number of cores/threads. This is because, as the number of thread/cores increase, the number of memory requests increase, and they block the SPEs until the previous requests are served. Another important observation is that as we scale the number of cores/threads, at some point (which varies for each application), the runtime suddenly increases. This is because, as the number of threads/cores increase there is a larger memory traffic and this causes the overload of the memory bus.



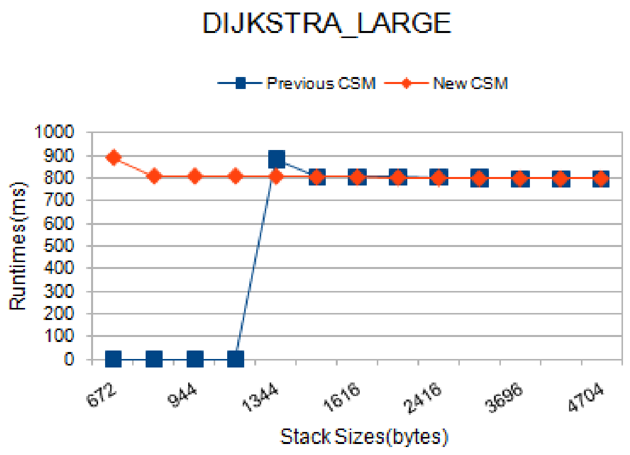
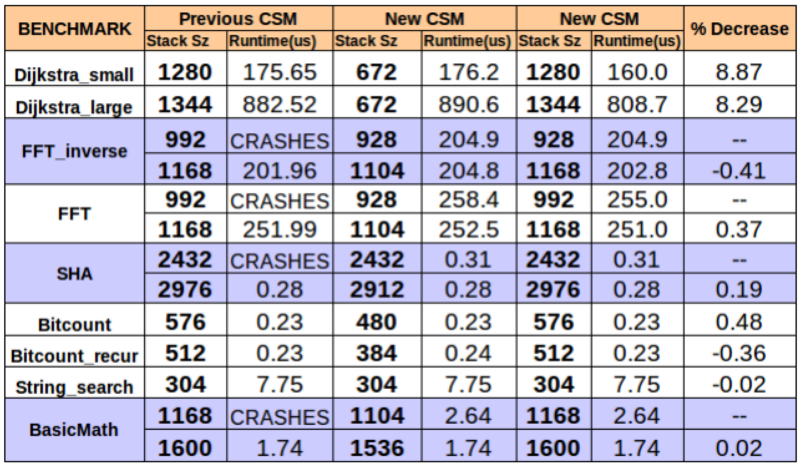


Fig. 12. Two main observations from these graphs: i) The minimum space required on the local memory to run the benchmarks was larger than what our technique needs. ii) When both techniques work, the runtimes are comparable.



BENCHMARK	Previous CSM		New CSM		New CSM		% Decrease
	Stack Sz	Runtime(us)	Stack Sz	Runtime(us)	Stack Sz	Runtime(us)	
Dijkstra_small	1280	175.65	672	176.2	1280	160.0	8.87
Dijkstra_large	1344	882.52	672	890.6	1344	808.7	8.29
FFT_inverse	992	CRASHES	928	204.9	928	204.9	--
	1168	201.96	1104	204.8	1168	202.8	-0.41
FFT	992	CRASHES	928	258.4	992	255.0	--
	1168	251.99	1104	252.5	1168	251.0	0.37
SHA	2432	CRASHES	2432	0.31	2432	0.31	--
	2976	0.28	2912	0.28	2976	0.28	0.19
Bitcount	576	0.23	480	0.23	576	0.23	0.48
Bitcount_recur	512	0.23	384	0.24	512	0.23	-0.36
String_search	304	7.75	304	7.75	304	7.75	-0.02
BasicMath	1168	CRASHES	1104	2.64	1168	2.64	--
	1600	1.74	1536	1.74	1600	1.74	0.02

Fig. 13. Two important observations: i) Our technique can manage stack data in a lesser space on the local memory, and ii) When compared with the previous technique, while using the same total space for stack data and stack management data structure, our technique has at-par performance.

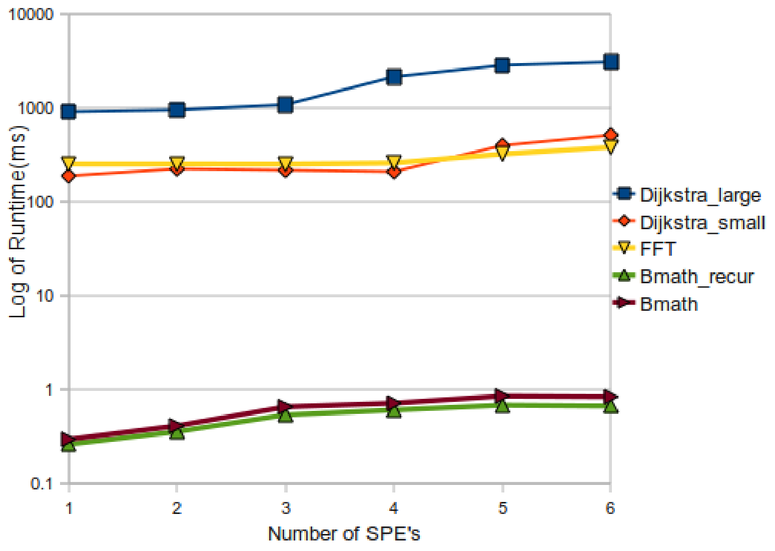


Fig. 14. As we scale the the number of cores, runtime increase as there is larger memory traffic that keep memory management thread in the PPE occupied. The sudden increase is due to the overloading of the memory bus with increasing evictions from each local memory.

VII. Conclusion, Future Work and Outlook

We propose a novel technique for memory management for the stack for limited local memory multi-cores. Our technique scales of any number of cores and ensures stack management for any arbitrary size of application in least possible size of stack. In addition our implementation takes into account the space management on the main memory and allocates eviction-aware memory, thus conserving and benefiting other memory hungry applications running on the main memory. Our stack management technique is not confined to limited local memory multi-core architectures but can also be used in general purpose systems.

Our work foresees further possibilities of research in efficient management of stack. The work can basically be done in three major areas. Firstly, the main memory manager thread function can be made to allocate space intelligently, predicting recursion to reduce the number of allocation requests. Also the total space required for management can partitioned into the stack management table space and frame space to optimize the number of DMA transfers caused due to table and function frame eviction. Secondly work can be done to resolve pointer references at a coarser level. Analysis can be performed to know when a pointer accesses will need the circular to linear translation as not all frames that are referenced are exported to main memory. This can be further strengthened by keeping the pointer references that are frequently accessed in the local memory. Thirdly the stack management scheme can be improved by evicting only a part of a function frame to the main memory to accommodate a new function.

As future architectures are likely to have distributed cores with limited local memory, scaling general purpose applications becomes a challenge. The ultimate quest is to increase the performance/watt. If an application can be run in smaller memory requirement, it eventually means that the memory per core can be reduced freeing up space on the die which can be used for other purposes like embedding more cores per chip which can increase the throughput.

REFERENCES

[1] *"ARMv5 ARM Architecture version 5(ARMv5TE)"*. http://www.arm.com/.

[2] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267, New York, NY, USA, 2004. ACM.

[3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, 2002.

[4] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems". *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, 2002.

[5] A. Dominguez, S. Udayakumaran, and R. Barua. *Embedded Computing*, 1(4):521–540, 2005.

[6] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233, New York, NY, USA, 2006. ACM.

[7] A. Eichenberger et al. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[8] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 238–243, New York, NY, USA, 2004. ACM.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "Mibench: A free, commercially representative embedded benchmark suite". *Proceedings of the Workload Characterization, 2001. WWC-4*, pages 3–14, 2001.

[10] A. Janapsatya, A. Ignjatović, and S. Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 612–617, Piscataway, NJ, USA, 2006. IEEE Press.

[11] C. R. Johns and D. A. Brokenshire. Introduction to the cell broadband engine architecture. *IBM J. Res. Dev.*, 51(5):503–519, 2007.

[12] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 628–633, New York, NY, USA, 2002. ACM.

[13] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee. A software solution for dynamic stack management on scratch pad memory. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 612–617, Piscataway, NJ, USA, 2009. IEEE Press.

[14] L. Li, H. Feng, and J. Xue. Compiler-directed scratchpad memory management via graph coloring. *ACM Trans. Archit. Code Optim.*, 6(3):1–17, 2009.

[15] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 115–125, New York, NY, USA, 2005. ACM.

[16] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. Sdrm: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Int'l Conference on High Performance Computing (HiPC)*, December,2008.

[17] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.

[18] S. Vangal et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. In *IEEE Journal of Solid State Circuits*, pages 29–41. IEEE Press, 2008.