EXECUTION PHASE PARTITIONING FOR DATA INTENSIVE APPLICATIONS

by

Mehmet Ucar, B.S.

THESIS

Presented to the Faculty of

The University of Houston Clear Lake

In Partial Fulfillment

of the Requirements

for the Degree

MASTER OF SCIENCE IN COMPUTER ENGINEERING

THE UNIVERSITY OF HOUSTON CLEAR LAKE

July. 2016

EXECUTION PHASE PARTITIONING FOR DATA INTENSIVE APPLICATIONS

by

Mehmet Ucar

APPROVED BY

[redacted]

Hakduran Koc, Ph.D., Chair

[redacted]

Thomas Harman, Ph.D., Committee Member

[redacted]

Liwen Shih, Ph.D., Committee Member

[redacted]

Ju Kim, Ph.D., Associate Dean

[redacted]

Zbigniew Czajkiewicz, Ph.D., Dean

DEDICATION

I dedicate this thesis work to my wife, Neslihan Ucar. I thank and love her and our children, Esat Emin and Ekrem for their continuous love and support in my life. I could not have made it without them.

ABSTRACT

EXECUTION PHASE PARTITIONING FOR DATA INTENSIVE APPLICATIONS

Mehmet Ucar, M.S.

The University of Houston Clear Lake, 2016

Thesis Chair: Dr. Hakduran Koc

In today's world, embedded systems run applications that process large amount of data. This necessitates the development of new techniques in order to run such applications more effectively. In this thesis work, we present a technique, called ultra-dynamic memory management scheme, in order to improve the performance of data-intensive applications running on single and multi-core embedded systems. Data intensive applications typically consist of many loop nests processing multi-dimensional arrays. The proposed approach partitions a given data-intensive embedded application into execution phases in an effective way. The target architecture uses software-managed on-chip memory components, i.e., Scratch-Pad Memory (SPM). Considering the fact that dynamic management schemes for software-managed memories are better than the static ones where the entire application is considered as one execution phase, our approach investigates the possibility of dividing a loop nest into multiple execution phases using various loop transformation techniques in order

TABLE OF CONTENTS

Chapter

LIST OF FIGURES

1. INTRODUCTION

1.1 Scratch Pad Memory (SPM)

Even though general-purpose computers have more computational power and better performance, embedded systems have become more popular in recent years due to their effectiveness in terms of power consumption, portability, size, and the availability of numerous applications. In general-purpose computer architectures with hardware caches, memory accesses are very costly due to high latency between the processor and off-chip memories. To reduce the number of off-chip memory accesses, on-chip Scratch Pad Memories (SPMs) are widely used in recent years. Latency problem is a big concern for computer developers especially for chip multiprocessors because all cores should use limited bus bandwidth to access off-chip memory. On the other hand, SPM based architectures have their own on-chip memories to keep frequently used data blocks and to resolve the latency problems. Severe latency in multi-core architectures may cause bus interventions and negatively affect the performance of the entire architecture. Software-managed memories are employed in multi-core architectures to resolve the latency issues. Such on-chip memories are proven to improve overall performance significantly especially in data-intensive applications.

SPMs are considered as very high speed internal memories because the access time to temporary local memories is very low. However, SPMs have very limited

designs in embedded systems, each core needed to have its own SPMs to maximize the performance.

1.2 Data-Intensive Applications

The data-intensive applications consist of many loop nests and multi-dimensional arrays. In data-intensive applications, the total program code size takes very limited space but cause heavy loads for processors. In other words, program codes take very limited portion of total data blocks in such applications. For example, image-processing applications contains many loop nests. These loop nests can be analyzed and dynamically allocated using the matrix model computation as discussed in this thesis.

1.3 Matrix Model Computation

The Matrix Model Computation (MMC) is considered to be a virtual data re-computation technique to organize data for embedded systems. This technique applies to only on-chip memories. The separated and re-computed phase blocks are stored in separate on-chip memories in order to efficiently use on-chip memories in embedded systems. The proposed work brings significant performance gain and reduces energy consumption in embedded systems.

We propose to utilize Matrix Model Computation in order to partition the embedded application into execution phases in a more efficient way. The proposed approach mainly provides better mapping between processors and SPMs and improve the overall performance. The MMC method optimizes execution time, memory

We have 9 operations and 9 different outputs represented as A,B,C,D,E,F,G,H,I. Therefore, Matrix 1 will be a 9x9 matrix.

We only needed 4 variables: so Matrix 2 will be a 9x4 matrix.

Matrix 1 shown below is an iMMC matrix. In the second step. we need to optimize matrix to get cMMC.

The first matrix is

$$\text{Matrix 1}=\begin{bmatrix} A & - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & a8 & H & - \\ - & - & - & - & - & - & a9 & a9 & I \\ a2 & B & - & - & - & - & - & - & - \\ - & - & - & - & a6 & F & - & - & - \\ - & - & - & a5 & E & - & - & - & - \\ - & a3 & C & - & - & - & - & - & - \\ - & - & - & a7 & - & a7 & G & - & - \\ a4 & - & a4 & D & - & - & - & - & - \end{bmatrix} \quad \text{and} \quad \text{Matrix 2}=\begin{bmatrix} a1 & a1 & - & - \\ a8 & - & - & - \\ - & - & - & - \\ - & - & a2 & - \\ a6 & - & - & - \\ a5 & - & - & - \\ - & - & - & a3 \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

Figure 2. Matrix Forms of Given Operations

1.3.1.2 Refactoring

The Canonical Matrix Model (cMMC) is constructed by organizing a sequence of the given information. Canonicalization is the process of forming cMMC. which guarantee no violation of data-flow dependency between the computation elements.

We need to refactor the cMMC form as much as possible to get the best results. The rearrangement of the row sequence data is known as refactoring cMMC. Refactoring is basically done by swapping rows. Refactoring also known as code allocation. As a result of cMMC, we produce a matrix that all output values are diagonal.

1.3.1.3 Partitioning the Computation

In this stage, we divide the refactored cMMC matrix into different blocks to be mapped to the separate processing elements. In example 1, we consider partitioning by block matrixes. Matrix blocks can be valid for partitioning. In order to partition a matrix into matrix blocks; each matrix block should have the following characteristic:

- Data-flow independent from another matrix block (no data dependency between matrix blocks)
- Easily identified from cMMC (iMMC does not carry enough information to identify matrix blocks)

In the first example, we already generated refactored cMMC matrix form; now we divide this matrix into 3 different matrixes;

$$\text{Matrix1.1}=\begin{bmatrix} A & - & - \\ a2 & B & - \\ - & a3 & C \\ a4 & - & a4 \end{bmatrix} \quad D$$

$$\text{Matrix 1.2}=\begin{bmatrix} D & - & - \\ a5 & E & - \\ - & a6 & F \\ a7 & - & a7 \end{bmatrix} \quad G$$

$$\text{Matrix 1.3}=\begin{bmatrix} G & - & - \\ a8 & H & - \\ a9 & a9 & I \end{bmatrix}$$

Figure 6. Services Matrixes for the Given Operations

the execution. However, in static allocation, data is statically placed into memory and each memory block holds its own unique data set.

We compare static data allocation versus ultra-dynamic data allocation which data is recomputed using MMC. Data is recomputed and placed into the memory block at the compile time in static allocation. However, we apply MMC in the execution level, and place data blocks into SPMs in this thesis work. To gain as much performance as possible, we also reduce the off-chip memory accesses as much as possible. In the experimental evaluation, we use both a single-core and a four-core embedded architecture and run data-intensive benchmark programs. Experimental evaluation and results clearly indicate that matrix model computation based dynamic data allocation approach improves overall performance.

1.5. Organization of Chapters

The rest of the thesis is organized as follow: Chapter 2 presents the related work; Chapter 3 explains matrix model computation in detail; Chapter 4 describes the use of scratchpad memories in embedded systems; Chapter 5 presents the system architecture and an example, Chapter 6 reports the experimental results and Chapter 7 includes conclusion and the future work.

Panda et al. [2] focuses more on-chip memory space utilization in embedded system applications. The main idea is to efficiently exploit on-chip Scratchpad memories by placing some application variables into on-chip scratchpad memory and off-chip cache. The target is to minimize the total execution time in embedded architectures. In the given embedded architecture, both the cache and scratchpad SRAM have only a single processor clock cycle, but off-chip memory takes tens of clock cycles to access. The main difference between hardware cache and the scratchpad memory is that scratchpad SRAM guarantees the access latency, but cache access time may be unpredictable because of capacity, compulsory and conflict misses. It is proposed to maximize execution time of the embedded application by a partitioning of scalar and array variables into off-chip DRAM and Scratchpad SRAM memories. In the proposed partitioning framework: some of the features that affect the partitioning process can be listed as;

- Scalar variables and constants
- Size of arrays
- Life-times of array variables
- Access frequency of array variables
- Conflicts in loops

It is important to use the on-chip memory space as efficient as possible. The given strategy, partitioning scalar and array variables into scratchpad memories, shows improvement in memory access latency of 30% over on-chip memory with random partitioning applications.

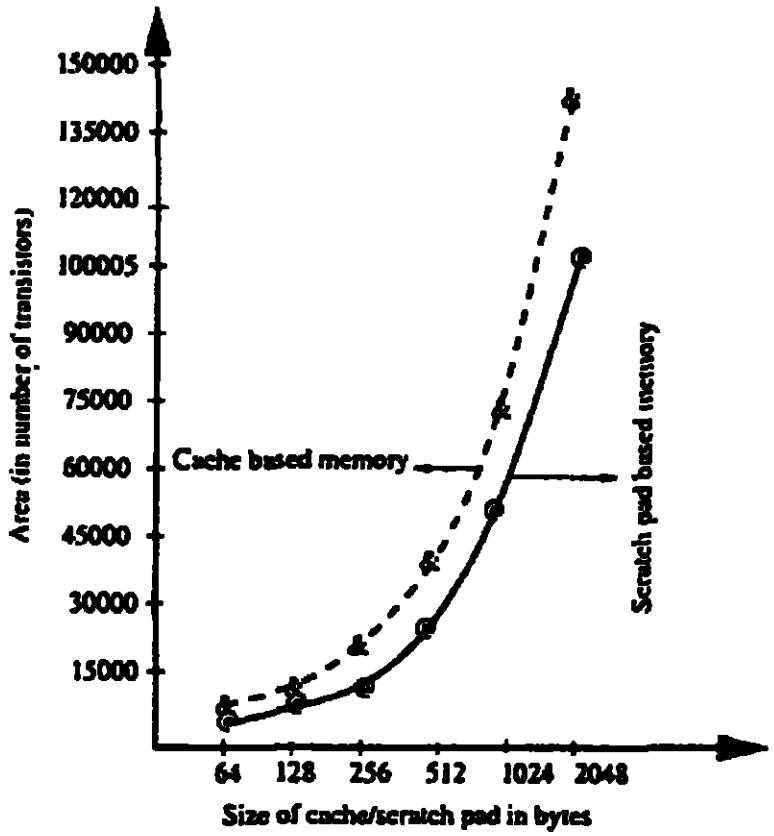

Figure 7 – Comparison of Cache and Scratchpad Memory Area [1]

Various devices are used to clearly express the advantages of using Scratchpad Memories. In both area/performance tradeoff and energy consumption, SPMs are proven to be more effective.

Janapsatya et al. [5] uses a dynamic allocation scheme which manages the instruction code between scratchpad memory and main cache memory. This also helps to increase performance, to reduce energy consumption, and to improve memory optimization.

Koc et al. [6] uses temporary array elimination as a method of memory space reduction. This is also referred to as a data re-computation technique because this temporary array elimination proposes to remove some temporary array data and

Pissanetzky [13-20] discusses MMC (matrix model computation) as a turing-equivalent virtual machine for mathematical analysis of systems. Some practical applications of the MMC are also discussed in Pissanetzky's articles. MMC proposed to mathematically analyze and represent systems. We are analyzing the data set in order to effectively place and allocate them into Scratchpad memory blocks.

Nguyen et al. [23] proposes an allocation scheme in which the size of the SPM is unknown during the execution time. Execution time and SPM size are usually related. If we use a different sizes of SPM memories in different architectures, architecture may not work properly. This research suggests a solution in which a software decides scratchpad memory allocation before the execution starts. So, once the execution process begins, we would know the SPM sizes during the execution. In this proposed idea, first SPM allocation is decided, then the given instruction block is modified, and recomputed program code is mapped to SPM blocks which are close to the processor.

Yang et al. [27] uses dynamic memory allocation in embedded systems to reduce off-chip memory access overheads caused by transferring data between SPMs and off-chip memory. The proposed compiler-assisted-iteration-access-pattern-based space overlapping method (ISOS) is an SPM space management technique. It results in significant runtime performance and improvement over static allocation schemes.

Jurnik et al [28] uses some extra methods to reduce off-chip memory accesses. These methods use relatively larger on-chip memories and avoid direct mapped caches, which cause an increase of memory access times. Dynamic run-time applications are proven to be more effective than static allocation schemes.

and/or decompressed. Research states that the code complexity keeps increasing, so code re-computation become harder to do. Code manipulation requires more memory space. Again, reducing costly off-chip memory accesses is very important in order to increase performance. For this research, Ozturk uses a compiler based approach which uses data compression to optimize memory space. The main goal is to find out what data block would be the most efficient to compress. There are two approaches used in this approach; static approach and dynamic approach. In static way; some processors are doing the compression and decompression and others do execution. However, in dynamic approach processors do compression/decompression and program execution at different time frames.

McIlroy, Dickman and Sventek [36] represents a dynamic memory allocation idea to increase performance in embedded architectures. This approach uses dynamic heap allocation in embedded systems. SPMs are very advantageous to use in embedded systems due to low latency, simple design, and predictable nature.

Ilya Issenin et al. [37] uses a technique to modify the program to place the data into SPM as much as possible considering the access latencies and dependencies.

Egger et al. [38] represents dynamic memory allocation technique with a memory management unit. In this memory management unit, physical addresses of SPM are mapped into a virtual memory. Frequently used data blocks are mapped into closest SPM blocks so energy consumption would be reduced. This approach uses profiling data and categorizes program into three regions: page-able, cacheable, and un-cacheable. Then cacheable blocks of the program are mapped into SPMs. In

with different memory designs which help optimize applications with compiler support. This research shows a matrix or vector abstraction which provides the information used by compiler along with existing memory hierarchy. With compiler help, vector/matrix information develops new memory hierarchy to increase performance in embedded architectures. In this approach. information is written as mathematical abstraction which shows flow of data from/to SPM memory blocks. Proposed algorithm is proven as an effective way to optimize the code and memory hierarchy.

Avissar [61] uses a compiler based approach to partition data into available data blocks. Both ILP and software support are used in this approach. So, data is automatically distributed into multiple memory banks effectively.

Udayakumaran et al. [65] proposed a dynamic allocation for embedded systems. Advantages of SPM usage in embedded systems include less area. faster execution time. and less energy consumption. This research uses dynamic allocation for global and stack data. In the proposed dynamic allocation, program requirements may be changed during the execution time. This approach is very effective compared to static allocation solution. In addition, memory access times are very predictable in this approach.

Yemliha et al. [66] uses many techniques to reduce resources and energy consumption in embedded architecture. Some of these techniques are code re-writing, loop scheduling. code and data placement. Code is rebuilt in this research to increase performance. This allocation is done with compiler support. After code allocation the proposed research utilizes locality with parallelism diagram representation. In circle

Ozturk et al. [77] discusses a new energy minimization metrics along with memory and performance metrics. Mini devices become very popular, and they should consume less energy for effectiveness, the article uses a memory partitioning technique which partitions memories into banks and places idle banks into low power mode. In addition, data compression is also used to reduce used memory space. This allow to increase in-idle memory banks to minimize energy consumption.

Koc et al. [78] shows an ILP formulation to re-computation to SPM management. This research also proposes to reduce the number of off-chip data using the values of on-chip SPM data. IPL stands for integer linear programming. ILP decides what SPM block to be used and makes recomputation decisions. So, application is being optimized to increase performance. There are six benchmarks used in this research and around 12% performance increase is achieved.

S. Yang et al. [79] focuses on optimizing memory usage in embedded architectures.

Olivier Temam et al. [84] proposes a technique which uses cache conflicts to reduce execution time when data is copied. Kim et al. [23] shows an idea to keep stack data in SPM blocks in embedded system applications.

Object Oriented software developers first used refactoring in their code. However, human help was required in refactoring object-oriented code. On the other hand, many refactoring tools are developed to take the human factor out.

MMC has proven to be a great data refactoring technique. MMC is used as a re-computation technique in this thesis work and is proposed to solve the allocation problem. We will use MMC recomputed data in SPM blocks and calculate performance gains using some common data intensive benchmarks.

3.2 Practical Example and Notations

We will demonstrate MMC in the example below. This example illustrates the notation, and definition of MMC.

We randomly chose eight operations.

- a=v1-v2
- h=v1-g
- i =g + h
- b=a/v3
- f= e* v1
- e=d/v1
- c=b+v4
- g=d-f
- d=a*c

a7. a8, a9) and computed outputs as (P, Q. R. S, T, U. V. W. X) in a matrix called Services (C) matrix. The fourth column shows the input variables in order and it is called a sequence matrix (Q).

iMMC was developed as an imperative form of MMC. It is a universal touring-complete virtual machine. According to Pissanetzky[15], iMMC consists of two sparse matrices. The first matrix is the services matrix C. which holds the computation operations. The second matrix is called a sequence matrix. which holds the sequence of the data.

Matrices can be modified for better performance during the computation. So, software code can be moved back and forth. Figure 9 shows the services (C) and sequence (Q) matrices for the given data operations.

$$C=\begin{bmatrix} - & - & - & a1 & - & a1\,V & - & - \\ a2 & - & R & - & - & - & - & - \\ P & - & - & - & - & - & - & - \\ - & - & - & - & - & - & a4 & a4\,X \\ a5 & a5 & - & S & - & - & - & - \\ - & - & - & - & a6 & U & - & - \\ a7 & Q & - & - & - & - & - & - \\ - & - & - & a8 & T & - & - & - \\ - & - & - & - & - & - & a9 & W \end{bmatrix} \qquad Q=\begin{bmatrix} - & - & - & - \\ - & - & - & a2 \\ a3 & a3 & - & - \\ - & - & - & - \\ - & - & - & - \\ - & - & - & a6 \\ - & - & a7 & - \\ a8 & - & - & - \\ - & - & a9 & - \end{bmatrix}$$

Figure 9. Services and Sequence Matrices for the Given Data Set

Matrix C, stands for services matrix, and Matrix Q stands for Sequence Matrix. We have four constant inputs in the given data set. So. Matrix Q has four columns.

As discussed earlier. the iMMC is generated as a result of the first given data set. This data is converted into a matrix form. We use the MMC as a refactoring algorithm and reduce the system as soon as possible by removing redundant values. Furthermore. we can rearrange codes to reduce the processing time. So, refactoring

We will have two matrices as a result of cMMC application. A services Matrix, and an argument matrix. The services Matrix should be in a canonical form. The given data set is organized accordingly in the services matrix. The computed outputs (P,Q,R,S,T,U,V,W,X). represent the operations. All computed values are lined up diagonally, and all input variables are listed below the canonical line. Figure 11, shows the matrix representation of the cMMC form of the given data set.

$$C=\begin{bmatrix} P & - & - & - & - & - & - & - & - \\ a7 & Q & - & - & - & - & - & - & - \\ - & a2 & R & - & - & - & - & - & - \\ a5 & a5 & - & S & - & - & - & - & - \\ - & - & - & a8 & T & - & - & - & - \\ - & - & - & - & a6 & U & - & - & - \\ - & - & - & a1 & - & a1 & V & - & - \\ - & - & - & - & - & - & a9 & W & - \\ - & - & - & - & - & - & a4 & a4 & X \end{bmatrix}$$

$$Q=\begin{bmatrix} a3 & a3 & - & - \\ - & - & - & a7 \\ - & - & - & a2 \\ - & - & - & - \\ a8 & - & - & - \\ - & - & - & a6 \\ - & - & - & - \\ - & - & a9 & - \\ - & - & - & - \end{bmatrix}$$

Figure 11. Services and Sequence Matrices for the Given Data Set

Canonicalization process guarantees that no data dependency issues would happen. So, the computed output can be analyzed and placed into separately operated processors, or separate data storage units. Scratchpad Memory architectures usually consist of many separate memory blocks. Placing frequently used data blocks into SPMs are very crucial because SPMs have very low access latencies. Building

In order to place the data elements into SPMs for best performance, we use the following steps while re-computing data:

1. Matrix Modeling: to place the given code into a MMC form
2. Refactoring: to produce a cMMC
3. Partitioning: to partition the computation
4. Placing: to place the partitions to SPMs

The last step is to move the data intensive data into SPM blocks. As a result of the Matrix Model re-computation, we need to have the data refactored and placed into designated SPM blocks to minimize the data access latencies and improve the performance.

Pissanetzky et al. [14] discussed the MMC in code applications. Figure 13 shows the optimal output of the data set.

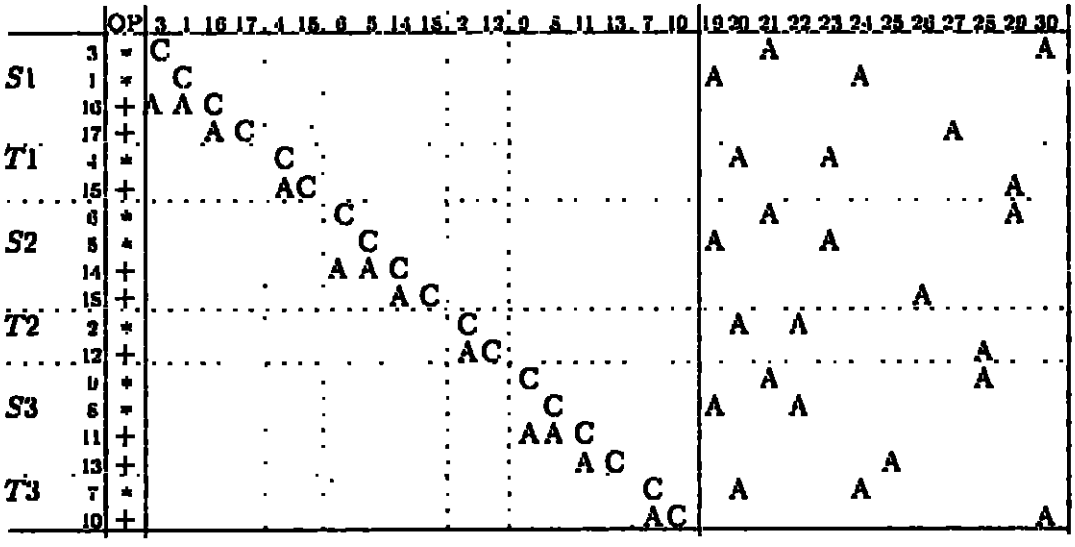

Figure 13. Ideally Recomputed Data Set in cMMC Form [14]

4. SCRATCH PAD MEMORIES IN EMBEDDED SYSTEMS

Scratch Pad Memories are a common type of Software Managed Memories. They are commonly used in embedded architectures because of their advantages over the hardware based cache counterparts. Both hardware managed and software managed memories are used to manage the on-chip caches effectively in embedded architectures. Hardware managed caches are usually not preferred in data intensive applications because of high hit rates and limited performance. On the other hand, SPMs provide higher performances because of minimal access latencies and data usage patterns. Some architectures may have both hardware-managed memories and software-managed memories. Many NVIDIA GPU Units like GTX470 and GT110 are examples of software managed architectures. But, developers still consider using hardware-managed caches in GPUs depending on the running applications.

General purpose computers use a wide variety of applications. These computers have on-chip and off-chip memories. including L1, L2, L3 caches. of chip RAM memory, and hard drives as hardware components. The embedded systems can not be used in general purpose computers because a wide variety of applications can not be optimized easily in embedded systems.

The embedded systems are used for specific applications only. For example, smart phones can be very high end embedded systems because smart phones and tablet computers are at the crossroads between embedded systems and general

Static allocation and dynamic allocation are two main types of memory management approaches in SPM management. Both static allocation and dynamic allocation were developed to increase performance and use SPMs as effective as possible.

In static allocation, data sets are analyzed and placed into SPM blocks. This statically placed data is fixed and can not be edited, moved, deleted, copied, or shifted during the execution time. So, there is no way to change the data in static allocation during the execution.

However, dynamic data allocation methods allow us to manage data during the execution. So, data in SPM blocks can be changed based on the application. Data can be moved back and forth in order to maximize the performance and access the most frequently used data sets in low latencies.

4.2. Differences Between SPMs and Hardware-Controlled Caches

The hardware controlled cache memories:

- Have high capacity to store large applications
- Are commonly used in general purpose computers
- Do not require software to manage the memory
- Have high energy consumption and increased area needs
- Have unpredictable data accesses
- Are less reliable than SPMs

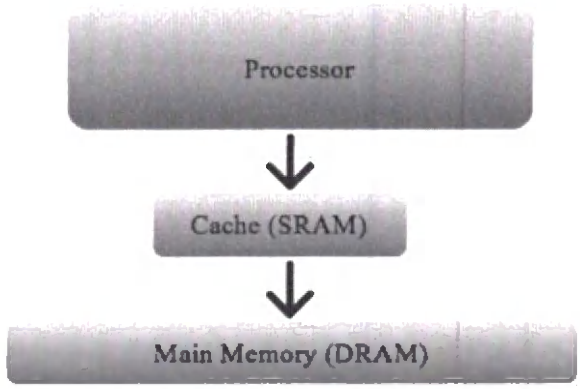

Figure 14. Cache Memory Configuration

Figure 14 shows a sample cache memory configuration consists of processor. DRAM, and SRAM.

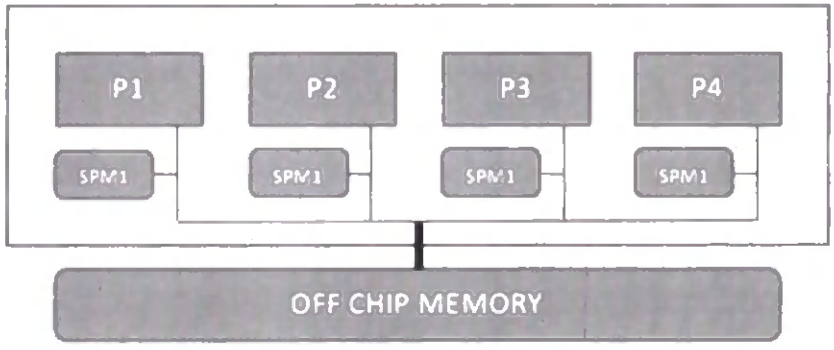

Figure 15. Sample SPM Architecture

SPM is a type of SRAM. SPMs are on-chip memories, and the content of the on-chip SPM memories can be edited and updated if needed through internal bus connection. Processors have access to all SPMs and off-chip memories in the SPM architecture given in Figure 15.

In static allocation scheme, data can not be modified, changed, or moved during the execution after its placed into SPMs. Static allocation can be used effectively for relatively basic applications, which do not require the code to be modified during the execution.

In dynamic data allocation technique, data blocks in the SPMs can be changed based on the application process. This dynamic approach helps complex applications to increase performance significantly.

In this thesis work, we will use Matrix Model Computation (MMC) as a dynamic data allocation method. So, MMC is used as an advance data allocation, and refactoring algorithm to increase performance.

4.4.1. Static Data Allocation

The memory is allocated before the data execution in static data allocation. So, the data can not be changed during the execution time. The data is generally placed into SPMs in the compile-time in embedded systems.

The static content in SPM blocks can not be edited, moved, or changed while the execution process occurs. There are many static data allocation schemes, which program data blocks are analyzed and partitioned into memory blocks including on-chip SPM, and off-chip memories [2].

Once the static allocation is complete, and program execution process starts; partitioned data is stable in the system. So, whatever is written into memory blocks

consumption. Some algorithms are based on accessing the data arrays from the loop operations [1], [60].

4.4.3. Ultra-Dynamic Allocation

The MMC application is proposed to resolve the allocation issue in data intensive applications in this thesis. The dynamic allocation has proven to be advantageous over the static allocation schemes. For example, if we consider a program block with five successive loops; each loop makes considerable changes in separate variable at the end of loop cycles. We prefer to keep only one variable at the SRAM, rest of the data is kept in DRAM. In static allocation, we can only keep one constant variable during the five successive loop period. However, in dynamic allocation, we have option to replace the SRAM space with the significant variable at the end of each clock cycle. So, we can keep all five variables in one SRAM space. This example clearly shows the significant performance improvement in dynamic allocation scheme.

4.5. Loop Fission

Loop fission is also called as loop distribution. The main idea is to divide the nested loop into separate loop pieces. The successive loop blocks are equivalent to the first given single loop block. Especially in data-intensive applications, loop

Computer architecture designers should consider using either loop splitting or loop fission. depending on the behavior of the application in embedded systems. Some complex and advanced embedded SPM architectures may use multiple data allocation algorithms.

4.7. Matrix Model Computation in Loop Fission

Once we carefully analyze both loop fission and loop splitting. We decided that the matrix model computed data is best applicable to loop fission application.

As seen on Figure 10. We have three separated matrices as a result of matrix model computation. Assuming that we have all our data set in a loop. we can convert one loop into three parts using loop splitting.

As a result. loop 1 will have the following data set;

$p = c1 - c2$

$q = p + c3$

$r = q / c4$

$s = r + q$

loop 2 will have the following data set:

$u = c1 * c4$

$v = c3 / c4$

And rest of the following values will be placed in loop3;

$w = u * v$

$x = u - w$

5. SYSTEM ARCHITCTURE AND EXAMPLE

SPMs are proven with many advantages such as less area requirement, low energy consumption, reduced cost, more precise predictability, reliability, and better performance. SPMs gathered attention of embedded architecture designers because of all these advantages. SPMs, as software managed SRAM memories, guarantee significantly faster access times. SPMs are very preferable for achieving reduced execution time especially in data-intensive applications. We assume to apply MMC re-computation as a dynamic data allocation which is called ultra-dynamic allocation in this thesis work. The SPMs are chosen as main memory components in this thesis work.

We will analyze both single core structures and multi-core architectures.

5.1. Single Core SPM Architectures

SPMs have a big disadvantage over hardware controlled counterparts, limited space. So, an off-chip memory is a requirement in most cases to hold the entire program in memory. However, we have an on-chip SPM and a software controlled on-chip cache to provide faster access times compared to off-chip memories. We assume that our sample code block is a data intensive application which can be placed into SPM memories.

The SPMs are software-managed memories. Software is responsible to move the data blocks in between memory components and to keep the data location.

processor as a multicore SPM architecture. Each core will have its local SPM and each core will access a shared L2 memory. The cores can also access other core's SPMs.

Figure 19. Shows an example of multi core SPM architecture that we use in this thesis work. All data in SPM is also available in L2 memory. The whole program data is also available in off-chip memories. SPMs are designed to avoid costly off-chip memory accesses. SPMs are software managed, all data should be divided into blocks and the frequently used data blocks should be placed into SPMs accordingly to increase overall performance. So, a core will access the local SPM, neighboring SPMs, L2 memory, and off-chip memory.

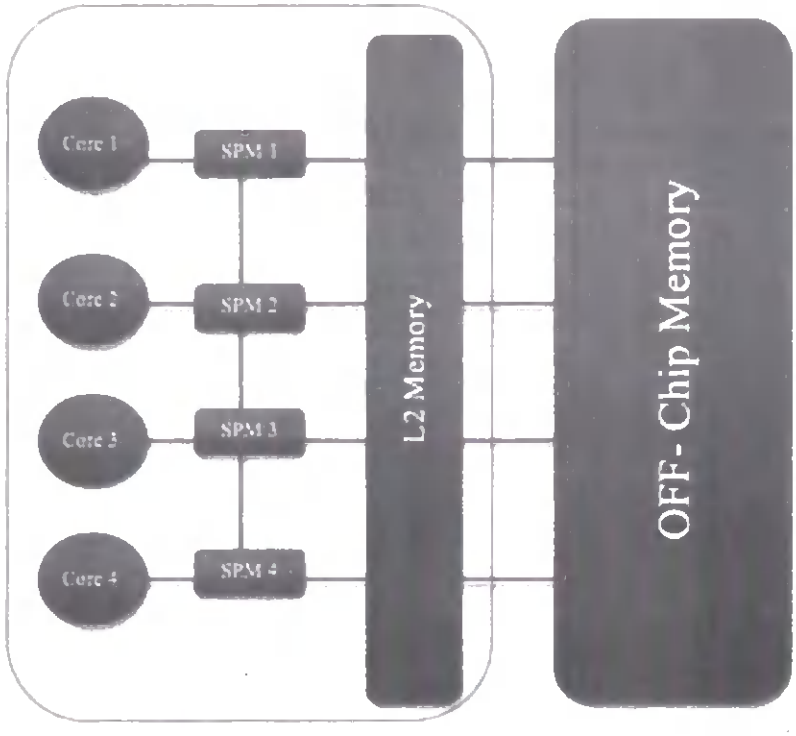

Figure 19. An Example of Multi Core SPM Architecture

```
u[i] = c1 * c4;
q[i] = p[i] + c3;
t[i] = r[i]- s[i];
w[i] = u[i] * v[i]; }
```

This example code has many arithmetic operations. The whole code is supposed to be in a loop. The code can be classified as a data intensive code in this example. We assume that addition and subtraction operations take 1 clock cycle, and division and multiplication operations take 4 clock cycles. We define that the cost of reading and writing data from local SPM is 2 clock cycles, access latency for remote SPM is 5 clock cycles, L2 cache access latency is 10 clock cycles, and off-chip memory access latency is 100 clock cycles.

Access Type	Pi – SPMi	Pi - SPMj	Pi – L2 Cache	Pi – Off-chip DRAM
Latency (cycle)	2	5	10	100

Table 1. Memory Access Latencies

We assumed that our configuration contains one processor for single-core, four processors for multi-core architecture. Each processor has two local SPM blocks in 16KB each. In addition to that, we have 10 blocks of L2 on-chip memory. Our on-chip L2 memory size is 80 KB. We assume that our on-chip cache memory is

OP	Operations	p	q	r	s	t	u	v	w	x	C1	C2	C3	C4
-	p= c1-c2	P	-	-	-	-	-	-	-	-	a3	a3	-	-
+	q= p+c3	a7	Q	-	-	-	-	-	-	-	-	-	a7	-
/	r=q/c4	-	a2	R	-	-	-	-	-	-	-	-	-	a2
+	s=p+q	a5	a5	-	S	-	-	-	-	-	-	-	-	-
-	t=s – c1	-	-	-	a8	T	-	-	-	-	a8	-	-	-
*	u=t *c4	-	-	-	-	a6	U	-	-	-	-	-	-	a6
/	v=s/u	-	-	-	a1	-	a1	V	-	-	-	-	-	-
*	w=c3*v	-	-	-	-	-	-	a9	W	-	-	-	a9	-
-	x= v-w	-	-	-	-	-	-	a4	a4	X	-	-	-	-

Figure 20. Phase Partitioning in cMMC

Figure 20 shows that recomputed data is divided into three forms. So we can have an idea of how we can place the data into the SPM blocks. We will have following loop blocks:

```
for ( i=0; i<100; i++) {
p[i] = c1 – c2;
q[i] = p[i] + c3;
r[i] =  q[i]  / c4;
s[i] = r[i] + q[i];
}
for ( i=0; i<100; i++) {
t[i] = r[i]- s[i];
u[i] = c1 * c4;
```

6. EXPERIMENTAL RESULTS

In this section of thesis work, we compare the static, dynamic, and ultra-dynamic approaches of both single core and multi-core embedded architectures. We used eight data intensive benchmarks. The benchmarks are taken from Perfect Club, Spec, and Livermore benchmark suites. They are written in C++ language and they consist of loop nests, which include data arrays with sizes from one to three dimensions. The data sizes of these benchmarks range from 120KB to 12045KB.

Bench. Name	Source	Data Size (KB)	Static (ClockCycle)	Dynamic (ClockCycl)	Ultra-dynamic
MXM	Spec	120	27.452.600	19.452.700	16.201.300
WSS	Spec	122.8	27.824.460	22.936.346	18.168.346
BMCM	Perfect Club	123.2	29.210.400	23,284.196	18.516.196
ADI	Livermore	241.2	24.864.900	22.570.100	16.856.990
TOMCATV	Spec	280.8	23.429.703	10.367.707	7.695.707
EFLUX	Perfect Club	285.6	16.547.609	11.708.145	6.596.721
VPENTA	Spec	600	27.584.420	22.828.518	14.504.518
APSI	Spec	12045.2	1.272.871.200	875.151.494	818.911.494

Table 2. The benchmarks used for experimental evaluation and performance improvements for single-core architecture.

Table 2 shows only single core architecture results. First column of table 2 represents the benchmarks used in this experiment. The second column shows the

We assumed to have two SPM blocks and 10 blocks of L2. Each block has 8KB of capacity in on-board L2 memory. Therefore, each processor has its 32KB local SPM and we have 80KB L2 on-chip memory. Using two blocks of SPM will help us to better place relatively larger code blocks into SPMs. Thus, TOMCATV benchmark has the highest gain (67.15%) when ultra-dynamic allocation is compared to static allocation. In addition, EFLUX has the highest gain when ultra-dynamic allocation is compared to dynamic allocation. Because EFLUX code can be applied into loop fission process and we can generate multiple code blocks easily with minimal data dependency in between the divided code blocks.

Bench. Name	Source	Data Size(KB)	Static (Clock Cycles)	Dynamic (Clock Cycles)	Ultra-dynamic
MXM	Spec	120	8.098,100	6.098,200	4,073,800
WSS	Spec	122.8	8.203,875	5.782,361	5,382,361
BMCM	Perfect Club	123.2	8,585,550	5,844,146	5,444.146
ADI	Livermore	241.2	7.539,900	6.745,190	4,592,990
TOMCATV	Spec	280.8	7.280,628	3.692.574	2.733.574
EFLUX	Perfect Club	285.6	4.341,029	3.020,625	1,744.401
VPENTA	Spec	600	8,749,550	7.564,648	4,721,648
APSI	Spec	12045.2	320.371,200	221,636,494	221.636,494

Table 4. The benchmarks used for experimental evaluation and performance improvements for multi-core architecture.

In Multi-Core architectures, we have the average performance gain of 44.86% for Gain 1, which represents the performance gain for ultra-dynamic allocation over static allocation. We also have an average performance gain of 23.08% for Gain 2, which stands for ultra-dynamic allocation over dynamic allocation. As a result, ultra-

7. CONCLUSION AND FUTURE WORK

In this thesis work, we presented an approach to partition a data-intensive embedded application into execution phases in a more effective way. Current dynamic memory management schemes typically consider a loop nest in such applications as one execution phase. Considering the fact that dynamic management schemes for software-managed memories are better than the static ones where the entire application is considered as one execution phase, our approach investigates the possibility of dividing a loop nest into multiple execution phases using various loop transformation techniques in order to improve the performance of embedded systems. The proposed approach, called as ultra-dynamic memory management in this thesis, first aims at partitioning loop nests into multiple execution phases using matrix model computation technique. Then, the data blocks are mapped to on-chip SPMs and off-chip memory based on their access frequencies within each execution phase considering the parameters such as the number of cores and available on-chip memory size. By doing so, our goal is to reduce the number of off-chip memory accesses; hence, improving performance.

Our experimental results conducted using several data-intensive benchmarks clearly show the viability of the proposed approach in single and multi-core embedded architectures. Our technique improves the performance around 24% on the average, over the dynamic memory management scheme.

8. REFERENCES

1. M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A Parikh, "Dynamic management of scratch-pad memory space," *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 690-695.

2. P. R. Panda, N. D. Dutt, A. Nicolau. "Efficient utilization of scratch-pad memory in embedded processor applications," *Proceedings of the 1997 European conference on Design and Test. 7*, 1997.

3. S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems." *In proceedings of the international conference oncompilers, architecture and synthesis for embedded systems*, 2003.

4. R. Banakar, S. Steinke, B. Lee, M. Balakrishnan and P. Marwedel. "Scratchpad memory: design alternative for cache on-chip memory in embedded systems." *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002. pp. 73-78.

5. A. Janapsatya, S. Parameswaran, and A. Ignjatovic. "Hardware/software managed scratchpad memory for embedded system." *Proceedings of the IEEE/ACM International conference on Computer-aided design*, 2004. pp.370-377.

6. H. Koc, E. Ercanli, M. Kandemir, and S. W. Son, "Compiler-directed temporary array elimination." *In Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems*, 2006.

7. B. Egger, S. Kim, C. Jang, J. Lee, S. Min, and H. Shin, "Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU," *IEEE Transactions on Computers, 59(8):1047–1062*, 2010

8. R. Banakar, S. Steinke, B. S. Lee, M. Balakrishnan, and P. Marwedel. "Comparison of cache-and scratch pad based memory systems with respect to performance, area and energy consumption," *University of Dortmund, Technical Report No. 762*, 2001.

21. Y. Guo, Q. Zhuge, J. Hu, M. Qiu, and E.-M. Sha, "Optimal data allocation for scratch-pad memory on embedded multi-core systems," *In Parallel Processing (ICPP), 2011 International Conference*, 2011, pp. 464–471.

22. H. Tajik, B. Donyanavard, J. Jahn, J. Henkel,and N. Dutt, "SPMPool: Runtime SPM Management for Embedded Many-Cores." *CECS TR 14-08*, 2014.

23. N Nguyen, A Dominguez, R Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," *ACM Transactions on Embedded Computing Systems (TECS) 8 (3)*, 2005.

24. M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad sharing strategies for multiprocess embedded systems: A first approach." *In Proceedings of theWorkshop on Embedded Systems for Real-Time Multimedia*, 2005.

25. F. Angiolini, L. Benini, and A. Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning." *In Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2003.

26. D. P. Volpato, A. K. Mendonca, L. C. dos Santos, and J. L. Guntzel, "A post-compiling approach that exploits code granularity in scratchpads to improve energy efficiency," *2010 IEEE Computer Society Annual Symposium on VLSI*, 2010, pp. 127–132.

27. Y. Yang, Z. Shao, L. Pan, M. Guo, "ISOS: Space Overlapping Based on Iteration Access Patterns for Dynamic Scratch-pad Memory Management in Embedded Systems." *Accepted in The 9th International Conference for Young Computer Scientists*, 2008.

28. Ben Juurlink . Pepijn de Langen, "Dynamic techniques to reduce memory traffic in embedded systems." *published in preceding CF '04 proceedings of the first conference on computing frontier ACM*, 2004.

29. M. Kandemir,J. Ramanujam, A. Choudhary, "Exploiting shared scratch pad memory space in embedded multiprocessor systems," *Published in: Design Automation Conference Proceedings 39th*, 2002.

30. H. Koc, O. Ozturk, M. Kandemir, S. H. K. Narayanan, and E. Ercanli, "Minimizing energy consumption of banked memories using data re-computation," *In Proceedings of International Symposium on Low Power Electronics and Design (ISLPED'06)*, 2006. pp.358-362.

42. F. Angiolini, L. Benini, and A. Caprara, "An efficient profile based algorithm for scratchpad memory partitioning," *IEEE Trans. CAD, vol. 24, no. 11*, 2005, pp. 1660-1676.

43. J. Yan and W. Zhang, "Accurately estimating worst-case execution time for multi-core processors with shared direct mapped instruction caches," *in Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '09). Beijing, China*, 2009, pp. 455-463.

44. S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel, "Reducing Energy Consumption by Dynamic Copying of Instructions Onto Onchip Memory," *Proc. 15th Int'l Symp. System Synthesis (ISSS '02)*, 2002, pp. 213-218.

45. M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu, "Compiler-directed scratch pad memory optimization for embedded multiprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 12, no. 3.*, 2004, pp. 281-287.

46. A. Janapsatya, A. Ignjatović, S. Parameswaran, "A novel instruction scratchpad memory optimization method based on concomitance metric," *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, 2006.

47. M. Verma, L. Wehmeyer, and P. Marwedel, "Cache-Aware Scratchpad Allocation Algorithm," *Proc. Int'l Conf. Design, Automation and Test in Europe (DATE)*, 2004.

48. V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Energy-Oriented Compiler Optimizations for Partitioned Memory Architectures," *In: CASES '00: Proceedings of the 2000 International Conference on Compilers*, 2000.

49. L. Li, L. Gao, and J. Xue, "Memory coloring: a compiler approach for scratchpad memory management," *in Proceedings of 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05). St. Louis, MO*, 2005, pp. 329-338.

50. S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratchpad memory using compile time decisions," *ACM Transactions on Embedded Computing Systems (TECS), vol. 5, no. 2*, 2006, pp. 472-511.

61. O. Avissar, R. Barua, and D. Stewart, "Heterogeneous memory management for embedded systems, " *in proceedings of the ACM 2nd international conference on compilers, architectures, and synthesis for embedded systems*,2001.

62. H. Sayadi, H. Farbeh, A. M. H. Monazzah, and S. G. Miremadi, "A Data Recomputation Approach for Reliability Improvement of Scratchpad Memory in Embedded Systems," *In Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'14)*. 2014.

63. H. Farbeh, M. Fazeli, F. Khosravi, and S. G. Miremadi, "Memory Mapped SPM: Protecting Instruction Scratchpad memory in Embedded Systems against Soft Errors," *to Appear in Proceedings of the 9th EDCC Conference*, 2012.

64. Y. Liu and W. Zhang, "Scratchpad Memory Architectures and Allocation Algorithms for hard real-time multicore processor," *Journal of computing science and engineering*. 2015.

65. S. Udayakumaran and R. Barua, "Dynamic memory allocation for scratch-pad memory using compile-time decisions," *In proceedings of the international conference on compilers, architecture and synthesis for embedded systems*. 2003.

66. Yemliha, Taylan, "Performance and Memory Space Optimizations for Embedded Systems," *Electrical Engineering and Computer. Science - Dissertations*. 2011.

67. Y. Guo, Q. Zhuge, J. Hu, J. Yi, M. Qiu, and E. H.-M. Sha, "Data placement and duplication for embedded multicore systems with scratch pad memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2013.

68. C. Huneycutt and K. Mackenzie, "Software caching using dynamic binary rewriting for embedded devices," *In proceedings of the international conference on parallelprocessing*. 2002, pp.621-630.

69. F. Balasa, I. I. Luican, and C. V. Gingu, "Scratch-pad memory banking for energy reduction in embedded signal processing systems," *Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium*. 2013, pp. 844 - 847.

70. A. Kannan, A. Shrivastava, A. Pabalkar, and J.-E. Lee, "A software solution for dynamic stack management on scratch pad memory," *In*

81. Y. Liu and W. Zhang, "Exploiting multi-level scratchpad memories for time-predictable multicore computing." *in Proceedings of the IEEE 30th International Conference on Computer Design (ICCD). Montreal, Canada.* 2012, pp. 61-66.

82. P. Panda, D. Dutt, and A. Nicolau, "On-chip vs. offchip memory: the data partitioning problem in embedded processor-based systems." *ACM Trans. Des. Autom. Electron. Syst., vol. 5, no. 3.* 2000, pp. 682–704

83. Y. Guo, Q. Zhuge, J. Hu, and E.-M. Sha, "Optimal data placement for memory architectures with scratch-pad memories." *in Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th Inter-national Conference.* 2011.

84. Olivier Temam, Elana D. Granstont William Jalby. "To Copy or not to Copy: A Compile- Time Technique for Assessing When Data Copying should be Used to Eliminate Cache Conflicts" *published in conference 1993 ACM 0-8186-4340-4/93/0011.*