Local Memory Store (LMStr): The Case for Hardware Controlled Scratchpad Memory for General Purpose Processors

Nafiul Alam Siddique*, Jeanine Cook†, Abdel-Hameed A. Badawy*, and David Resnick†
*Klipsch School of Electrical and Computer Engineering, New Mexico State University
†Sandia National Labs

***Abstract*—Processors have long employed hardware controlled caches to hide memory latency and decrease data movement. Yet, caches are not a panacea. But still they plays a vital role in processor performance, the generalized use of cache for various program variables introduces sometimes unnecessary data movement. Moreover, it requires to store and match data addresses to access the caches, that consumes significant amounts of power and area. Another possibility to achieve the goals of a cache is to use a software controlled scratchpad memory (SPM), can be used to store and access the programmer/compiler specified data. However, the selection and placement of data in scratchpad memory is bound to compiler optimization(s) for a specific type of variables or a specific program behavior. In this paper, we present a hardware controlled on-chip memory called Local Memory Store (LMStr) that can be used either solely as a scratchpad or as a combination of scratchpad and cache, storing any variable specified by the programmer or extracted by the compiler. LMStr is different than a traditional scratchpad in that it is hardware-controlled and it stores same type of variables in a block that is allocated based on availability and demand. In this work on LMStr, we focus on identify the potential for it. Namely, the advantages of storing temporary and program variables in blocks in LMStr and comparing the performance against a regular cache. To the best of our knowledge, this is the first work where scratchpad has been used in a generalized way and focused on storing temporary and programmer specified variables in blocks. We evaluate LMStr on a microbenchmark and on benchmarks from the mantevo suite. We simulate LMStr in the Structural Simulation Toolkit(SST) simulator. LMStr provides a 10% reduction in average data movement between on-chip and off-chip memory compared to a traditional cache hierarchy.**

I. Introduction

Power constraints on current microprocessor architectures are becoming at least as important as performance. Several studies have shown that a data movement constitute a large percentage of total energy consumption within the memory hierarchy. One such study reported that high performance server processors consume $28 - 40\%$ of their total energy consumption in moving data [1]. Typically, a cache occupies about 50% of the chip area, uses 70% of the total transistors on-chip and consumes about $25 - 45\%$ of the total chip power [2].

Typically, the cache organization is transparent to the programmer which simplifies writing code and generalizes the data placement. However, a significant amount of unnecessary data movement occurs due to conventional caching and its generalized use with different type of data variables (*i.e.* static, local, temporary, and global variables are all treated with the same brush).

Temporary variables are normally generated by the compiler to store intermediate values of operations. These variables cause a significant amount of unnecessary data movement. These variables are accessed only once or twice in a basic block in a context and again accessed in the next basic block with a different context.

Although, the data in these types of variables are rarely accessed in the same context, when a variable spills out of a register, the cache is bound to store it. As a consequence, these variables are updated to main memory as they are evicted from the cache since mostly temporary variable are written to. These variables pollute the cache and stress the interconnection bandwidth which is essentially a waste of a significant amount of energy [3]. Data movement is further exacerbated when cache lines are replaced or evicted. The hardware controls the replacement policies and often times a cache line is evicted and soon after, it gets re-fetched back again.

To reduce data movement and consequently power consumption, processor and memory architectures are evolving to include near-memory or in-memory computing, multi-level memories, lower-power memory technologies, and scratchpad memories. A lot of recent work has examined the use of scratchpad memories (SPM) as an alternative to caches for on-chip storage. Scratchpad memories are small on-chip static RAMs that are directly mapped onto the address space of the processor at a predefined address range of a process [4], and, therefore, requires compiler intervention to be utilized. Scratchpad memories are considerably more energy and area efficient than caches, as no address matching is required to identify stored data *i.e.* no tag bits to store or match.

The key to efficient use of scratchpad is data placement and balancing the responsibility of this task between the hardware and the software. However, though initially scratchpad memories were one of the design choices to store the temporaries to reduce data movement, currently the usage of scratchpad has moved towards program variables. One of the shortcomings of scratchpad memory is its efficiency, which relies on the compiler optimization for a specific scratchpad memory orga-

nization, where specific variables or specific program functions can be mapped. Therefore, the usage of scratchpad memories has been limited to embedded system where applications with specific characteristics can be executed.

This work proposes a memory hierarchy called Local Memory Store (LMStr) that implements an on-chip scratchpad memory, replacing the conventional cache hierarchy, backed by a traditional DRAM for high-performance computing systems.

The LMStr architecture is a high bandwidth local memory intended to generalize the usage of scratchpad memory to meet programmer requirements with the compiler guided mapping technique. However, LMStr can be extended to manage storage requirements for multiple multithreaded processes running in multicore system by its hardware controlled allocation technique that outside the scope of this paper.

The key challenges in realizing an efficient scratchpad memory implementation such as our LMStr in scientific (rather than embedded) computing are to identify the most critical data variables and manage them in the scratchpad space.

A lot of work has been done in the area of compiler optimization to identify appropriate types of variables such as global, array, stack, or scalar variables to store in the scratchpad [5]–[10]. However, all these methods rely on profiling information that essentially identifies the most frequently used data/variables or the most cache conflicting data/variables or use a cost function to find the most viable data to store in SPM. On the other side, compiler guided dynamic scratchpad allocation techniques have been proposed where data reuse patterns, array live range, average power consumption, critical loops, or critical program point(s) have been measured to store data in the scratchpad [10]–[13]. Among these works, Avissar *et al.* [5] proposed to store most frequently used stack variables with a profile guided static allocation technique. The stack contains temporary variables, this procedure may store few frequently used temporary variables in the scratchpad memory. As we propose to store temporary variables as well as programmer suggested other variables according to the programmer's suggestions, our research is clearly different than these works.

Moreover, these already proposed techniques are specifically aimed for a variable or for a characteristic of the application determined by profiling or by complex cost function(s). On the contrary, programmer's suggestions can be an easy way to decide variables to store in the scratchpad. Our hardware controlled data placement strategy is also a generalized technique for any type of variables to store in the scratchpad. We also propose a technique to generate data blocks by the compiler according to the type of variables and build a block generator to evaluate the performance that combines memory addresses into memory blocks based on temporal and spatial locality.

The contributions of this paper are as follows:

- Propose a compiler guided hardware controlled on-chip memory hierarchy, LMStr, that is a scratchpad memory or is a combination of scratchpad and cache, that can be a replacement for conventional cache memories.
- Allowing the programmers to explicitly specify the program variables or types of variables to store in the scratchpad memory in LMStr memory. Therefore, instead of keeping the on-chip memory organization transparent to the programmers, we provide the structure of the on-chip memory to both the programmer and the compiler to map the blocks of the program variables in the scratchpad.
- Justifying the use of th scratchpad memory to store the temporary variables as well as other types of variables.
- Quantitative performance comparison (including data movement analysis) of traditional cache/memory hierarchy to LMStr implemented as all scratchpad and as part scratchpad and part cache.
- Development of a tool that combines memory references into variable-size references using temporal and spatial locality characteristics of the original reference stream.

II. LMStr Architecture

At a top level, the LMStr architecture is a combination of software (*i.e.* compiler) and hardware (*i.e.* memory system) design. In this section, we discuss the architecture of the LMStr memory model and the overheads to be incurred due to necessary extensions to the architecture.

A. LMStr Memory Model

The organization of LMStr memory comprises of four components: an access control unit (*LMEng*), a mapping table (*LMRef*), a directory (*LMDir*), and a shared data store (*LMStr*) as shown in Figure 1.

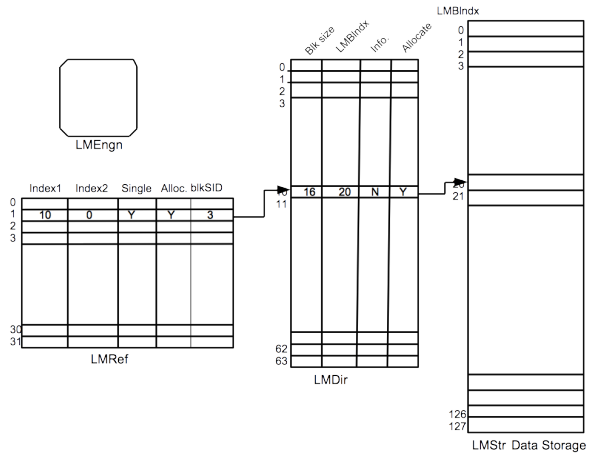

Fig. 1: LMStr Architecture

The LMEngn acts as a controller for the local memory store. There are four primary actions handled by the *LMEngn* that are described below:

- Allocate a block in the LMStr:
  - Search for a free space in the LMStr with size greater than or equal to the size of the new block. If found, allocate the block in LMStr and update the amount

of free space in LMStr. If the space was not found, then the LMStr is bypassed.
  - Allocate an entry in the LMDir and store the *LMBlndx* in that entry. The *LMBlndx* is the index into the starting address of the space in LMStr, allocated to a particular data block.
  - Store the LMDir entry, that comprises of the Blk Size (Block size requested) and the LMBIndx, into the corresponding LMRef entry.
- De-allocate a block from the LMStr:
  - De-allocate the LMDir entry.
  - De-allocate the LMStr storage.
  - Updates available free space in the LMStr.
- Bypass LMStr:
  - Data requested will bypass the LMStr; data is fetched from memory and sent directly to the CPU.
  - When writing, update main memory.
- Defragments LMStr space.

The local memory reference (LMRef) table keeps information pertaining to the process data that is currently stored in the scratchpad. When a process is spawned, the OS loads the LMRef mapping (generated by the compiler) periodically from the main memory, and LMEngn tries to store the data into storage.

The LMRef table holds the indexing information into the LMDir (through which the LMStr is accessed) and flags. Therefore, every memory request must first access the LMRef to be satisfied. However, a single LMRef entry is able to index to two LMDir entries (index1 and index2). Among them, the second LMDir entry (index2) is used while multiple blocks of data are currently stored in LMStr, mapped with same LMRef entry. The first LMDir entry (index1) is used to store the information of the currently used data block.

The LMDir is used to find blocks in the LMStr data storage. It stores the size of the requested block (Blk size), its starting address (*LMBlndx*) in LMStr and a serial number of the currently stored block mapped to same LMRef entry (blkSID). Because LMStr blocks can vary in size, the LMDir must implement a block size field.

The LMStr data storage is where the process data is stored and accessed in a manner similar to a fully-associative cache without the tag checks.

B. LMStr Overhead

The overhead of LMStr model originates from hardware modification(s). The hardware of LMStr architecture consists of LMRef tables, an LMDir table and a LMStr storage space. All the elements of LMStr are proposed to build with fast 6T SRAM as cache to achieve high speed data access.

III. LMStr Methodology

The LMStr model is a compiler driven hardware controlled methodology primarily designed for temporary and programmer/compiler suggested variables that requires runtime hardware support to store data blocks.

The basic idea of LMStr model is that the programmers are responsible to flag the individual program variables or types of the variables (static, local, global, temporary, and arrays) to store in scratchpad memory. Then, the compiler generates blocks by combining similar type of flagged variables and each block is mapped to an entry of a table (LMRef entry) with a placement algorithm. Next, at the time of program execution, the block is physically indexed with the LMRef entry which index to an entry in a table (LMDir). The table index to a shared data storage where the data is actually stored. Therefore, essentially the compiler is required to optimize and identify the underlying LMStr hardware and programmer/compiler suggested variables. The hardware support for LMStr includes a mapping table with multiple entries (LMRef table) and a predefined, tentative amount of space for each process.

A. Proposed Compiler Optimization for LMRef Management

In LMStr memory model, the compiler is extended to generate data blocks by combining similar type of program variables and map them to the entries of LMRef table. Typically, according to the usage and position in program code, the compiler classifies the program variables into six distinct types as local, global, static, array, temporary, and heap variables. The compiler maps the variables in groups in virtual address space in a predefined mapping template.

The compiler would be extended to track each of the suggested memory items and their data flow graph in the program code. The compiler extension needs to cooperate with the LMStr model in three ways. Firstly, extend the code generation phase of the compiler to generate blocks with similar type of variables in procedure/function or basic blocks. Secondly, extend the phase to map the blocks in LMRef entries by graph coloring algorithm [14]. Thirdly, introduce a new data structure *LMRef descriptor* that tracks variables mapping in the LMRef table.

We propose to optimize the compiler to identify the blocks of each type of variables in program functions/procedures where the program/function is defined as an *initial region*. Typically, in our design the compiler has knowledge of the number of entries in LMRef table and suggested size that the process can use. Therefore, if the total size of the blocks or the number of LMRef entries required are higher than the given resources we divide some of the initial regions to basic blocks. However, largest required LMRef entry and SPM size for any basic block of that procedure are the new required LMRef entry and SPM size for that procedure. Then check again for the required resource for whole program and compare it with available resource. After determining the regions for every procedure, graph coloring algorithm has been used to map the blocks to LMRef entries [14]. A data structure *LMRef descriptor* is used to track the variables of each block.

1) Memory Data Requests: The convention to request data from scratchpad in LMStr is significantly different than conventional memory access request. In current architecture, the data is requested by CPU with its virtual address, and if the

data is available in cache it is sent to the CPU, and if not available in cache it is brought to cache from main memory with a block of consecutive bytes and sent the data to CPU. On the contrary, data is stored in LMStr in variable sized blocks, where a block consists of multiple individual data items that may not be from consecutive memory references. Therefore, to store a data block in scratchpad memory, compiler maps the block to LMRef entry and at the time of execution explicitly asks main memory to send the data block with multiple memory addresses. Thus, in LMStr, instead of requesting data with addresses, CPU request data with a pointer to an LMRef entry, an offset to get the actual position of the data in the block and size of the data.

In section III-B, we discuss how hardware manages the memory request.

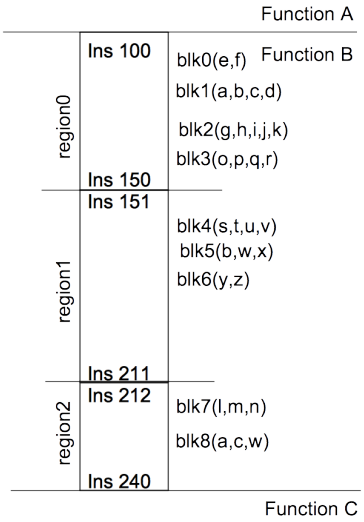

Fig. 2: Mapping of a function, region and variable blocks

2) Out-of-order Execution Management: Typically, the data blocks are mapped to different LMRef entries at compile time (*e.g.* static scheduling) that relies on the live range of data blocks on a program code in a region. On the other hand, the instructions are executed out-of-order with dynamic scheduling according to the availability of resources and required operands.

Figure 2 explains how LMStr functions that is divided into three regions, where each region has to execute a set of instruction with their own blocks of variables. These blocks consist of data items. Lets, assume data block (blk1) in region0 and block (blk5) in region1 has a common data item 'b'. Though, in static scheduling region1 would not start to execute unless region0 finishes, due to dynamic scheduling in out-of-order execution region1 may start early that may introduce co-existence of data blocks blk1 and blk5 in LMStr concurrently. Therefore, two copies of the same data variable 'b' may stay in LMStr at the same time. This concurrent existence of a variable introduces two types of design constraints as LMRef entry hazard and data duplication hazard. The design issues and optimization to resolve the hazards are explained as following,

a) LMRef entry hazard: In LMStr design blocks are mapped to LMRef entries by graph coloring algorithm that maps multiple blocks to an LMRef entry while avoiding overlap with the live ranges of blocks. In out-of-order execution multiple blocks may need to be mapped to store to an LMRef entry concurrently that introduces LMRef entry hazard. This issue can be resolved by assigning an serial number to each block (*blkSID*) that are mapped to same LMRef entry at compile time. It is possible by scanning the code after code generation phase.

b) Data duplication hazard: In LMStr memory, data to/from scratchpad from/to main memory are moved in blocks. The blocks consist of multiple data items that are accessed by different instructions. However, the block transfer sequence generated by the compiler (static scheduling) ensures that the latest data items to store in LMStr by storing all the updated/dirty data items in main memory first (un-allocate a block from scratchpad) and then bring the most updated block from main memory to scratchpad. In out-of-order execution, the block transfer sequence breaks that introduces multiple instances of a data item in scratchpad as pre-fetching the data in blocks. We notify the consistency issue as a data duplication issue and can solve it by inserting inter block data update instruction at compile time (section III-B4).

As a solution of this issue, we update a variable in a block when the variable is dirty in another block. Typically, at the end of live range of a variable in a region, the variable is updated to its next usage in another region.

B. Hardware Management Methodology

In LMStr memory model, the processor has a LMRef table with fixed number of entries. The program is compiled with the knowledge of entries in a private LMRef table has. At the time of program execution the blocks are moved to/from SPM from/to main memory by the suggested mapping generated at compile time.

Therefore, an SPM data storage has been proposed where data stored as blocks. As compiler generates blocks according to types and usage in a region, it is inevitable that blocks would be in variable sizes. However, it is required to create a bridge between compiler mapped LMRef tables and hardware controlled data storage. We propose to use a mapping table, where each entry of the LMRef table point to at least one distinct entry of the table.

However, the hardware organization is managed by a controller. This controller finds out a free place for a new block, frees storage while a de-allocation request is issued, identifies and sends the requested data to the CPU, manages SPM bypass requests and does de-fragmentation of the storage space. Typically, due to a large variability of block sizes it is certain that, holes created in storage space at the time of block de-allocation cannot be completely filled with new blocks and thus requires de-fragmentation. The controller does

the de-fragmentation while the storage space is idle. In de-fragmentation, data blocks are moved to fill the holes and corresponding shared table entry is updated with the new position of the block.

1) Block Allocation: At the time of execution the blocks are allocated to LMStr with the compiler generated LMRef mapping. A typical block allocation request consist of LMRef entry, blockSID and size of the block. At first, the allocation flag of the requested LMRef entry is checked if the LMRef entry is already allocated. If the entry is un-allocated, the LMEngn finds an un-allocated entry in LMDir table and a free place equal to or greater than size of the requested block. The index of the free place (LMBlndx), blkSID and size of the block is stored in the LMDir entry. Also the LMDir entry is flagged as data. The entry of the LMRef table of the core is updated with the LMDir index, and flagged as allocated and single.

In out-of-order execution, due to dynamic scheduling of block allocation, multiple blocks may need to be stored to the same LMRef entry concurrently. This situation arises when, a block allocation request is issued that is mapped to an LMRef entry which is already allocated by another block. The LMEngn updates the *index2* entry by allocating a second LMDir entry. The new LMDir entry is used to index to an LMBlndx that stores the blkSID, size of the block, LMBlndx, and allocation flag for each blocks mapped and stored concurrently to this LMRef entry. The entry stores the LMBlndx, number of blocks stored in the index, type of the information as non-data, and allocation flag. Now the LMEngn search for another un-allocated entry in LMDir. The index of this LMDir entry is stored in the entry of the LMRef table of the core in *index1* entry as the position of currently used block. The LMDir stores the LMBlndx, size of the data, type as data and allocation flag.

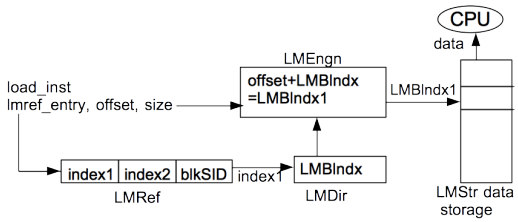

Fig. 3: Data access request flow in LMStr

2) Data Access: The data access request in LMStr memory comprises of LMRef entry, offset and size of the data. The LMRef entry index to a LMDir entry that points to a LMBlndx. The LMBlndx is the starting position of the data block that holds the requested data item. The LMEngn adds the offset with the LMBlndx to get the actual position of the requested data and send the data. Figure 3 shows the data access request flow in LMStr. Here, the position of the requested data in LMStr data storage is *LMBlndx1*. The CPU would receive the data from *LMBlndx1* to LMBlndx1 + size. However, for static scheduling, the data access request with the mentioned pattern is adequate to get the actual data. In out-of-order execution, as multiple blocks may stay with the same LMRef entry, it is needed to find its data in an efficient way. Therefore, to overcome these issues, in our design we extend the memory access request with blkSID.

Typically, every time a data is accessed it is required to check the blkSID. If the blkSID matches, use the *index1* to get the LMDir entry for the position of the data block in data storage. Rest of the strategy is exactly similar to the static scheduling. If the blkSID does not match with the stored blkSID, the second LMDir entry in *index2* is used to get the information of the blocks with same LMRef entry. The blkSID of the requested data is used to find the position of the block (LMBlndx) in data storage. Then, use the same LMDir entry in *index1* to store the LMBlndx and update the blkSID with new blkSID. The data can be accessed with *index1*. However, finding a free entry in LMDir table, updating LMDir table and searching a free place in LMStr data storage is done by LMEngn.

3) Block De-allocation: The block de-allocation request(s) require the use of an LMRef entry and blkSID. The LMRef entry index to LMDir entry/entries that point to an LMBlndx. If there are multiple blocks with this LMRef entry blkSID is checked with the blkSID stored in data storage. The second LMDir entry from index2 is used to get the block information. Then the actual LMBlndx, and the size of the required block is found with the blkSID. The data block is either removed or updated to main memory depending on the type of the block. However, if the block is recently used, the LMRef entry can be used directly to de-allocate the LMDir entry and LMBlndx of the data block.

4) Inter Block Data Update: The blocks consist of multiple data items, out-of-order block allocation introduces multiple copies of same data item in LMStr data storage. Among the multiple copies, some of them are old copies that are required to be updated before use. Therefore, an instruction consist of LMRef entry, offset, size of the data, blkSID has been issued for each data item that are required to update the old data in another blocks. Typically, only static, global and array variables require inter block data updates.

Generally, after the data item is last used in a block, the inter block data update request checks for the existence of the block containing the data item through the requested LMRef entry. If the block is found, the data is updated and if the block is not found LMEngn ignores the request. The number of inter block data update requests can be reduced by tracking every data item in a block whether they are dirty or not. It would require an extra bit for each data item stored.

5) Data storage Defragmentation: Typically, due to a large variability of block sizes it is certain that, holes created by block de-allocation in storage space cannot be completely filled with new blocks and thus requiring de-fragmentation. Therefore, it is mandatory to unify the holes to make rooms for new blocks. The controller, LMEngn does the de-fragmentation while storage space is in an idle state. In de-fragmentation, the data blocks are moved to fill the holes and

corresponding shared table entry (LMDir entry) is updated with the new position of the block.

IV. Experimental Setup

The practical implementation of LMStr memory model requires a compiler extension and memory hardware modifications. Typically, the compiler extension is required to generate blocks consisting of similar types of variables used in a region. Moreover, a compiler is responsible for generating code to move blocks back and forth to/from the SPM from/to main memory, generate data access request, and maintain data consistency between multiple copies. Therefore, it requires a significant modification in the compiler architecture to accurately use the LMStr memory model. As compiler optimization is not the focus nor the target of this paper, therefore, we have created a similar environment and generated similar blocks with a block generator simulator.

In this paper, the hardware model of LMStr memory is implemented in a full system simulator, the Structural Simulation Toolkit (SST) [15]. We have implemented the LMStr and cache functional simulator in "Ariel" processor framework that is already implemented in SST. The Ariel processor can generate multiple Ariel cores. In this framework, the application executes in multiple virtual Ariel cores in simulator environment. The memory references of the application running in a core of the Ariel processor model can be fetched with PIN tool.

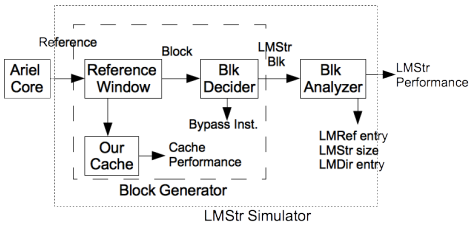

Fig. 4: LMStr and cache implementation flow chart in Ariel processor model

In our experiments, we have used the references from ariel core to generate blocks, and compared the performance of cache and our proposed local memory store (LMStr). Figure 4 explains the flow chart of LMStr model in SST. In the following sub-sections, we discuss about the block generation, block analyze, application characteristics, performance metrics, and LMStr configuration in the SST simulator.

A. LMStr Block Generator

No existing compiler(s) supports generating data blocks consist of similar type of variables or no memory simulator can access memory with blocks. However, memory access pattern and memory mapping in virtual address space show us some directions from where we can generate variable size data blocks. Current compilers are intelligent enough to make the data access pattern/data optimization in such a way that explores the data spatial and temporal locality.

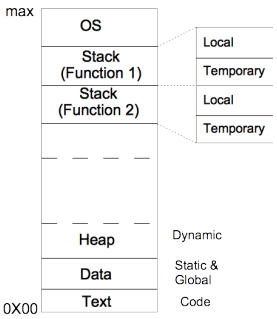

Fig. 5: Program mapping in virtual address space

Also, in current convention, the same type of data variables in a procedure or function are grouped together and mapped in segment in virtual address space. Figure 5 describes the program mapping in virtual address space where stack variables (local and temporary variables) of each function/procedure are grouped separately and mapped in two segments according to their types. Similarly, static and global variables are in a group and shared between functions.

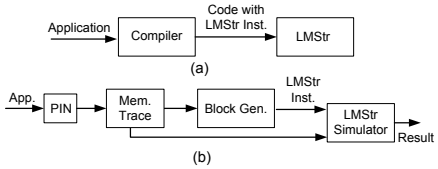

Fig. 6: LMStr block flow, (a) Actual system implementation, (b) Implemented memory model

Therefore, if we can carefully and thoroughly observe the memory references access sequences we may identify the localized data and can generate the data blocks from the localized data that may be similar to a compiler generated block. Figure 6(a) shows the block flow diagram in LMStr memory in a typical system and figure 6(b) shows the block flow in implemented LMStr memory in a simulator. In Figure 6(a), in an actual system, block is identified by compiler and moved directly to LMStr with some special instructions.

However, we need to manually identify the types of the variables in program codes and flag their address references at the time of execution. Typically, we get the virtual address mapping of the static and global variables from the *elf* executable file format. The addresses of the local and array variables are found at the time of execution. Therefore, we use *I/O* statements for each of these variables to get the runtime addresses. To identify the address of temporary variables, we need to identify all of the variables in a program code.

After a block is generated, LMStr block uses a function to decide if the block is to be stored as LMStr block or non-LMStr block. In this preliminary stage, we decide a LMStr block according to its usage count. However, in a typical implementation a cost function should be implemented in a compiler to identify the LMStr block at compile time.

In our implemented model, we generate the memory references with PIN tool [16] and then pass the memory requests through a reference window and combine the consecutive memory addresses to generate the blocks according to their types. The window is an array where each entry stores a unique address and size of the data blocks with the address. If a neighbor address appears, it is combined with the address and the size of block increases. The block generator is consists of the window and the block decider. The block decider decides if the data block is worthy to be stored in scratchpad.

B. Block Analyzer

After deciding a block as an LMStr block, it is required to analyze the block access profile to determine the required storage in LMStr and estimate the number of entries required in an LMRef table for each process. LMStr architecture suggests to prefetch the compiler generated block into LMStr at the time when the block has been first accessed. In our simulations, we generate the blocks by combining consecutive memory references to the block at the time of program execution. Therefore, we do not have any idea about the size of a block when the block has been first accessed. However, we get the actual size of the block when the block is evicted from memory. Also, the number of entries in LMRef table cannot be determined unless to be sure that the blocks which are eligible to store as LMStr block, that is also decided when the block is evicted. Therefore, we implemented an block analyzer that analyzes the generated blocks when they are evicted from the window.

C. Application

To understand the LMStr memory performances, we investigated a program code and its variables. We need to estimate the performance by mapping some of the blocks of variables in the scratchpad and the rest of them in the cache. Therefore, for performance estimation purposes, we focus on a micro-program as matrix multiplication and identified all the program variables in the program code. The code is written in C++ and implemented a parallel version in OpenMP as SST simulation toolkit supports OpenMP. We investigated the performances by storing the blocks of different variables in SPM according to their types.

Additionally, to estimate the performance for real benchmark, we use four benchmarks from the Mentevo benchmark suites in different application domain. However, we mapped all the variables to the LMStr.

D. LMStr Configuration

We have evaluated the performance of matrix multiplication by placing blocks of variables in different memory (either in SPM or cache) in multi-core system and compared the performance with cache. Typically, we identified all types of variables available in matrix multiplication and store them according to their type as blocks. At first, we measure the performance for each type separately mapped to scratchpad and finally, store all types of variables' blocks together to scratchpad.

TABLE I: LMStr Base Configuration

LMStr parameter Name	Parameter Used	Definition
Min. blk size	1 Bytes	Size of the smallest data item that can be stored in LMStr as a block
Max. blk size	1 KBytes	Maximum size of a block that can be stored in LMStr
LMRef entry	64	Number of entries initially given for LMRef
LMDir entry	128	Number of entries initially given to LMDir
LMStr size	64 KBytes	Initial size of the LMStr storage
No. of Cores	1 to 8	No of cores share the LMStr

In table I the minimum block size and maximum block size defines the size of the smallest and largest data item that can be stored in LMStr as a block respectively. Smaller minimum size improves utilization but poor for exploiting locality. Larger block size exploits locality for some applications but reduce utilization. Number of LMRef entry used to define the number of entries in LMRef. However, the LMRef entries eventually will be the same as the number of entries of block generator window. Therefore, the number of entries in the window can exploit more data locality but effectiveness of the block may reduce as a selected LMStr block may stay at LMStr for long time but may be accessed comparatively less number of times. LMStr size is needed when multiple processes are running and sharing LMStr that is our future research. Maximum use of the shared LMStr by all the applications may not be same as individual LMStr requirement. LMDir entry is also needed when multiple process share LMStr.

V. LMStr Performance

We have measured data movement, area, miss rate and utilization as performance parameters for LMStr memory model and compared it having a regular conventional cache. The data movement defines the average number of bytes moved to/from main memory from/to LMStr and data movement by bypassing LMStr.

Data Utilization of a LMStr block presents the percentage of total bytes in stored blocks been touched at least once before the blocks are evicted.

Moreover, we measured the actual scratchpad size and LMRef entries required by the process at any time of execution. The actual LMStr size required is the maximum possible storage space the process is required. However, most of the time, process occupies a lot less of this amount. LMRef entry defines maximum number of entries a process actually occupies in LMRef table at any time of execution. Data Utilization of a LMStr block presents the percentage of total bytes in stored blocks that has been touched at least once before the blocks are evicted.

In this section, we compare the performance of LMStr for different variables mapped to scratchpad memory with 64 KBs, 4-way (64-byte block, 8 words per block) L1 cache.

TABLE II: Required hardware configuration for Matrix multiplication

Var. Type	Static & Global	Local	Arrays	Temporary
LMRef	20	4	4	32(125)
LMDir	20	4	4	64
LMstr(B)	200	28	4088	5892(5412)

TABLE III: LMStr Performance Normalized by Cache Performance

Var. Type	Static & Global	Local	Arrays	Temporary
Data Move (%)	10.25	2.05	0.61	3.05
Utilize(%)	4.6	4.3	220.7	-4.27
Overhead(%)	3.64	3.38	9.58	11.61

A. LMStr Performance for Matrix Multiplication

Table II describes the actual LMStr size, LMDir and LMRef entries required to store blocks of different variables in scratchpad memory in a matrix multiplication.

In this table, the temporary variables require 125 entries in LMRef and LMDir table for not to be overlapped in same entry concurrently. However, as at runtime multiple blocks can be mapped to same LMRef entry through using blkSID, it is possible to map them with 32 LMRef entries. While multiple blocks are mapped and placed with same LMRef entries, two LMDir entries are required for each such LMRef entries. Therefore, at most we need 64 LMDir entries. However, the first LMDir entry of the two LMDir entries stores the information of blocks that are mapped and stored with same LMRef entry. We have estimated that the size of block information is 30 bits (6 bits for blkSID, 13 bits for LMBlndx, 10 bits for size and 1 flag bit) for each block. If we assume that maximum 125 blocks information need to be stored, the storage space need to be increased 480 Bytes.

Table III presents the performance of LMStr while different variables are mapped to scratchpad and rest of them are stored at cache. Though, for all variables data movement reduces, we get most reduced data movement for static and global variables mapped to scratchpad memory. However, due to additional storage component of LMStr, the overhead increases for all the variables. The data block utilization improves for all variables stored in scratchpad except temporary variables.

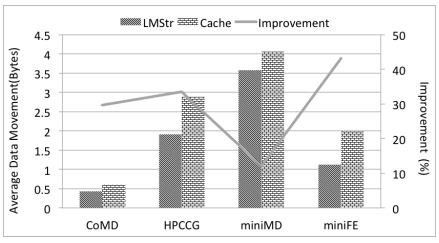

Fig. 7: Average Data Movement for LMStr and cache

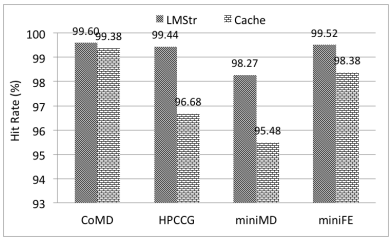

Fig. 8: Hit Rate of On-chip LMStr and Cache

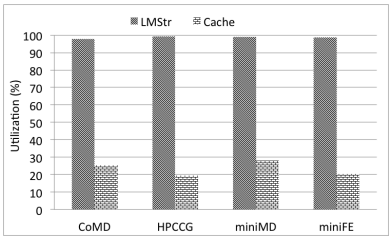

Fig. 9: Data Block Utilization for LMStr and Cache

B. LMStr Performance for Mantevo Benchmarks

We have also measured data movement (Figure 7), on-chip memory hit rate (Figure 8) and data block utilization (Figure 9) for four benchmarks (*i.e.* CoMD, HPCCG, miniFE and miniMD) from Mantevo suites. We have also compared the performance of LMStr with 64KB L1 cache. For these applications, in experiment setup we assume that there is no cache on memory hierarchy. Therefore, all the memory requests are served by LMStr memory only. Our measured result shows significant performance improvement for all the benchmarks.

In Figure 7, the columns explains the average data movement in each memory access and the dots express the reduction of data movement for LMStr memory compared to cache. The maximum data movement improvement is achieved for miniFE benchmark and miniMD gains least. However, miniMD has the lowest hit rate among all the benchmarks (Figure 8). Typically, the hit of LMStr is counted when the data request is served with LMStr data block, and miss is counted when the data request bypass LMStr and served directly to CPU from main memory. LMStr achieves a significant amount data block utilization compared to cache for all benchmarks (Figure 9). The data block in LMStr is controlled by software that ensures maximum possible usage of a block. However, the utilization never reaches to 100% due to the constraint set for minimum block size that was 8 Bytes.

VI. Related Work

This paper proposes a novel memory hierarchy for high performance computing systems that reduces data movement and increases utilization of on-chip data blocks by using a scratchpad memory. In this section, we present related work in two topic areas:

1) Placement strategies and program variables for embedded SPM,

2) Scratchpad use in high performance systems. Among the previous researches, compiler guided runtime scratchpad allocation matches with our proposed Local Memory Store.

Most of the previous work on placement strategies relies on profiling information that essentially identifies the most frequently used data/variables or the most cache conflicting data/variables as the most viable data to store in SPM.

Avissar *et al.* [5] considered global and stack variables and choose between SPM and cache. They proposed a profile-guided compiler based static allocation method that fixes SPM allocation variables at compile time but data is not relocated from DRAM to SPM or vice versa during runtime. The result shows only keeping 20% of data in SPM decreases runtime by 56% compared to cache.

Steinke *et al.* [6] and Wehmeyer *et al.* [7] considered global variables, functions and basic blocks and choose between SPM banks. Instead of using one single large SPM, Wehmeyer partitions the large SPM into banks that reduce the energy consumption by 22%.

Ranjan *et al.* [8] proposed a technique that profiles the size and access frequency of an application's scalar and array variables to minimize the total execution time of embedded applications. They minimize the memory access time by 29% to 33% over architectures of comparable area.

Panda *et al.* [9] considered arrays and scalar variables and use profiling to identify the most cache-conflict prone data to place in SPM. They reduce the overall memory latency by 30% to 33% compared to cache with same capacity.

Li *et al.* [10] proposed a compiler approach, memory coloring, based on a general graph coloring algorithm. In this approach, live ranges of arrays are split by cost benefit analysis (based on the access frequencies and data transfer cost between SPM and DRAM) at compile time, and most (basically loops) are stored in SPM during execution. This scheme finds that an array requires around 2 KBytes of SPM space for optimal array access.

Kandemir *et al.* [11] proposed dynamic allocation strategies to exploit reuse patterns by determining the maximal set of loop iterations (possibly coming from different nests in the code) that reuse the elements in the SPM. This technique reduces off-chip memory access by 39.4% over comparable memory hierarchy.

Udayakumaran *et al.* [12] propose a dynamic allocation technique that identifies program points where it may be beneficial to insert code to move a variable or parts of a variable from DRAM to SPM. The scheme reduces runtime by 39.8% and energy by 31.3% on average for their benchmarks, when compared to Avissar et al [5] by storing repeatedly used data in SPM.

Dominguez *et al.* [13] propose runtime allocation strategies focused on heap data. Their simulation results show that this method reduces the average runtime by 34.6% and the average power consumption by 39.9% for the same size of SPM fixed at 5% of total data size, when compared to placing all heap variables in DRAM and only global and stack data in SPM.

Note that the research presented on placement strategies is all done in the context of scratchpad for embedded systems, where applications are characterized by more regular computation and dense, regular memory access patterns. There is little recent work on the use of scratchpad memory for scientific applications in more high performance computing contexts. Bender et al. [17] use scratchpad and DRAM as two distinct entities in the same level of main memory. They implement their approach using the SST simulator and show that data movement can be significantly reduced for a sorting algorithm.

VII. Conclusions & Future Work

This work is introducing the concept of using a scratchpad memory as a hardware controlled on-chip memory in high performance computing system. The performance of SPM in computer system is particularly dependent on selecting the exact critical data blocks as arrays, variables, functions and basic blocks. It should be done by the compiler alongside with the programmer's suggestions.

Our block generator generates blocks on the basis of data access locality, it is required to extend the compiler to map the critical variables specially temporary variables in scratchpad memory and needs to estimate the performance for them. Currently, the LMStr simulator is a functional simulator that needs to be extended for cycle accurate simulation.

Furthermore, the actual vision of LMStr memory is to use it as shared on-chip memory in multi-core systems executing multiple multi-threaded applications. Though, in this preliminary research we simulated LMStr for single core, surely our future work will be focused on implementing shared LMStr for multi-threaded application.

In conclusion, there is a current pressing need for new on-chip memory architectures that can solve the problems of conventional memories. Current hardware controlled on-chip (*i.e.* cache) memory hierarchy suffer from many disadvantages from poor utilization, high power consumption, and increased data movement to and from main memory. As current technology trend is moving towards from multi-core to many-core high performance systems such disadvantages are further exacerbated where memory performance is vital. Therefore, future technology may proceed to software controlled on-chip memory (*i.e.* scratchpad memory) alongside with hardware controlled on-chip memory. To this end, we have proposed an on-chip memory architecture Local Memory Store (LMStr) which is a scratchpad based memory that keeps data under control of the running processes and makes the scratchpad resource sharing easier. Preliminary results show that the LMStr can achieve significant better utilization of on-chip memory and reduce data movement.

References

[1] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013, pp. 56–65.

[2] L. I. Xii, R. Banakar, R. Banakar, S. Steinke, S. Steinke, B. sik Lee, B. sik Lee, M. Balakrishnan, M. Balakrishnan, P. Marwedel, and P. Marwedel, "Comparison of cache- and scratch-pad based memory systems with respect to performance, area and energy consumption," 2001.

[3] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, 2012, pp. 376–388.

[4] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, April 2007, pp. 1–6.

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

[6] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 409–415.

[7] L. Wehmeyer, U. Helmig, and P. Marwedel, "Compiler-optimized usage of partitioned memories," in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, ser. WMPI '04, 2004, pp. 114–120.

[8] "Memory architectures for embedded systems-on-chip," in *High Performance Computing HiPC 2002*, ser. Lecture Notes in Computer Science, S. Sahni, V. Prasanna, and U. Shukla, Eds., 2002, vol. 2552.

[9] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proceedings of the 1997 European Conference on Design and Test*, ser. EDTC '97, 1997, pp. 7–.

[10] L. Li, L. Gao, and J. Xue, "Memory coloring: a compiler approach for scratchpad memory management," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, Sept 2005, pp. 329–338.

[11] M. Kandemir, I. Kadayif, and U. Sezer, "Exploiting scratch-pad memory using presburger formulas," in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, 2001, pp. 7–12.

[12] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '03, 2003, pp. 276–286.

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

[14] L. Li, L. Gao, and J. Xue, "Memory coloring: a compiler approach for scratchpad memory management," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, Sept 2005, pp. 329–338.

[15] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, Mar. 2011.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, and G. Lowney, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005, pp. 190–200.

[17] M. A. Bender, J. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. Resnick, and A. Rodrigues, "Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015, pp. 835–846.