Mehran University Research Journal of Engineering & Technology
Vol. 38, No. 2, 305-312 April 2019
p-ISSN: 0254-7821, e-ISSN: 2413-7219
DOI: 10.22581/muet1982.1902.05

Management of Scratchpad Memory Using Programming Techniques

KAVITA TABBASSUM*, SHAHNAWAZ TALPUR**, SANAM NAREJO**, AND NOOR-U-ZAMAN LAGHARI**

RECEIVED ON 25.09.2018 ACCEPTED ON 20.10.2018

ABSTRACT

Consuming the conventional approaches, processors are incapable to achieve effective energy reduction. In upcoming processors on-chip memory system will be the major restriction. On-chip memories are managed by the software SMCs (Software Managed Chips), and are work with caches (on-chip), where inside a block of caches software can explicitly read as well as write specific or complete memory references, or work separately just like scratchpad memory. In embedded systems Scratch memory is generally used as an addition to caches or as a substitute of cache, but due to their comprehensive ease of programmability cache containing architectures are still to be chosen in numerous applications. In contrast to conventional caches in embedded schemes because of their better energy and silicon range effectiveness SPM (Scratch-Pad Memories) are being progressively used. Power consumption of ported applications can significantly be lower as well as portability of scratchpad architectures will be advanced with the language agnostic software management method which is suggested in this manuscript. To enhance the memory configuration and optimization on relevant architectures based on SPM, the variety of current methods are reviewed for finding the chances of optimizations and usage of new methods as well as their applicability to numerous schemes of memory management are also discussed in this paper.

Key Words: Scratch, Scratchpad, Memory Architectures, Power Consumption, Programing Techniques, Optimization.

1. INTRODUCTION

During application execution the temporary data storage through a high-speed local memory is referred as "Scratchpad Memory". Direct memory access instructions are used to transfer the data between the main memory and scratchpad memory locations on the contrary to being copied, as in the hardware consistent approaches of most caches as shown in Fig. 1. Scratchpad offers distinctive benefits such as; savings of on-chip area, probably guaranteed dormancy, and energy usage reduction (in the situation of hard real time applications) [1-2]. In general purpose computing these attributes are also beneficial; however explicit data movement and memory-aware programming burden presents an important contest to compilers being used these days and discourages mainstream architecture adoption [3]. This manuscript will describe a system for

Authors E-Mail: (kavita@sau.edu.pk, shahnawaz.talpur@faculty.muet.edu.pk, sanam.narejo@faculty.muet.edu.pk, nooruzaman.laghari@faculty.muet.edu.pk)
* Information Technology Centre, Sindh Agricultural University Tandojam, Pakistan.
** Department of Computer Systems Engineering, Mehran University of Engineering & Technology, Jamshoro, Pakistan.

managing the scratchpad through programming, by identifying a range of current methodologies, exploring initializations and important guidelines for the enhancement as well as for improvement. An API (Application Programming Interface) in computer programming is a set of definition of subroutines, tools for building software and communication protocols. An API may be for an operating system, web-based system, software library, database system, or computer hardware. Among two shared articles or applications it works as a medium.

To maximize usability, the suggested scheme will follow an elementary set of requirements. The purpose of this research is to make the most of additional criteria and it is needed that a selected system may give preference to programmability including data locality, hardware-aware execution drive efficiency and general presentation when probable. Intermediate representations are required to provide the back for a set of functionality which is friendly with compiler intermediate and shared languages.

2. LITERATURE REVIEW

Scratch memory with the hard-ware implementations have an extensive history. The IBM Cell architecture and multicore DSP (Digital Signal Processor) chips are recent examples of concern. In these years different improved software memory management approaches for making usability of these types of architectures better have been constructed. At runtime the transfer and storage of the frequently come across forms of memory objects are of their primary concern. To load as well as to run a generic procedure from the perspective of the required functionality, different approaches of immediate interest will be discussed in this research.

2.1 Static Allocation of Memory

On scratchpad architectures the static memory distribution of code is easily recognize as appeared in Fig. 2. Size requirements of global data and fixed-size text code may statically be computed by compilers and with sufficient available capacity into a scratchpad memory these can be trivially loaded. With symmetric access to the main memory [4] and by including an instruction cache some architecture further simplifies this process. During the existence of dynamic contention or constraints on vacant memory space the job of static object apportionment turn out to be further puzzling, but supporting overlays and linkage editing can be mostly handled by loaders. Code may be limited to pointer values and load locality free addresses depending on the accessibility of virtual memory addressing modes and multitasking capabilities.

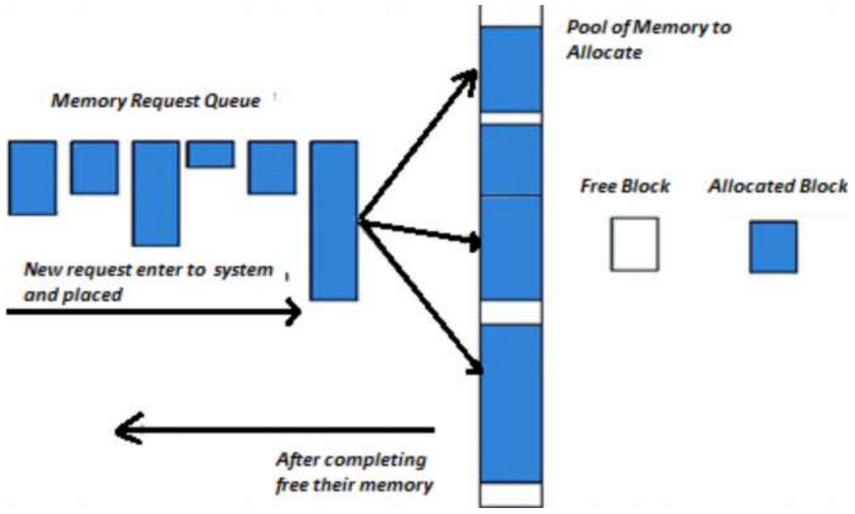
Memory Request Queue
New request enter to system and placed
After completing free their memory
Pool of Memory to Allocate
Free Block
Allocated Block

FIG. 1. MEMORY ALLOCATION EXAMPLE

By observing simple metrics, the static distribution procedure can be enhanced, such as frequency of access, sizes of data types accessed, control movement features of the adjacent code and accordingly by arranging distribution tasks. Comparative to the distinctive size of quick subroutines this is probable and only firmly required on the structural design with strictly restricted vacant memory size.

2.2 Automatic Allocation of Memory

During the application execution of scratch memory comprising systems, the stack utilization is properly maintained. Runtime allocation of a functional frame is the mutual sample of this kind of memory utilization. In the local memory of an executing application, data assigned needed to be accessible whenever required. Pointer arithmetic and simple stack variable accessed are data dependencies that may include but must not be statically resolvable. While retargeting applications to scratchpad designs in assuring performance portability, consequently efficient management of these data is very important.

Using software managed cache rule called as "Comprehensive Globular Stack Organization" [5] it was an initial effort to manage the stack. For insertion overlapping this arrangement identified the API functions for every function request in an application, stated in the earlier unit just look like overlays. Inside a fixed-size memory the system permits automatic allocation according to instantaneous application and availability requirements by loading frames from main memory and through ejecting stack structures from resident scratch memory.

Later in an innovative arrangement named as "Smooth Stack Data Organization" [6] certain restrictions of the first method were mentioned. To the complete stack space this technique lengthens the organization granularity instead of individually at each function level. By providing automated pointer management (giving functions for write back and loading among the places and at the phase of definition through transforming local addresses into global addresses), to practice a linear queue structure that make the library simple, and to computerize the attachment of API calls by relating an acquisitive algorithm. Through adjusting the location of API calls the insertion algorithm targets towards the moderate organization overhead, to the ideal cutting of a particular application's weighted call graph, formalized as applying a fixed-size constraint (according to the required stack size with weights statically assigned, for each function a directed graph containing nodes, and weighted by call count, edges for each function call). By indicating that their algorithm reaches the ideal outcome as well as for managing the function call placement, the authors deliver the formulation of ILP (Integer Linear Programming) but leaves the estimation of values and the creation of the weighted call graph as an open problem. With the nontrivial control flow actual weights of programs cannot be calculated statically.

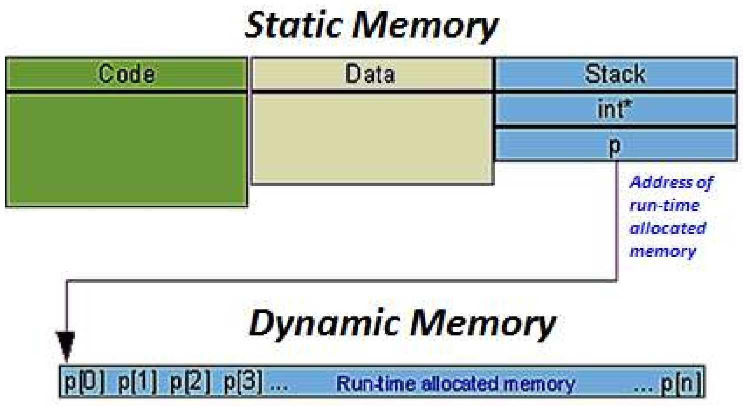
Static Memory
Code
Data
Stack
int*
p
Address of run-time allocated memory
Dynamic Memory
p[0] p[1] p[2] p[3] ... Run-time allocated memory ... p[n]

FIG. 2. BASICS OF STATIC AND DYNAMIC ALLOCATION

2.3 Dynamic Memory Allocation

As with automatic distribution mention above, at runtime the dynamic distribution of memory items is maintained, but related to data for example, heap variables that may demand a large quantity of space that is possibly indeterminable ahead of time as shown in Fig. 3. Through the system calls languages similar to C that offer dynamic allocation services used for automatic allocation rather than demanding the API call insertion process, these function calls also needs the suitable properties of interpreting distribution demands.

For the programmed heap data management one of the present approaches is known as "Completely Automatic Heap Data Organization" [7]. As per local memory accessibility, by authors this method was demonstrated clearly, for maintaining the pointer transformations among global and local address spaces by means of an implementation, that improved with additional API functions in GCC 4.1.1 the _malloc and _free functions. By means of a system of non-standard virtual memory simulation this pointer alteration can be automatically assumed, note that by the same authors in continuation

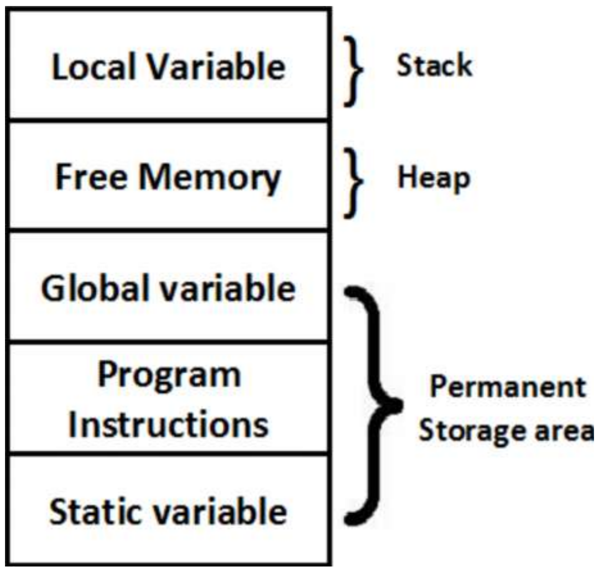

FIG. 3. MEMORY DISTRIBUTION DURING DYNAMIC ALLOCATION

to work in its replacement policy, by preventing the associativity of the software managed heap cache data, the configuration runtime overhead was reduced for newly ejected memory objects [8] by the addition of a "\victim buffer". Conversely, entire constraints were depends on application, but the architecture associating SIMD (Single Instruction, Multiple Data) comparison instructions was stated to be more effective to accelerate tag operations.

3. APPROACHES

Scratch memory with a two-part structure after fulfilling the claimed requirements, may be considered through a comprehensive generalize tactic for software management. Rules for program executions are created initially in a functionally precise manner, by using a sequence of demonstration. Later, working on one or more of these representations following a sequence of conversions are defined.

Assumptions can be validated into simplify subsets of execution logic, or others are clear in order to de obfuscate valuable belongings; the data dependence graph and the static control flow are the other included common representations. This method is analogous to that applied in most of the compilers that depend on a well-defined intermediary representation with the exception of applications comprising the hand-tuned assembly level code. Rather than code having expected degraded meaning as an outcome during the previous or else suboptimal manual optimization struggles, as compiler optimizations remain towards progress it is expected that the greatest efforts for the program writer is to develop as well as to be focus on condition that give flawless commands to a proficient tool chain. As thread numbering rise or as thread performance deviates so is the capability of the program writer to compose the equivalent code restricted also via their skill to grip synchronization, similarly area constraints for a limited code distance dynamic optimization methods applied in hardware are also restricted.

3.1 Scratchpad Memory Allocation

In two categories scratch memory applications are repeatedly divided: as a complementary storage component that can be incorporated in memory or as an individual vacant local memory stock that can be built in architecture as shown in Fig. 4. In the memory hierarchy they may be appeared at one or more levels and access constraints or follow a set of privilege that may be globally accessible whether functionally or physically imposed. Scratchpads merely support elementary modes of physical addressing only; same as in shared embedded structures it may support virtual memory only. Which particular approaches of memory distribution and management will produce the maximum advantage; deprived of emulation of lost functionality of operating systems it supports; at a high level all of these variables jointly determine the activities of the defined memory scheme.

By concentrating on the associated memory designs process, from the above segments it is concluded that the proposed memory allocation methods can be used with any of the scratch comprising scheme as long as steadiness is sustained in the illustration of objects. Effective jobs that can be controlled by automatic software defined scratch memory contain static code distribution, Subject to the scheme of the target memory system, heap and dynamic data allocation, stack frame management, software-defined shared memory places (local or virtual), message passing and inter-process communication, address translation and virtual page size adjustment.

By considering the execution and target building environment a system is proposed in which the tool chain (with or without external libraries, defined as comprising of complete phases necessary for running and building an application,) is optionally allowed to obtain the complete data. The layout and optional statistics about the order of dimension of the target runtime memory organization, and the desired level of scratchpad computerization (comprising the scratchpad access rules as well as capabilities implicitly) are the key parameters included. Anywhere when constraints are often mostly sensitive the latter is non-compulsory in all circumstances however can produce countless aid meant for embedded structures.

For static memory allocation scratches are mostly favorable. To further optimize the applications and the usage of runtime profiling devices that would deal with practical data the approaches are previously defined that are designed for static allocation can be improved, but in the situation of difficult applications this method would need extra computational resources but not essentially assured about improve outcomes. When needed static memory modifications can be performed manually and it is therefore recommended that for static objects whenever capacity allows allocation in a heterogeneous memory order it supports scratchpads.

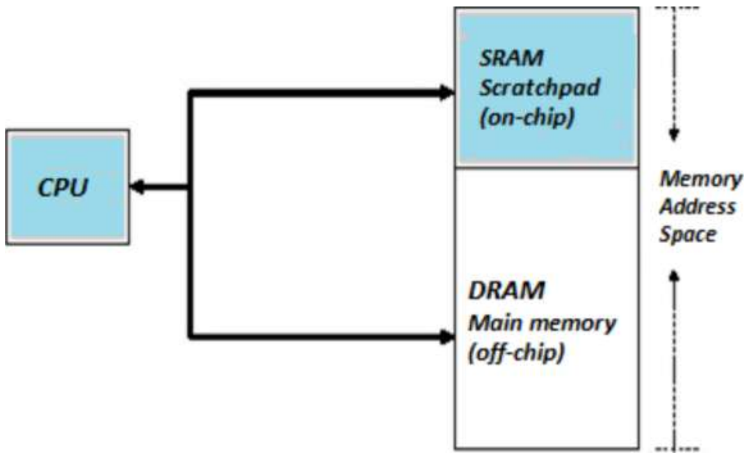

FIG. 4. SCRATCHPAD MEMORY ALLOCATION

In the literature graph the theoretic approaches may create provably exact optimizations of stack variable distribution. Near the execution environment the unique chance to increase efficiency is then to make use of supplementary data. By application-specific assumptions and/or architecture-specific features this contains techniques using which the management overhead can be additionally decreased. Many of the run time calls needed by the referenced systems comes to be pointless, while virtual memory management is providing as an example. The frame in a hardware-managed cache system memory inside an effective development frame may be controlled nearly identical to such a procedure, related to the procedure for avoiding defaulting distribution to caches by means of the exemption where these observations may target scratchpad positions. By means of any appropriate arrangement of instruction-level parallelism, specific to scratchpad memory management even in architectures with hardware acceleration abilities, supposing scheduling necessities can be seen, or even by using SIMD assessment instructions can be added in architectures enhancements symmetric to the speeding up of tag processes.

At a higher level to take memory distributions equally as a data movement problem, is an another direction for improvement. Limited hardware managed cache coherence or for example, those presenting new procedures of non-uniform memory access in more composite hybrid memory designs, the method can be stretched for maintaining optimization at a system-level perspective. For data accesses to all of these dissimilar memory sections such that predictable costs can be determined a model can be parameterized, the optimization process can further be informed. From each element the expected band-width, access rules (containing latency, approvals and request routing in the case where alternating routes be present) and the arrangement of the memory order are the important system level characteristics included. To guide greedy optimization algorithms more robust cost functions or via simple estimated cost parameters these can be used to guide if the above attributes are provided. Related to those using in the commercially accessible processor generator project flows with extra detailed graphical models, for describing the data transfers and memory access operations will be created also that would authorize for extra precise cost functions. On the target system thereof by means of profiling illustrative benchmark code and further possibility would be to develop these overheads or optimization curves. To perform design space exploration, the resulting software setting comes out to be a very great tool for system engineers rather than depending only on the distinctive constant, inside target explanation data, if a tool chain were capable to access the hardware explanation models.

For advanced automatic heap management, the further improvements for assuring the occurrence of the bundle of data in local memory with reducing overhead which is included in this method, while when to support and called access patterns are expected (or at least probabilistic) to maintain the prefetching of heap data. By the runtime profiling or static analysis such patterns may be determined. Wherever deterministic access patterns are already known [9] for moving data efficiently the hardware prefetching allows the probable software prefetching but does not get the hypothetical overhead.

3.2 Optimization

By the hardware on which it runs, the implementation of software memory management effectively remains primarily restricted. For competently overlapping or interleaving computation and communication the hardware embodiments suggest favorable procedures. Through double-buffering through three cache stages [10], on DSP platforms to efficiently mask data movement overhead among memory places the accessibility of DMA (Direct Memory Access) relocations was presented. Through high bandwidth on-chip networks many core designs provide similar capabilities; from data locality software on these designs significantly benefits, as power consumption and high latencies in off-chip DRAM (Dynamic Random Access Memory) accesses remain evaded [11]. By the balance resources (e.g. functional unit operand counts, SRAM (Static Random Access Memory) port counts and register latency) the highest possible efficiency of memory management is basically restricted in any of these cases.

Whether incorporated in a tool chain or provided as a support library, it is usually required that any scratch memory management system permits optimization, that will affect by the user's capability to link code as well as debug code amongst unlike languages or operate without lacking specific source-to-source conversions or that includes the program coded in assembly language. Optimization authorizations must create the usage of every providing target account data when enabled. For the architectures of interest memory object packing are exactly fit and certain techniques in particular related to vectorization and auto-parallelization are also observed. When the optimized program configuration maps better in the target arrangement these methods can create code and accessible resources of the architecture can be efficiently utilized further.

According to the polyhedral model [12], which is one of the specific methods of basic concern where optimization applied. Insides a nested loop organization, a bulk of execution logic present in scientific and high performance computing where these methods are particularly useful. Polyhedral optimization that may be measured as a commonly appropriate method, however several of these restrictions can be achieved by bearing random conventional rules previously but now it can be properly characterized in this model and there are so many other limitations that need to be fulfilled by a code section.

As activities linked using loop caches or further dynamic logic might be matched by the backbone through a program, in this approach scratchpad memory is considered to be the most common representation of local memory to design. Our concept of a memory item is clear as one through which a superset of the above-mentioned distribution approaches stays related. Spreading further than current methodologies that attempt to simply maximize metrics of data locality throughout the automatic distribution on scratch-pad schemes by suggesting relating analogous methods to the hardware conscious ne-grained management of memory objects in addition to the use of present techniques valid to converts on loop constructions.

Including loop tiling the efforts of automatic parallelization have produced many methods. Caching approaches may increase the general performance of a processor but decreases the speed of the system [13]. As bandwidths and memory sizes may be trivially resolved when measured in scratchpad design the basic premise of tiling will be significantly simple. About an execution environment if suppositions around worst-case access potential can be prepared, some classes of non-uniform, tiling alterations following all uniform, and synchronization (reliant on the memory model in consideration or pipelining) data the requirement come to be somewhat available. Further if Static control flow likewise come to be more simply decidable (this concluding fact is mainly dynamic by collective core plus execution part calculations), if hardware interrupts can be supposed to happen at suitably low frequency supposed to be unimportant or are not implemented. Data dependence becomes quite accessible as these may be deprived of loss of generalization equally represented; within an execution component separately further vectorization or SIMDization is not considered.

Preceding an iterative succession of passes the suggested technique depends. The compulsory illustrations comprising the data reliance graph, particularly the task intermediary arrangement or an AST (Abstract Syntax Tree), control flow graph (allowing to statically-resolved call calculations using weights given) and some related specified lower-level representations are first produced. According to the existing particulars about accessibility of resources and the execution setting the application can initiate to be covered. At higher levels of abstraction these tiling's begin where become progressively lower level and the highest communication latencies, when the in-order issue is known and available.

However, for resolving the specific systems when related, the ILP designs of the above system can be invoked. For producing ideal outcomes, a range of estimation methods can be placed in circumstances where under time constraints or required assumptions may not be made.

4. CONCLUSION

For the data access requirements and by utilizing a C application as a base in a principally hardware agnostic way, an innovative methodology for the program management of scratch memory has been defined. Using a new proposed approach based on previous works, a number of known techniques are combined whereas elementary modules are applied in place of tool chain extensions. Totally novel techniques to optimization as well as important opportunities for upgrading of every task are identified. To develop the productivity of data movement in hybrid memory systems scratchpad memory holding configurations or its modules may possibly be used by applications and mutually, this memory management scheme should suggest significantly enhanced programmability.

ACKNOWLEDGMENT

Authors would like to acknowledge respectable referees for their valuable recommendations that directed us to improve the worth of the manuscript. Authors are also thankful to the Institute of Information & Communication Technologies, Mehran University of Engineering & Technology, Jamshoro, and Sindh Agricultural University, Tandojam, Pakistan, for providing platform to carry out this research.

REFERENCES

[1] Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P., "Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems", Proceedings of 10th ACM International Symposium on Hardware/Software Codesign, pp. 73-78, 2002.

[2] Shivaraj, K., and Dharishini, P.P.P., "Design and Simulation Analysis of Time Predictable Computer Architecture", MSRUAS-SAS Tech Journal, Volume 14, No. 1, pp. 5-8, January, 2015.

[3] Pena, A J., and Balaji, P., "Toward the Efficient Use of Multiple Explicitly Managed Memory Subsystems", IEEE International Conference on Cluster Computing, pp.123-131, 2014.

[4] Verma, M., Wehmeyer, L., and Marwedel, P., "Efficient Scratchpad Allocation Algorithms for Energy Constrained Embedded Systems", Power-Aware Computer Systems, pp. 41-56, Springer, 2005.

[5] Kannan, A., Shrivastava, A., Pabalkar, A., and Lee, J.-e., "A Software Solution for Dynamic Stack Management on Scratch Pad Memory", Proceedings of Asia and South Paci_c Design Automation Conference, pp. 612-617. IEEE Press, 2009.

[6] Lu, J., Bai, K., and Shrivastava, A., "SSDM: Smart Stack Data Management for Software Managed Multicores", Proceedings of ACM 50th Annual Design Automation Conference, pp. 149, 2013.

[7] Bai, K., and Shrivastava, A., "Heap Data Management for Limited Local Memory (LLM) Multi-Core Processors", IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 317-325, 2010.

[8] Bai, K., "Compiler and Runtime for Memory Management on Software Managed Manycore Processors", Ph.D. Thesis, Arizona State University, 2014.

[9] Lee, J., Kim, H., and Vuduc, R., "When Prefetching Works, when it Doesn't, and Why", ACM Transactions on Architecture and Code Optimization, Volume 9, No. 1, pp. 2:1-2:29, March, 2012.

[10] Gao, Y., "Automated Scratchpad Mapping and Allocation for Embedded Processors", Ph.D. Thesis, University of South Carolina - Columbia, 2014.

[11] Mattson, T.G., Van der Wijngaart, R., and Frumkin, M., "Programming the Intel 80-Core Network-on-a-Chip Terascale Processor", Proceedings of ACM/IEEE Conference on Supercomputing, pp. 38, 2008.

[12] Grosser, T., Groesslinger, A., and Lengauer, C., "Polly Performing Polyhedral Optimizations on a Low-Level Intermediate Representation", Parallel Processing Letters, Volume 22, No. 4, pp.1250010, 2012.

[13] Javaid, Q., Zafar, A., Awais, M., and Shah, M.A., "Cache Memory: An Analysis on Replacement Algorithms and Optimization Techniques", Mehran University Research Journal of Engineering & Technology, Volume 36, No. 4, pp. 10, Jamshoro, Pakistan, October, 2017.