Stack Data Management for Limited Local Memory (LLM) Multi-core Processors

Ke Bai, Aviral Shrivastava and Saleel Kudchadker
Compiler Microarchitecture Lab
{Ke.Bai,Aviral.Shrivastava,Saleel.Kudchadker}@asu.edu

***Abstract*—Limited Local Memory (LLM) architectures are power-efficient, scalable memory multi-core architectures, in which cores have a scratch-pad like local memory that is software controlled. Any data transfers between the main memory and the local memory must be explicitly present as Direct Memory Access (DMA) commands in the application. Stack data management of the cores is an important problem in LLM architecture, and our previous work outlined a promising scheme for that [1]. In this paper, we improve the previous approach, and now can i) manage limitless stack data, ii) increase the applicability of stack management, and iii) perform stack management with smaller footprint on the local memory. We demonstrate these by executing benchmarks from the MiBench suite on the IBM Cell processor.**

***Index Terms*—Stack, local memory, scratch pad memory, embedded systems, multi-core processor, IBM Cell, MPI**

I. Introduction

As we scale from a few-core processor to many core processor, scaling the memory architecture becomes challenging. Limited Local Memory (LLM) multi-core architectures are scalable, distributed memory architectures. In an LLM processor each core has only a local memory without caches, therefore is quite power-efficient. Any data transfers between the main memory and the local memory must be explicitly present as DMA commands in the application. The popular IBM Cell processor [2] is a good example of LLM architecture. LLM architectures are programmed in MPI-like multi-tasking paradigm with explicit communication between the tasks. The tasks are mapped to the cores, which means the local memory in the core is shared by code, stack, global and heap data of the thread. If the thread can fit into the local memory, then the application will execute extremely power-efficiently. Otherwise all code and data of a thread must be managed in the local memory.

Local memories in LLM multi-core processors are very similar to the Scratch Pad Memories (SPMs) in embedded systems. SPM is extensively studied and techniques have been developed to manage code [3], [4], [5], global variables [6], [7], [5], [8], stack [8], [9] and heap data [10]. While all these works are related, they are not directly applicable for local memories in LLM architecture. This is because of the difference of the memory architecture of SPMs in embedded systems and LLM architecture: applications can execute on embedded processors without using the SPM. Frequently needed data can be mapped to the SPM to improve performance and power. On the other hand, local memory is the only memory hierarchy of the core of a LLM processor. **Therefore using local memory in LLM cores is not an optimization, but is a necessity.**

Previously, we have proposed schemes for managing code [11], [12], heap [13] and stack data [1] in LLM architecture. This paper improves our previous stack management technique — Circular Stack Management (CSM) [1]. CSM essentially keeps the top few frames in the local memory, and moves the older ones to the main memory. While effective and efficient we identify three problems in the previous approach. We fix them in this paper. The limitations of previous work, and contributions of this work are as follows:

- **Support unlimited stack:** CSM requires that the total amount of stack space required by the task must be known at compile-time. As a result, it does not support arbitrary depth of recursion. *We improve this by proposing an interface by which a core can request for dynamic memory allocation in the main memory.*
- **Finite and small footprint in the local memory:** Stack Management requires a management table, which contains information about where data is, in the local memory or main memory. CSM assumed that this table can be maintained in the local memory. However, the table itself can exceed the local memory and *we manage it between the local and the main memory.*
- **Stack Pointer Resolution:** There can be a problem when there is a reference to a local variable of a previous function. Data management may have moved that stack frame that contains the variable to be accessed to the main memory, and therefore looking it up using the local memory address is impossible. *We solve this problem, by always using main memory addresses for pointers to local variables.*

After our enhancements, the task that executes on a core can use unlimited stack space, work with smaller local memory footprint, and allow access to local variables of other functions through pointers. We demonstrate these by executing benchmarks from the MiBench suite, executing on the IBM Cell processor in Sony PS3.

II. Circular Stack Management

The Circular Stack Management (CSM) [1] scheme operates at the level of function frames. The basic technique is to export function frames to the main memory if there is no more space on the local memory and bring them back when needed.

The eviction and fetch of frames is achieved by using API functions *fci* and *fco*, that need to be inserted just before and after every function call. Function *fci(f)* guarantees enough space to accommodate the stack frame of f. If not, it evicts as many oldest functions as required to make enough space. Similarly *fco()* makes sure the frame of caller is in the local

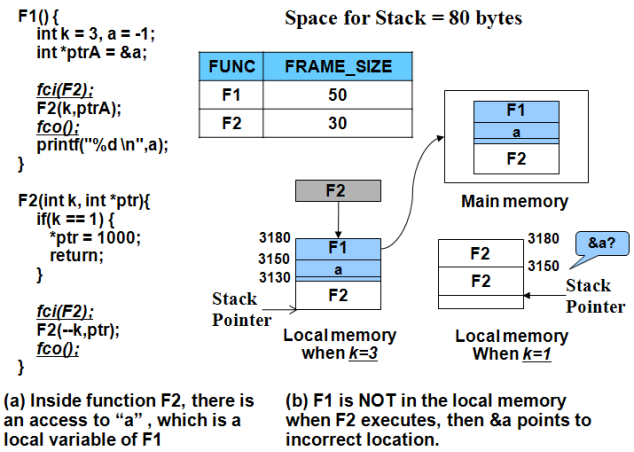

Fig. 1. **Pointer Threat:** *When the frame of F1 is evicted to the main memory and F2 comes in, the pointer* **ptr** *in F2 which refers the local variable* **a** *in the frame of F1 cannot be referenced.*

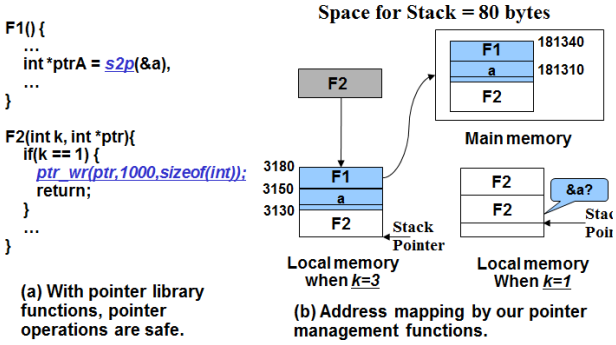

Fig. 2. *Pointers in Fig. 1 resolved by library functions.*

memory. If not, then it is brought from the main memory. All this management requires a table, which contains the information about whether a function frame has been moved to the main memory, and their main memory address. Next section describes some key limitations of the existing CSM approach, and our approach to improve them.

III. Enhancement of Circular Management

A. Pointer Threat and Resolution

CSM works efficiently for applications that do not have pointer references to any previous evicted frames. However, if so, then there is a problem. Fig. 1 illustrates the pointer threat. Fig. 1 (a) shows that the pointer `ptrA` pointing to the local variable `a` is passed as a parameter to recursive function *F2*. The total stack space required for this application will be $50 + 30 \times 3 = 140$ bytes. 80 bytes of stack space requires stack management. When *F1* is called, its function frame is created in the stack, with a location for `a`. Suppose the frame of function *F1* starts at address `0x3180`, and space is allocated for `a` at `0x3150`. Then after the assignment, `ptrA` contains the value `0x3150`. Now all goes fine until the first call to *F2*. At this point, the function frames of both functions *F1* and *F2* are in the stack. Now when *F2* (with $k = 3$) calls another instance of *F2* (with $k = 2$), the CSM function *fci* will remove *F1* out of the local memory, and relocate it to the main memory. When the execution calls the third instance of *F2* ($k = 1$), it falls into the base case to update the value of `ptrA`. The assignment instruction will update the contents of local memory address `0x3150` to 1000. This is clearly wrong, since the variable `a` of function *F1* is actually in main memory, and not in the local memory.

The challenge here is that, the kind of code illustrated in Fig. 1 (a) is all too common, and this pointer problem will show up in any data management solution, not specific to CSM. Resolving the pointer addresses is not trivial. If the variable pointed by pointer is relocated to the main memory, to find its global address becomes a challenge, since we are trying to find a global address from a local address, but the relation from local address to global address is a one-to-many relation. As a result, whenever a pointer is set, it must be set to a global address, rather than a local address. Fig. 2 illustrates the mechanism of our pointer resolution. Fig. 2 (a) shows two changes in the program that was shown in Fig. 1 (a). The first change is that the initialization of the pointer $ptrA$ has been changed to $s2p(\&a)$. The function *s2p* converts the local address of a variable into the global address by first finding which function stack frame the pointer belongs to (e.g. $F1$). Then it computes the *offset* of the pointer variable (e.g. $\&a$) as the relative displacement from the start address of the frame ($F1$) in the local memory to the local pointer address. Finally, it returns a global address, which can be calculated by first getting the global start address of this function frame ($F1$) that is stored by *fci* function before the $F1$ is called and then subtracting the displacement. Fig. 2 shows that the stack top is at the local address `0x3180`, which is stored in the management table. When *ptrA* is initialized, it will get the global address of the variable by the help of *s2p* function. This is done by firstly computing the local address `0x3150` for a. Then the *offset* is computed as `0x3180 - 0x3150 = 0x30`. The start global address of the function frame of $F1$ is looked up from the table, and is `0x181340`. Using these, we can compute the global address of the variable a as `0x181310`. The second change is when there is a write to $ptrA$ in function $F2$. The write to $ptrA$ has been replaced with ptr_wr function. If there were a pointer read, then we would need to use the ptr_rd function. The functions ptr_rd and ptr_wr work directly with the main memory using DMA calls. $ptr\_wr(ga, val, sizeof(val))$ just writes the val in the global address specified in ga, by a DMA call. In this example, ptr_wr can modify a to the value 1000 directly in the main memory. $ptr\_rd(ga, size)$ would instead just bring the value from ga to the local memory in a small buffer, and returns its address. When some other pointer is read, this buffer will be overwritten. By performing this direct main memory transaction, we do not create any data coherency problems.

B. Memory Overflow and Its Resolution

There are two aspects of memory overflow in the previous approach. One is the overflow of the memory space in the

main memory, and the second is the overflow of the Stack Management Table or SMT in the local memory.

CSM should allocate a large enough space at the start of the program in the main memory to accommodate all the stack data of the execution cores. But it is impossible in general due to recursive functions. For recursive functions, the stack space required may be unbounded. And when the main memory allocation fills up, any further DMAs can write into the address space of other execution cores, causing an access fault in the best case, and wrong results in the worst. This implies that at some time, the execution core must request the main core to allocate more memory. Since this cannot be done by a DMA call, and therefore mailbox facility in the Cell processor is utilized. In addition, we implement a new thread on the main core that will continuously listen to requests from the execution cores, and allocate memory when requested. Then it sends the start addresses of the allocated space to the execution core. On the execution core, this functionality is implemented in the *fci* function. Before eviction, *fci* checks whether space is enough in the main memory. If not, it sends a request via the mailbox to the main core. The memory management thread on the main core accepts this request, allocates more memory (e.g. two times) than the request, and finally sends the start and end address of the newly allocated memory to the execution core, which can then be used for further stack management. The functionality of *fco* is very similar, except that if all the functions from a memory region have been brought back to the local memory, then the memory is free-ed.

The other memory overflow problem in the CSM is that of the overflow of the SMT. Every time *fci* is called, it creates a new entry in SMT. When a function returns, its entry can be deleted from SMT. CSM maintains SMT on the local memory. For unbounded recursion, this table can grow arbitrarily large, and any amount of space on local memory will not be sufficient. Just like stack frames, the SMT itself should be managed. In other words, some part of the SMT must be evicted to the main memory to make space for new entries. Dynamic management of SMT is achieved by setting an initial fixed size of the SMT and monitoring if it gets filled. When *fci* adds a new entry in the SMT for the coming function and the SMT is full, the entire SMT is exported and its entry pointer is reset to the start entry of the SMT. When *fco* accesses the already empty table, one table-full entries are fetched back to the local memory, and the table pointer is set to the end entry of the table. Note that after this scheme of dynamic management of SMT, all management is done in constant-sized space. The memory requirements in main memory however are still dynamic, and is managed through the use of the memory management thread on the main core – just like data management of function stack frames.

IV. Experiments

We demonstrate the need and effectiveness of our approach by experiments on the Sony Playstation 3. We implemented our approach as a library with the GCC 4.3.2. We compile and run benchmarks from the MiBench suite [14]. These benchmarks are modified to be multi-threaded by keeping all I/O functionality of the benchmark in the main thread on PPE and the core functionality is executed on the SPE.

Enabling Limitless Stack Depth: To demonstrate the need of our technique we executed a simple recursive function *rcount*, and plot the runtimes in Fig. 3. This simple application requires 8480 bytes for the code and 496 bytes for global data, and the rest 246 KB can be used for stack. The function stack frame size of this application is 32 bytes, and therefore, without stack management, this application only works for $n < 7872$.

Fig. 3 shows the recursive function only works for $n < 29440$ with previous stack management. This is because we set stack region size as 16 KB for the previous technique. Therefore, the rest 230 KB can be used for stack management table. However, as n increases, the space in the local memory used by the stack management table also increases, and therefore all the rest space is used up for storing management table entries. There is a leap when $n > 512$ for two CSMs. This happens because the 16 KB for stack data have been filled up and the eviction is needed for new stack data.

Most important observation from the Fig. 3 is that our technique has no limitation on the stack depth that it can support. As compared to previous technique [1], we are not restricted by the size of the stack management table and the memory allocation in the main memory. When the number of entries in the table exceed the fixed size we mentioned, 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.

Increase in Applicability: Our technique promises to run any application in the least amount of stack on the Limited Local Memory architectures. Given a benchmark, we find out the size of the largest stack frame, and also find out the maximum stack depth by profiling. We then run these benchmarks using space on local memory equal to the size of the largest function frame plus the maximum size of stack management table.

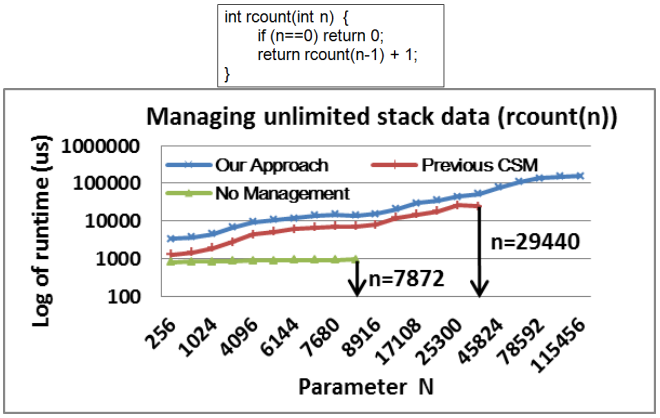

Fig. 3. *Without stack management, the recursive function* rcount *only works for* $n < 7872$*. The previous CSM only support* $n < 29440$*. However, our technique enables limitless stack depth.*

Benchmark	Previous CSM				New CSM	
	Stack Sz(bytes)	Runtime(us)	Stack Sz(bytes)	Runtime(us)	Stack Sz(bytes)	Runtime(us)
BasicMath	168	CRASHES	218	1575747.1	168	1582032.8
SHA	1944	CRASHES	2024	1083.7	1944	1104.3

TABLE I

(1) BasicMath *and* SHA *can not run with the minimum stack region size without our pointer library functions. (2) These two benchmarks can run with a larger stack size after many fails of simulations. (3) Our technique resolves the pointer problem of CSM.*

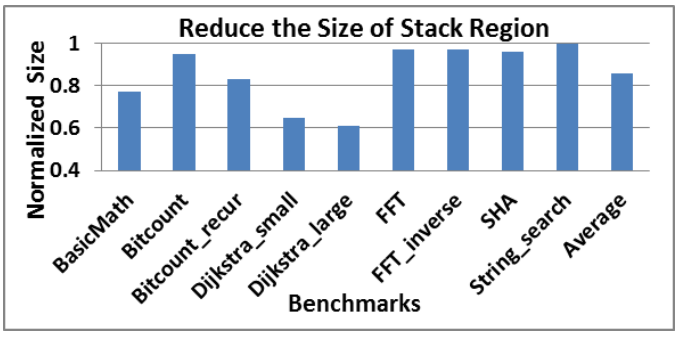

Fig. 4. *(1) Normalization of minimum stack region size means* $\frac{size\_of\_our\_approach}{size\_of\_previous\_CSM}$*; (2) Average space saving by our technique is* 14.3% *and the improvement will be more significant if the application has a larger function call depth.*

This minimum stack size is shown in the second column of Table I. We also show the runtime of the application, if it fails, *CRASHES* is printed. It can be noted that benchmarks *BasicMath* and *SHA* crash. Our stack management can work with less space on the local memory. The sixth column lists the minimum space on the local memory required by our scheme, and the seventh column lists the time required to execute the application with this size. The main observation is that our technique successfully resolves the pointer problem, and therefore works for a wide range of benchmarks.

Stack Management in Smaller Space: Our technique can manage stack data in a smaller space on the local memory. The minimum space that the previous approach [1] requires on the local memory is the sum of the largest function stack size and the size of the stack management table (SMT). The SMT contains one entry for each function instantiation. We manage the SMT dynamically between the local memory and the main memory, therefore work with just one entry. As a result our technique occupies much less space on the local memory. On average our approach uses 14.3% less space on the local memory. We believe that the difference will be more significant if the application has a larger function call depth. Using less space on the local memory is extremely crucial, since the local memory is typically small and shared by global, stack, heap data and the application code. In order to maximize the flexibility of mapping, it is vital to be able to map each individual data in as little space as possible.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 612–617.

[2] C. R. Johns and D. A. Brokenshire, "Introduction to the cell broadband engine architecture," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 503–519, 2007.

[3] 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."

[4] 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*. NY, USA: ACM, 2004, pp. 259–267.

[5] 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*. NY, USA: ACM, 2005, pp. 115–125.

[6] 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*. New York, NY, USA: ACM, 2002, pp. 628–633.

[7] L. Li, H. Feng, and J. Xue, "Compiler-directed scratchpad memory management via graph coloring," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 3, pp. 1–17, 2009.

[8] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic Allocation for Scratch-pad Memory using Compile-time Decisions," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 472–511, 2006.

[9] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *Embedded Computing*, vol. 1, no. 4, pp. 521–540, 2005.

[10] 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*. New York, NY, USA: ACM, 2004, pp. 238–243.

[11] 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.

[12] S. C. Jung, A. Shrivastava, and K. Bai, "Dynamic Code Mapping for Limited Local Memory Systems," in *ASAP '10: Proceedings of the 2010 International Conference on Application-specific Systems, Architectures and Processors*, 2010, pp. 13–20.

[13] K. Bai and A. Shrivastava, "Heap Data Management for Limited Local Memory (LLM) Multi-core Processors," in *CODES+ISSS 2010: Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[14] 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*, pp. 3–14.