2011

Performance and Memory Space Optimizations for Embedded Systems

Taylan Yemliha
Syracuse University

Abstract

Embedded systems have three common principles: real-time performance, low power consumption, and low price (limited hardware). Embedded computers use chip multiprocessors (CMPs) to meet these expectations. However, one of the major problems is lack of efficient software support for CMPs; in particular, automated code parallelizers are needed.

The aim of this study is to explore various ways to increase performance, as well as reducing resource usage and energy consumption for embedded systems. We use code restructuring, loop scheduling, data transformation, code and data placement, and scratch-pad memory (SPM) management as our tools in different embedded system scenarios. The majority of our work is focused on loop scheduling. Main contributions of our work are:

We propose a memory saving strategy that exploits the value locality in array data by storing arrays in a compressed form. Based on the compressed forms of the input arrays, our approach automatically determines the compressed forms of the output arrays and also automatically restructures the code.

We propose and evaluate a compiler-directed code scheduling scheme, which considers both parallelism and data locality. It analyzes the code using a *locality-parallelism graph* representation, and assigns the nodes of this graph to processors. We also introduce an Integer Linear Programming based formulation of the scheduling problem.

We propose a compiler-based SPM conscious loop scheduling strategy for array/loop based embedded applications. The method is to distribute loop iterations across parallel processors in an SPM-conscious manner. The compiler identifies potential SPM hits and misses, and distributes loop iterations such that the processors have close execution times.

We present an SPM management technique using Markov chain based data access prediction for irregular accesses.

We propose a compiler directed integrated code and data placement scheme for 2-D mesh based CMP architectures. Using a Code-Data Affinity Graph (CDAG) to represent the relationship between loop iterations and array data, it assigns the sets of loop iterations to processing cores and sets of data blocks to on-chip memories.

We present a memory bank aware dynamic loop scheduling scheme for array-intensive applications. The goal is to minimize the number of memory banks needed for executing the group of loop iterations.

PERFORMANCE AND MEMORY SPACE OPTIMIZATIONS FOR EMBEDDED SYSTEMS

By

Taylan Yemliha
B.S. Bogazici University, 1993
M.S. Marmara University, 1996

DISSERTATION

Submitted in partial fulfillment of the requirements for the
degree of Doctor of Philosophy in Computer and Information Science
in the Graduate School of Syracuse University

May 2011

Professor Mahmut T. Kandemir

Professor Kishan Mehrotra

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded computers are widely used, common usage areas ranging from cell phones to brake systems in high-end automobiles. General-purpose processors are designed to work well in various situations. While embedded processors must also have a certain level of flexibility, they are often customized for a particular application. Customization may be expensive, but the large number of embedded computers sold justify that cost in many cases. Because of this, some of the design guidelines that are commonly followed in general-purpose computer design may not be used for embedded computers.

In general, embedded systems have three common principles. First, they need to provide real-time performance; since embedded computers are used in important, even critical tasks. Second, their power/energy consumption should be low; hence preventing heating problems and increasing battery life. Third, they should be cheap; in most cases, embedded computers cannot have a lot of hardware. This is true not only due to cost issues, but also physical space limitations.

In accordance with these principles, we want an embedded system to perform well on limited hardware, while consuming as little power as possible. It is important that the system has a good performance, but it is not the only metric that is important. Therefore, working solely on enhancing the performance may not be a good idea if it increases the power consumption considerably, since it may cause heating, and drain the battery.

1.1 Our Goal

The aim of this study is to explore various ways to enhance embedded systems' performance and reduce the use of resources. We use code restructuring, loop scheduling, data transformation, code and data place-

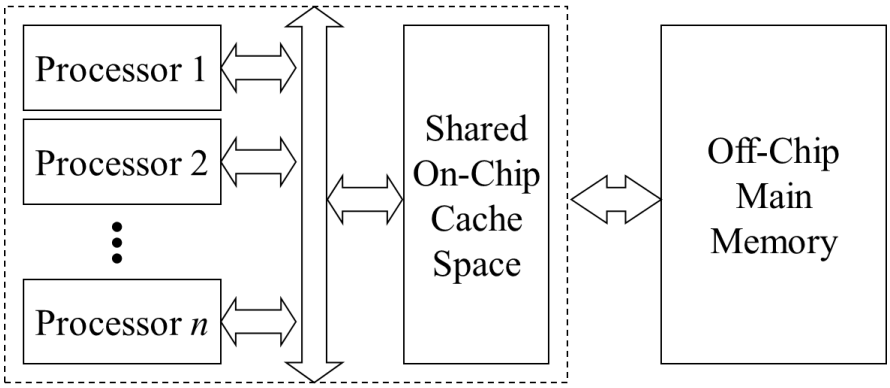


Figure 1.1: A CMP architecture with a shared on-chip cache.

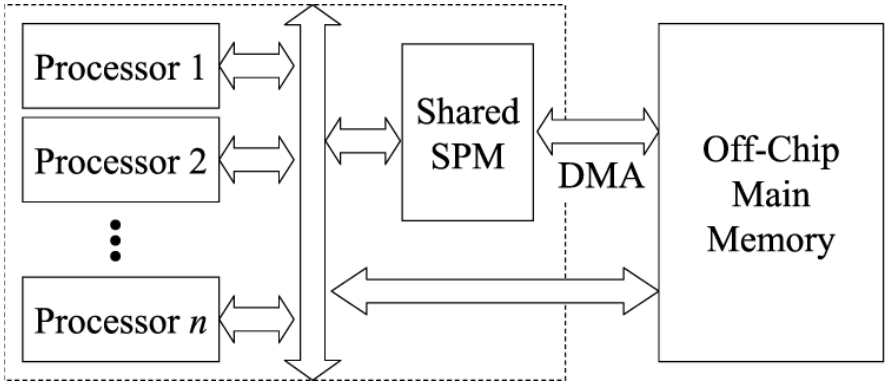


Figure 1.2: A CMP architecture with a shared on-chip SPM.

ment, and SPM management as our tools in different embedded system scenarios, while seeking to increase performance, improve memory access and reduce energy consumption.

While some of our methods make no assumptions about the hardware of the system; as there are different embedded system architectures, we also aimed at optimization utilizing particular hardware components, and hence assumed the existence of particular architectures in some of our work.

1.2 Target System Architectures

1.2.1 Chip Multiprocessors

As transistor sizes continue to shrink and the number of transistors per chip keeps increasing, chip multiprocessors (CMPs) are becoming a promising alternative to remain on the current performance trajectory for both high-end systems and embedded systems. The best way to meet the results expected from embedded

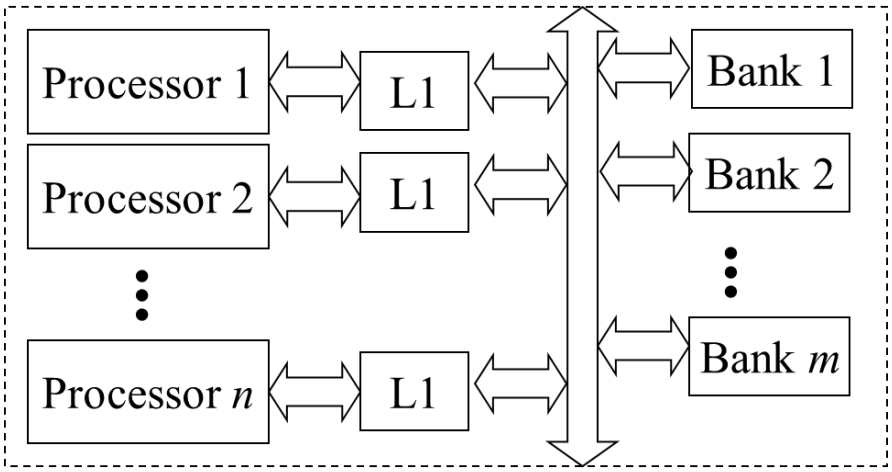


Figure 1.3: A CMP architecture with on-chip banked memory.

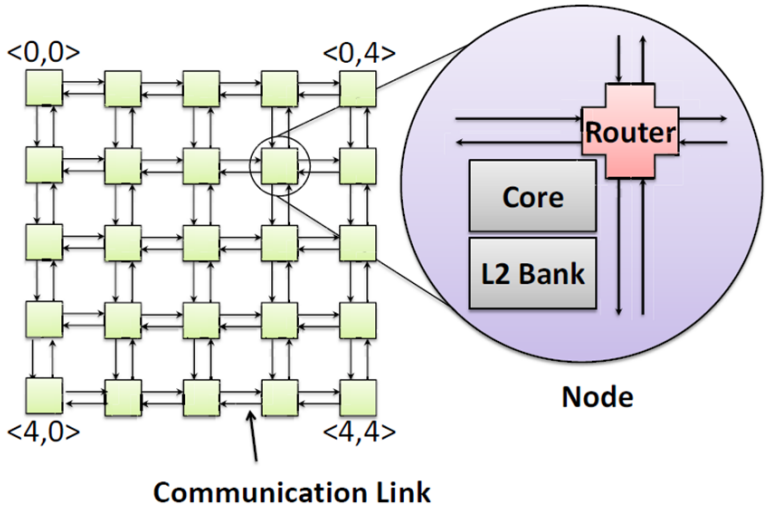


Figure 1.4: A Network-On-Chip architecture.

computers is by using multi-processors. This is particularly true when we must meet real-time constraints and are concerned with power consumption. Today, embedded computers are organized into multiprocessors.

The most recent advances in microprocessor design involve putting multiple processors on a single die (computer chip). These designs are known as Chip Multiprocessors because they allow for single chip multi-processing. These designs are popularly called multicore (each processor is called a core) and are completely replacing the traditional single core designs.

Recently, the CMP has become the preferred method of improving overall system performance. This is a departure from the approach of increasing the clock frequency or processor speed to achieve gains in overall system performance. Increasing the clock frequency has started to hit its limits in terms of cost-

effectiveness. Higher frequency requires more power, making it harder and more expensive to cool the system. This also affects sizing and packaging considerations. So, instead of trying to make the processor faster to gain performance, the response is now just to add more processors. As chip capacity increased, placing multiple processors on a single chip became practical. Major CPU developers have started producing CMP processors. Each computer chip manufacturer is trying to increase the number of cores that can be placed on a single chip economically.

From a logical point of view, there is no real significant difference between programming for multiple processors in separate packages and programming for multiple processors contained on a single chip. There may be performance differences, however, because the new CMPs are using advances in bus architectures and connections between processors. In some circumstances, this may cause an application that was originally written for multiple processors to run faster when executed on a CMP. Aside from the potential performance gains, the design and implementation are very similar.

One of the major factors that can potentially slow down widespread use of embedded chip multiprocessors is lack of efficient software support. The primary problem is that regular desktop software has not been designed to take advantage of the new CMP architectures. In fact, to see any real speedup from the new CMP architectures, desktop software will have to be redesigned. In particular, automated code parallelizers are badly needed, since it is not realistic to expect an average programmer to parallelize a large complex embedded application over multiple processors, taking into account several factors at the same time such as code density, data locality, performance, power and code resilience. As chip multiprocessors proliferate, programming support for these devices is likely to receive a lot of attention in the near future.

Parallelism and multiprocessing come at a cost. In some cases, introducing the overhead of parallel programming techniques into a piece of software can decrease its performance. Not every software application is suitable for multiprocessing or multithreading. The kind of software we are focused on is based heavily on loops handling arrays.

Mesh Processors

A mesh is a network of processors in which every node is connected to all of its neighbors. We can build meshes in different dimensions, including dimensions larger than three. A mesh network is scalable in that a network of dimension $n + 1$ includes subnetworks that are meshes of dimension n.

A mesh network balances connectivity with link cost. All links are fairly short, but a mesh provides a rich set of connections and multiple paths for data. The shortest path between two nodes in a mesh network is equal to its Manhattan distance, which in general is the sum of the differences between the indexes of the

source and destination nodes.

Since future technologies offer the promise of being able to integrate billions of transistors on a chip, the prospects of having hundreds to thousands of processors on a single chip along with an underlying memory hierarchy and an interconnection system is entirely feasible. This would mean mesh computing will be a mainstream computing architecture.

Figure 1.4 shows the high level architecture of a Network-On-Chip system. We focus on this architecture in Chapter 7.

1.2.2 Memory

In our work, we target CMP architectures, with various memory models, including shared cache, shared SPM, and on-chip banked memory.

Memory system utilization is an important issue for many embedded systems that operate under tight memory limitations. This is a strong motivation for recent research on reducing the number of banks required during execution of a given application. Reducing memory space requirements of an application can bring three potential benefits.

First, if we are to design a customized memory system for a given embedded application, reducing its memory requirements can cut the overall cost. Second, if we are to execute our application in a multi-programmed environment, the saved memory space can be used by other applications, thereby increasing the degree of multi-programming. Third, it is also possible to reduce the energy consumption in a banked memory system by reducing the amount of memory space occupied by application data and placing the unused banks into low-power operating modes.

Figure 1.4 shows the high level architecture of a CMP system with on-chip banked memory. We focus on this architecture in Chapter 6.

Cache Memory

Memory behavior plays a major role in both performance and energy consumption. Cache is memory placed between the processor and main system memory (RAM). Cache is faster than RAM, but does not have the capacity of main memory. A cache is designed to move a relatively small amount of data close to the processor. Caches use hardwired algorithms to manage the cache contents; hardware determines when values are added or removed from the cache. Cache increases the effective memory transfer rates and, therefore, overall processor performance. Cache is used to contain copies of recently used data or instruction by the

processor. Small chunks of memory are fetched from main memory and stored in cache in anticipation that they will be needed by the processor.

One of the primary functions of cache is to take advantage of the temporal and spatial locality characteristics that programs tend to exhibit. Temporal locality is the tendency to reuse recently accessed instructions or data. Spatial locality is the tendency to access instructions or data that are physically close to items that were most recently accessed.

Cache is often divided into two levels: Level 1 and Level 2. Level 1 cache is small in size sometimes as small as 16K. L1 cache is usually located inside the processor and is used to capture the most recently used bytes of instruction or data. Level 2 cache is bigger and slower than L1 cache. Currently, it is stored on the motherboard (outside the processor), but this is slowly changing. L2 cache is currently measured in megabytes. L2 cache can hold an even bigger chunk of the most recently used instruction, data, and items that are in the near vicinity than L1 holds. Because L1 and L2 are faster than general-purpose RAM, the more correct the guesses of what the program is going to do next are, the better the overall system performance because the right chunks of data will be located in either L1 or L2 cache.

Figure 1.1, illustrates a simplified view of a shared memory based architecture. In this architecture, multiple CPUs share an on-chip cache space. We also assume the existence of a large off-chip memory space, shared by all processors in the system. We focus on this architecture in Chapter 3.

Scratch Pad Memory

Scratch-pad memory (SPM), is a small, high-speed on chip data memory (SRAM) that is physically addressed but mapped into the virtual address space. Along with traditional memory hierarchy consisting of cache levels and main memory found in everyday systems, embedded systems increasingly make use of SPMs. Leveraging the power of SPMs is crucial to extract maximum performance from application programs.

Figure 1.2 shows the high level architecture of a system with an SPM that is shared by all processors. We focus on this architecture in Chapter 4 and Chapter 5.

Physically, SPM is within the same chip as the processors so that it can be accessed much faster than the off-chip main memory. Logically, however, SPM is within the same address space as the off-chip main memory; therefore, both the SPM and the main memory accesses use the same memory access (load/store) instructions. Many systems (e.g., [51]) also provide a DMA channel to speed up data transmission between the SPM and the main memory.

SPM differs from a conventional cache in that it is explicitly managed by software, while cache is implicitly managed by hardware. SPM's access time is predictable, unlike accesses to a cache, and predictability

is a key attribute of SPMs. The advantages of on-chip scratch-pad memory over a conventional hardware managed on-chip cache is twofold. Firstly, references to a cache are subject to conflict, capacity and compulsory misses, while references to scratch-pad guarantee that they will result in a hit, as data movements are managed by software. Secondly, scratch-pads are accessed by direct addressing. This mitigates the overhead of expensive hardware cache tag comparison, typically present in set associative caches.

Like cache memory, the size of SPM is chosen to fit on-chip and provide high-speed access. Due to its limited size, SPM usually cannot hold all the data accessed by the application. In order to take the most advantage of the SPM, one must be careful in SPM management (i.e. determining which data should be stored in the SPM at different points of the application code). While a programmer can select a set of frequently-accessed data and manually write code to load them into the SPM, a more desirable approach is to employ a compiler that effectively analyzes the data access patterns exhibited by the application code and identifies the frequently reused data. Many compiler-based strategies have been proposed by the previous efforts (e.g, [34, 56, 54]).

SPM can be managed using a combination of compile-time and runtime decision making. While regular accesses like scalar values and array expressions with affine subscript functions have been tractable for compiler analysis (to be prefetched into SPM), irregular accesses like pointer accesses and indexed array accesses are not easily amenable for compiler analysis. We address this issue in Chapter 5.

1.3 Software for CMPs

Chip multiprocessors are a promising candidate for a billion-transistor embedded computing era. While there have already been several different chip multiprocessor architectures from both academia and industry, necessary software support for extracting maximum performance from these architectures is still in their infancy. This is unfortunate because unless we are able to parallelize a given application code effectively across processors, there is little benefit to be gained from the parallelism provided by CMPs. In this context, compiler support is particularly crucial, since it is not realistic to expect an average programmer to parallelize a large complex embedded application manually.

1.3.1 Loop Scheduling

In a parallel system with multiple CPUs, one of the key problems is to assign loop iterations to processors. This process is called loop scheduling, and has been studied in the past for various types of parallel architectures. The majority of our work is also focused on loop scheduling.

Previously published loop scheduling techniques can be roughly divided into static and dynamic techniques. Static scheduling techniques try to perform the loop iteration-to-processor assignment at compile time, before the application starts executing, whereas dynamic techniques postpone this assignment to runtime. As compared to static loop scheduling, the main advantage of dynamic scheduling is the ability of capturing the variations across the workloads of different CPUs, and exploiting this information when performing iteration assignment at runtime. These dynamic variations can occur due to different reasons such as conditional flow of control and cache behavior. On the other hand, the main advantage of static scheduling over dynamic scheduling is that, it doesn't require any system resources at runtime.

1.3.2 Our Target Software

Since we use loop scheduling, we are targeting the type of software that consists mainly of loops manipulating arrays. Array/loop based codes constitute an important class of embedded applications, since most image/video processing applications are coded as a series of nested loops that operate multidimensional arrays. A straightforward parallelization of a loop by distributing its iterations across available processors evenly - where every processor are assigned a similar number of iterations - may not be the best option in many cases, though it is easy to automate within a compiler. This is because, in general different loop iterations can take different number of cycles to finish, due to conditional execution constructs (e.g., if-statements) within loop body and dynamic memory behavior (e.g., data locality). While most image/video codes do not really have too many conditional constructs in loop bodies, data locality behavior can be an important factor shaping the overall performance; therefore, both static and dynamic scheduling techniques use different methods to reduce variation and achieve load balancing.

1.4 Our Work

The rest of the chapters are organized as follows. In Chapter 2, we propose and evaluate a compiler-based memory saving strategy [126] that exploits the value locality in array data by storing arrays in a compressed format. Based on the compressed forms of the input arrays, our approach automatically determines the compressed forms of the intermediate and output arrays and also automatically restructures the application code to work with compressed arrays. Our experimental results show that this scheme reduced both the memory space requirements and the execution cycles of the applications tested.

In Chapter 3, we propose and evaluate a compiler-based code scheduling scheme [127], which considers both parallelism and data locality at the same time. Our approach uses a *locality-parallelism graph* (LPG) to

capture the parallelism and data reuse, and assigns the nodes of this graph (sets of loop iterations) to the processors. Our experimental results indicate that our approach improves overall execution latency significantly. We also introduce an ILP (Integer Linear Programming) formulation of the problem, and compare its results to our heuristic approach.

In Chapter 4 we propose and evaluate a compiler-based loop scheduling strategy [122] that distributes loop iterations across processors in an SPM-conscious manner. In this strategy, the compiler analyzes the loop, identifies the potential SPM hits and misses, and distributes loop iterations over processors such that the processors have similar execution times, taking into account the difference between access times for the SPM and main memory. Our experimental results indicate that the proposed approach brings a significant performance improvement.

In Chapter 5 we propose and evaluate an SPM management technique [129] using Markov chain based data access prediction for irregular (e.g., pointers, index arrays) accesses that have hidden data reuse. Our experimental results indicate that the proposed approach brings a significant performance improvement in a set of applications with both regular and irregular access patterns.

In Chapter 6 we present a memory bank aware dynamic loop scheduling scheme [58] that minimizes the number of memory banks that need to be used for executing a group of loop iterations. The goal of this approach is to minimize memory energy consumption in banked memory systems, which can be a significant portion of the overall energy consumption; and that is an important metric to consider during scheduling, especially in battery-operated embedded systems. Our approach considers the bank access patterns of loop iterations and assigns iteration sets to processors such that, if possible, the number of memory banks that are used at the current state is not increased. Our experimental results show that the proposed scheme leads to much better energy results when compared to prior techniques and is also competitive in performance.

In Chapter 7 we propose and evaluate a compiler directed integrated code and data placement scheme [128] for 2-D mesh based CMP architectures. Our approach uses a Code-Data Affinity Graph (CDAG) to represent the relationship between loop iterations and array data. It assigns the sets of loop iterations to processing cores and sets of data blocks to on-chip memories, taking into account the on-chip memory capacity and load imbalance across different cores as well as the topology of the NoC. Our experimental results show that our CDAG based placement scheme brings improvements in both performance and energy consumption.

We conclude with a summary and a discussion of future work in Chapter 8.

Chapter 2

Code Restructuring for Operating with Compressed Arrays

2.1 Introduction

In this chapter, we propose and evaluate a static array compression and code restructuring scheme, targeting CMP architectures. Our approach doesn't make any assumptions about the memory subsystem of the architecture.

Many embedded computing systems operate under tight memory constraints. As a consequence, programmers need to restructure the data access patterns of their applications and reorganize data in memory to make best use of the available memory space. Unfortunately, as embedded applications are becoming increasingly complex and processing ever-increasing datasets, this programmer-based approach to code/data restructuring may not be a viable option (in terms of scalability) in the near future. While automatic compiler support can be of help for this problem, many known compiler optimizations today have been developed for high-end computing systems (e.g., large scale parallel machines) with no memory constraints, and so, pay little attention to the memory space demands of the application code being compiled.

Therefore, the compilers targeting embedded systems with tight memory budgets should adopt novel compilation techniques to reduce memory space demand. Prior research has already investigated several such techniques, as will be discussed in detail in Section 2.2. One of the common characteristics of many of these techniques is that they trade-off performance with memory savings. In other words, memory space savings come with performance degradation and/or extra power consumption. While this can be tolerated to a certain

extent in some embedded systems, there also exist many execution environments for which such a trade-off may not be an option. Based on this discussion, it is clear that an automated optimization that reduces *both* memory space requirements and the number of execution cycles (and possibly power consumption) at the same time would be very useful in practice.

One of the unexplored directions is the possibility of working with compressed data. In many applications, input arrays are given as compressed and the programmer is left with the task of determining the compressed formats of intermediate and output data sets. Due to the lack of automated tools, this task is very demanding and error prone. Specifically, observing that many data arrays used in embedded applications exhibit a high degree of value locality* (i.e., most of the array elements have the same value), we propose a novel compilation technique that exploits this property.

Loop scheduling has been an important problem in compiler area since early times of parallel computing. Consequently, prior research considers both static and dynamic compilation techniques with the aim of balancing workloads of individual processors and improving runtime behavior. While some of the static techniques take into account variations among different loop iterations due to conditional constructs and cache behavior, most of current implementations distribute loop iterations equally across parallel processors.

Our proposal has two components. The first component is a compiler algorithm that determines the compressed formats of intermediate and output data sets, given the compressed formats of input data sets. The second component is a scheduling algorithm which performs loop iteration-to-processor assignment taking into account whether an iteration operates on compressed or uncompressed data; thereby generating a restructured version of the code that operates on these compressed arrays. Since the restructured version of the code takes advantage of value locality and skips certain computations that would normally be performed in the original code, we save execution cycles (and potentially power as well, though the latter is not explicitly quantified). A high level view of our approach is depicted in Figure 2.1.

We implemented our approach and made experiments with 4 array-based applications. In our experimental evaluation, we also compare our proposal to an alternate scheme where scheduling is done without considering the type of the data on which iterations operate. The experimental results collected so far are promising and show that our compiler-based approach reduces both execution cycles (about 14% on the average) and memory space demands (about 19% on the average). While data compression has been used in the past for reducing memory space requirements, communication/data transfer latencies, and energy con-

*Value locality in this context indicates the case where a significant fraction of the elements of a given array have the same value. For example, if 20% of the elements of a given array have value a and are clustered in a region of the array and 35% of the elements of the same array have value b and are clustered in another region, we say that the array exhibits value locality. Note that, in order to exploit value locality, our approach does not need to know the actual (numeric) values of a and b at compile time.

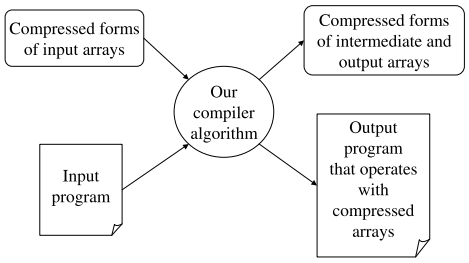


Figure 2.1: A high level view of our approach.

sumption, our work is unique in that it uses automated compiler support to determine both the compressed array formats and the restructured code that works with these compressed arrays.

The rest of this chapter is organized as follows. In Section 2.2, we discuss the relevant work on instruction and data compression. In Sections 2.3 and 2.4, we present the mathematical details of the proposed compression and parallelization approaches, respectively. In Section 2.5, we quantify the memory space and performance benefits of our approach. Section 2.6 concludes the chapter with a summary of our major findings.

2.2 Related Work

We discuss related work in two categories: compression and CMPs. Prior research considered compression techniques to reduce the memory footprint of both program code and application data. Most of the code compression techniques proposed in the literature implement an encoding scheme. A pattern-matching technique is described by Cooper and McIntosh [27]. They try to coalesce the instruction sequences by this pattern-matching technique to reduce the size of a given program code fragment. Code compression has been used within the context of VLIW architectures as well. For example, Ros and Sutton [106] apply code compression algorithms to instruction words in VLIW architectures. Lekatsas and Wolf [65] propose a runtime decompression unit to decompress the code on-the-fly before it gets executed. Arithmetic coding along with a Markov model has been employed in this scheme. Liao et al [72] concentrate on the applications from the DSP domain. In their approach, compressed data is expressed using a dictionary and a skeleton which can be executed by different methods. Along a similar direction, methods that use profile information have been proposed as well [120, 29]. Code compression has also been applied to VLIW architectures that use variable-to-fixed (V2F) coding [119, 112]. An extension to this approach, called the variable-sized-block method, is presented by Lin et al [74].

On the data compression side, the prior efforts include both hardware and software approaches. Hardware-assisted on-the fly data compression and decompression is proposed by Benini et al [13]. Specifically, they

propose an architecture where compressions and decompressions occur between the cache and the main memory. In this sense, the uncompressed cache lines are compressed on-the-fly before they are written back to the main memory. A similar approach is applied to VLIW processors by Macii et al [77]. They aggressively try to minimize energy consumption by using a differential technique on a hardware compression unit. Yang et al [124] reduce miss penalties and the number of cache misses by applying compression to a first level cache. Data compression has been investigated as a viable solution in the context of SPMs (scratch pad memories) as well. For example, Ozturk et al [96] propose a compression-based SPM management. They also formulate the data compression/decompression problem using ILP. The impact of using data compression is studied in [1] from a system performance angle. In [3], Ahn et al discuss effective algorithms for data compression and decompression. Data compression has also been used to decrease the energy consumption in banked main memories [98]. Ozturk and Kandemir [98] propose a compiler-assisted compression and migration technique to cluster data with similar access patterns in the same set of banks to exploit the low-power operating modes in banked-memories. The impact of data compression on different peripherals has also been investigated. Xu et al [121] present energy savings on a handheld device through data compression. Chen and Fowler [22] use data compression to efficiently manage sensor networks. MPSoC architectures are becoming popular for designing embedded systems. They gain ground in both academic environments and industry [38, 41, 11, 84, 86, 102]. Hammond et al [40] compared three alternative microarchitectures: MPSoC, SMT, and superscalar. They found the MPSoC architecture favorable over the others in both software and hardware trends. Olukotun and Hammond [93] studied conventional uniprocessor and MPSoC. They found MPSoCs to have major advantages over conventional uniprocessors in several aspects such as hardware design, performance, and power, and therefore concluded that the transition to MPSoCs is inevitable. In [88], Nayfeh et al explored the optimal ratio of processors to cache memory size in terms of cost/performance. They studied the trade-offs between cache size and number of processors in an MPSoC system, and showed that for parallel applications, clustering via shared caches provides an effective mechanism for increasing the total number of processors in a system without increasing the number of invalidations. Richardson [105] studied the design issues in the MPOC project, a chip multiprocessor for embedded systems. A four-stage pipeline and co-resident on-chip DRAM are found to improve performance. Wolf [117] reviewed several commercial MPSoC designs and identified the unique challenges, from hardware to software, for MPSoCs in embedded domains. Gomma et al [36] utilizes the extra on-chip parallelism for improving MPSoC's reliability against hardware transient errors. The proposed technique, CRTR (Chiplevel Redundantly Threaded multiprocessor with Recovery) achieves fault tolerance by executing and comparing two copies of a given application. Several techniques, such as asymmetric commit and Death- and Dependence-Based Checking Elision (DDBCE), are proposed for hiding inter-processor latency and checking overhead.

2.3 Working with Compressed Data

In this section, we present the mathematical details of our compression approach. First, in Section 2.3.1, we explain how we represent a compressed array within our compiler (i.e., the array abstraction to the compiler). Then, in Section 2.3.2, we discuss our compression algorithm in detail.

2.3.1 Compressed Array Format Abstraction

Many data arrays used in embedded applications exhibit a high degree of value locality and these data arrays are generally accessed/modified by nested loops. Our approach exploits this observation. When all the data elements in a rectilinear region of an array contains an identical value[†], we refer to such a region as a *uniform block*. Note that an array may have more than one uniform block. Essentially, if we know the size and location of the uniform blocks within an array, we can partition this array into a set of uniform and non-uniform blocks.[‡]

If the array references on the right hand side of an assignment statement within a loop nest have large uniform blocks, this means the same values are being accessed many times throughout the loop execution from different memory locations, i.e., a high degree of value locality. Furthermore, if the uniform blocks of these arrays have a non-empty intersection, then some of these values might be actually used in the same loop iteration.

In this work, we define a compressed array format to capture uniform and nonuniform blocks of a given data array in a concise manner. The proposed technique takes as input a set of input arrays in the compressed format (the arrays with no uniform block are represented in the conventional way). The compressed format captures the uniform blocks in the array in the following fashion:

$$A :: \{d : s_1 : s_2 : ... : s_d : u : \\ [x_{11}, x_{21}...x_{n1} : y_{11}, y_{21}...y_{n1} : v]; [...]; ...\},$$

where A is the name of the array, d is the number of dimensions of the array, s_1 through s_d are the size of each dimension, u is the number of uniform blocks within this array, $(x_{11}, x_{21}...x_{n1})$ and $(y_{11}, y_{21}...y_{n1})$

[†]In cases where approximate computation is applicable, we can handle regions that contain "similar" values as well, instead of "identical" values.

[‡]Our current implementation relies on programmer assistance to identify uniform blocks of input arrays. Note that, the more accurate the programmer is in specifying the uniform blocks of input arrays, the more memory savings our approach will achieve. Note also that the uniform blocks of the intermediate and output arrays are determined by our approach automatically; the programmer needs to specify the partitioning of input arrays only.

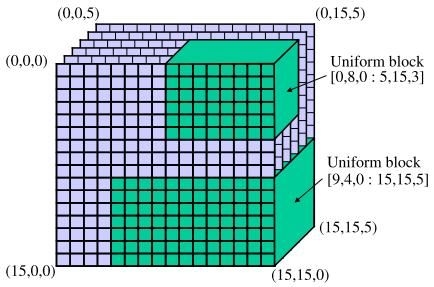


Figure 2.2: Format of compressed array $X :: \{3 : 16 : 16 : 6 : 2 : [0, 8, 0 : 5, 15, 3 : a]; [9, 4, 0 : 15, 15, 5 : b]\}$.

specify the location of each uniform block, and v gives the value of the this uniform block. Note that the array may be multidimensional. For example, an entry such as

$$X :: \{3 : 16 : 16 : 6 : 2 : \\ [0, 8, 0 : 5, 15, 3 : a]; [9, 4, 0 : 15, 15, 5 : b]\}$$

represents a three-dimensional array X of size 16×16×6 with two uniform blocks. One of these contains the elements in the region delimited by (0,8,0) and (5,15,3) and the other capturing the elements from (9,4,0) to (15,15,5), as shown in Figure 2.2.

Our approach first identifies the nonuniform blocks of a given array based on the specified uniform blocks. For example, for the array in Figure 2.2, we identify 4 nonuniform blocks: $[0, 0, 0 : 5, 5, 5]$, $[0, 5, 0 : 5, 15, 5]$, $[6, 0, 0 : 8, 15, 5]$, $[9, 0, 0 : 15, 3, 5]$. After performing this for each input array of a given loop nest, our compiler-based approach next restructures the loop nest (which will be explained in detail later in this section) and determines the compressed format of the output array(s) in the nest. In moving from one loop nest to the next, we also update the set of arrays whose compressed formats have been determined so far. In this way, when the entire program has been processed, our approach determines the compressed formats of all the arrays with value locality and restructures all the loop nests (i.e., the data access pattern of the entire application) that operate on these arrays.

2.3.2 Mathematical Details

Our focus is on applications with affine loop bounds and affine array subscript functions. Given vectors $\vec{u} = (u_1, u_2, ..., u_n)^{\mathrm{T}}$ and $\vec{v} = (v_1, v_2, ..., v_n)^{\mathrm{T}}$, we define an order relation "$\leq$" as:

$$\vec{u} \leq \vec{v} \iff \forall i, 1 \leq i \leq n \rightarrow u_i \leq v_i.$$

Further, we define integer vector set $[\vec{u}, \vec{v}]$ as:

$$[\vec{u}, \vec{v}] \equiv \{\vec{I} \mid \vec{u} \leq \vec{I} \leq \vec{v}\}.$$

We focus on loop nests of the following form:

$\mathcal{T}$: for $i_1 = u_1$ to v_1
  for $i_2 = u_2$ to v_2
  ...
    for $i_n = u_n$ to v_n {
      $Y[\vec{I}] = f(X_1[G_1\vec{I} + \vec{g}_1], X_2[G_2\vec{I} + \vec{g}_2], ..., X_m[G_m\vec{I} + \vec{g}_m])$;
    }

In this loop nest, we evaluate the value of array Y based on the value of arrays X_1, X_2, ..., X_m. Specifically, the value of array element $Y[\vec{I}]$ depends on the values of array elements $X_1[G_1\vec{I} + \vec{g}_1]$, $X_2[G_2\vec{I} + \vec{g}_2]$, ..., $X_m[G_m\vec{I} + \vec{g}_m]$, where $\vec{I}$ (referred to as the "iteration vector") is the vector of the index variables, i.e., $\vec{I} = (i_1, i_2, ..., i_n)$; and the affine function $G_k\vec{I} + \vec{g}_k$ $(0 \leq k \leq m)$, where G is a constant matrix and $\vec{g}$ is a constant vector, maps the iteration vector to the subscript of array X_k. As an example, for the reference to array X_1 in Figure 2.4(a), G_1 is identity matrix and $\vec{g}_1$ is $(0\ 0)^{\mathrm{T}}$.

Let us assume, without loss of generality, array X_1 contains r uniform blocks, which are represented as $X_1[\vec{U}_1^{(1)}, \vec{V}_1^{(1)}]$, $X_1[\vec{U}_1^{(2)}, \vec{V}_1^{(2)}]$, ..., $X_1[\vec{U}_1^{(r)}, \vec{V}_1^{(r)}]$. Note that these uniform blocks do not intersect with each other, i.e., we have $[\vec{U}_1^{(p)}, \vec{V}_1^{(p)}] \cap [\vec{U}_1^{(q)}, \vec{V}_1^{(q)}] = \phi$. Consequently, if data dependencies permit, we can partition loop nest $\mathcal{T}$ into a set of loop nests:

$$\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_r\},$$

such that the array access instruction $X_1[G_1\vec{I} + \vec{g}_1]$ in loop nest $\mathcal{T}_0$ does not access any element of any uniform region of X_1, and the array access instruction in loop nest $\mathcal{T}_j$ $(j = 1, 2, ...r)$ does not access the element in any uniform region of array X_1 other than $X_1[\vec{U}_1^{(j)}, \vec{V}_1^{(j)}]$. In other words, the iteration space of loop nest $\mathcal{T}_j$ $(j = 1, 2, ..., r)$ can be specified as:

$$\mathcal{I}_j = \{\vec{I} \mid \vec{U}_1^{(j)} \leq G_1\vec{I} + \vec{g}_1 \leq \vec{V}_1^{(j)}\},$$

and the iteration space of loop nest $\mathcal{T}_0$ can be computed as:

$$\mathcal{I}_0 = \{\vec{I} \mid \forall 1 \leq j \leq r : (G_1\vec{I} + \vec{g}_1 < \vec{U}_1^{(j)} \text{ and } \vec{V}_1^{(j)} < G_1\vec{I} + \vec{g}_1)\}.$$

Notice that, by using a tool (such as Omega Library [94]) that performs polyhedral arithmetic, one can construct a loop nest for each iteration space $\mathcal{I}_j$ ($j = 0, 1, ..., r$), i.e., we can construct a loop nest that iterates over the elements contained in a given $\mathcal{I}_j$.

Using the method described above, we first partition loop nest $\mathcal{T}$ into loop nests $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_r$ based on array access instruction $X_0[G_0\vec{I} + \vec{g}_0]$. After that, using the same method, we further partition loop nests $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_r$ based on array access instruction $X_1[G_1\vec{I} + \vec{g}_1]$. Note that there is no need to further partition loop nest $\mathcal{T}_0$ since the elements of array Y whose values are evaluated in this loop nest do not belong to any uniform block of array Y. We repeat this loop partitioning procedure, each time based on a different array access instruction $X_k[G_k\vec{I} + \vec{g}_k]$, until all the array access instructions are considered. Finally, we obtain a set of loop nests, each of which evaluates the value of a uniform block of array Y.

Figure 2.3 shows the sketch of our compiler algorithm. We can observe that our compiler partitions each loop nest that accesses arrays with uniform blocks based on the uniform blocks in these arrays. In addition, when the compressed format of an intermediate array is determined, we add the information about the format of this array into a repository so that it can be used in the analysis of the next loop nest. In a sense, we incrementally build a set of compressed formats for arrays with value locality.

In our implementation, we also change the declaration of each array according to its uniform blocks, so that memory space savings are achieved. For example, compressed integer array $X :: \{2 : 10 : 10 : 2 : [0, 0 : 3, 3 : a]; [5, 5 : 9, 9 : b]\}$ can be declared in C code as:

int Xa = a; // uniform block $[0, 0 : 3, 3 : a]$

int Xb = b; // uniform block $[5, 5 : 9, 9 : b]$

int X1[6][4]; // nonuniform block $X[0, 4 : 3, 9]$

int X2[1][10]; // nonuniform block $X[4, 4 : 0, 9]$

int X2[5][5]; // nonuniform block $X[5, 0 : 9, 4]$

```
for each loop nest T in the application do
  partition loop nest T based on the uniform blocks
    of the arrays accessed by T;
  for each array Y appears on the right hand side do
    determine the uniform blocks in Y;
    add the compressed format of array Y to repository;
  end
end
```

Figure 2.3: Sketch of our compiler algorithm.

2.4 Compression-Aimed Parallelization

In this section, we present our parallelization approach for code that uses compressed data. First, in Section 2.4.1, we explain the mathematical details. Then, in Section 2.4.2, we discuss some factors that could affect the results. Finally, in Section 2.4.3, we give examples to show how our approach works.

2.4.1 Theory

In the case of on-chip multiprocessors we want to distribute the outermost loop iterations to the multiple processors working in parallel.

In a general case, we have p processors.

If our loop is like:

for i = lb_i *to* ub_i

loop-body

The number of iterations for the outermost loop is $N = ub_i - lb_i + 1$.

When we distribute this loop to to p processors, each processor will get a portion of the loop:

$p_1 : lb_i \rightarrow lb_i + N/p - 1$

$p_2 : lb_i + N/p \rightarrow lb_i + 2N/p - 1$

...

$p_j : lb_i + (j-1)N/p \rightarrow lb_i + jN/p - 1$

...

$p_p : lb_i + (p-1)N/p \rightarrow lb_i + pN/p - 1 = ub_i$

Here, the assumption $N \gg p$ ensures that the slightly uneven distribution of iterations due to rounding of N/p has a minimal effect.

The objective in loop distribution is to divide the iterations among the processors in such a way that the time spent by each processor is approximately the same. In other words,we want the cost for each processor to be approximately equal; that is, if c_i denotes the cost for processor i, then $c_1 \cong c_2 \cong ... \cong c_p$.

When the arrays used in the loop are in compressed format, the loop is broken down into smaller uniform and non-uniform loops as described in previous sections. Let's say *size(loop)* gives the number of outermost iterations for a nested loop structure *loop*. Let's further assume that we have U uniform loops and N_u non-uniform loops; the uniform loops are accessed as $u_i(i \in 1..U)$ and the non-uniform loops are accessed as $nu_j(j \in 1..N_u)$.
Then the total number of iterations for outermost loops is as follows:
Uniform: $S_u = \sum size(u(i))$
Non-uniform: $S_{nu} = \sum size(nu(i))$

In a normal partition, each processor gets an equal amount of iterations, that is $N/p = (S_u + S_{nu})/p$. But uniform and non-uniform loops may be grouped in specific processors. Since we expect uniform iterations to take much less time with respect to non-uniform ones, this will disturb the balance, and just assigning the same number of iterations to each processor is no longer sufficient. In such a case, we have to adjust the distribution process for optimized results.

To distribute the iterations, for balanced execution times, we use the following approach. The goal is to assign each processor p_i approximately the same (S_u/p) number of non-uniform iterations. Each processor should also get approximately the same (S_u/p) number of uniform iterations.
Then we can calculate the workload on processor p_i, that is, $Cost_i$ as:
$Cost_i = \alpha_i.c_nu + \beta_i.c_u$,
where α_i and β_i are the number of non-uniform and uniform iterations on p_i, respectively, and c_nu and c_u are the respective average costs of a non-uniform and a uniform iteration.
$\alpha_i = S_{nu}/p$
$\beta_i = S_u/p$

Since non-uniform operations are cost-wise dominant, distributing these evenly to each processor is more important, but we are also distributing the uniform ones evenly.

2.4.2 Discussion

If there are k statements in the loop body, due to the use of compressed representation it is possible that for some iterations a subset S_1 of the statements will be uniform, and for some iterations, some other subset S_2 will be uniform. In general, there can be $Q = 2^k$ such combinations.

Then we can calculate the cost for processor p_i as follows:

$Cost_i = \sum_{k=1}^{Q} \gamma_k.c_k$

where γ_k and c_k are the frequency and cost of combination k, respectively.

There are some important factors that may have an effect on the execution time for the statements. One of these is the cache architecture of the system. The cache architecture of a multiprocessor system can have a dramatic effect on the execution time, depending on the data access patterns of an application, since access times for cache and memory are very different. Another important factor is the use of conditionals. If there are conditional statements (e.g. an *if* statement) within the loop body, different branches may possibly have very different execution paths that may considerably differ in execution time.

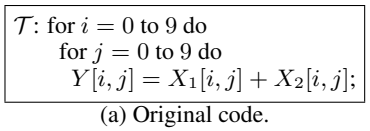
2.4.3 Examples

We now present two examples to illustrate the working of our approach. In the first example, we want to focus on loop partitioning, so we assume a single processor system, and avoid parallelization. In the second example, we assume the system has 2 processors, and show parallelization as well as partitioning.

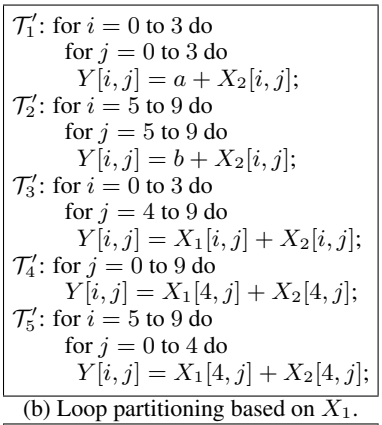
Single Processor Example

Figure 2.4(a) shows a loop nest that operates on three arrays. Our goal is to determine the compressed format for array Y, based on the given compressed format of the input arrays X_1 and X_2. Figure 2.4(b) shows the result of partitioning this loop nest based on the uniform blocks of X_1. In the resulting loop nests, $\mathcal{T}_1'$ and $\mathcal{T}_2'$ access the uniform blocks of array X_1, and therefore, they are selected for further partitioning, this time based on the uniform block of array X_2. The other loop nests $\mathcal{T}_3'$, $\mathcal{T}_4'$, and $\mathcal{T}_5'$ do not need any further partitioning.

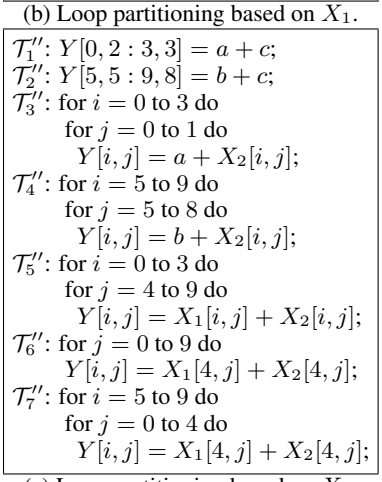
Figure 2.4(c) shows the final resulting code after partitioning $\mathcal{T}_1'$ and $\mathcal{T}_2'$ based on the uniform block of array X_2. From this generated code, we can observe that array Y contains two uniform blocks, namely $Y[0, 2:3, 3]$ and $Y[5, 5:9, 8]$, whose values are $a + c$ and $b + c$, respectively.



(a) Original code.



(b) Loop partitioning based on X_1.



(c) Loop partitioning based on X_2.

Figure 2.4: Example. The compressed format of arrays X_1 and X_2 are $X_1 :: \{2 : 10 : 10 : 2 : [0, 0 : 3, 3 : a]; [5, 5 : 9, 9 : b]\}$ and $X_2 :: \{1 : 10 : 10 : 1 : [1, 2 : 9, 8 : c]\}$, respectively.

Multiprocessor Example

Figure 2.5(a) shows a loop nest that operates on three arrays. Our goal is to determine the compressed format for array Y, based on the given compressed format of the input arrays X_1 and X_2. Figure 2.5(b) shows the result of partitioning this loop nest based on the uniform blocks of X_1 and X_2. In the resulting loop nests, $\mathcal{T}'_1$ accesses the uniform blocks of arrays X_1 and X_2, whereas $\mathcal{T}'_2$ accesses the non-uniform blocks. Since $\mathcal{T}'_1$ and $\mathcal{T}'_2$ contain an equal number of iterations, each one is assigned to separate processor.

Figure 2.5(c) shows the final resulting code after partitioning $\mathcal{T}'_1$ and $\mathcal{T}'_2$ and assigning to processors in a balanced manner. From this generated code, we can observe that the number of non-uniform block iterations performed by each processor is equal. The number of uniform block iterations performed by each processor is also equal.

$\mathcal{T}$: for $i = 0$ to 1599 do
  for $j = 0$ to 1599 do
    $Y[i,j] = X_1[i,j] * X_2[i,j]$;

(a) Original code.

P_1
$\mathcal{T}'_1$: for $i = 0$ to 799 do
  for $j = 0$ to 1599 do
    $Y[i,j] = u * v$;

P_2
$\mathcal{T}'_2$: for $i = 800$ to 1599 do
  for $j = 0$ to 1599 do
    $Y[i,j] = X_1[i,j] * X_2[i,j]$;

(b) Parallelization and loop partitioning.

P_1
$\mathcal{T}''_1$: for $i = 0$ to 399 do
  for $j = 0$ to 1599 do
    $Y[i,j] = u * v$;
$\mathcal{T}''_2$: for $i = 800$ to 1199 do
  for $j = 0$ to 1599 do
    $Y[i,j] = X_1[i,j] * X_2[i,j]$;

P_2
$\mathcal{T}''_3$: for $i = 400$ to 799 do
  for $j = 0$ to 1599 do
    $Y[i,j] = u * v$;
$\mathcal{T}''_4$: for $i = 1200$ to 1599 do
  for $j = 0$ to 1599 do
    $Y[i,j] = X_1[i,j] * X_2[i,j]$;

(c) Balanced parallelization and loop partitioning .

Figure 2.5: Example for 2 processors. The compressed format of arrays X_1 and X_2 are $X_1 :: \{2 : 1600 : 1600 : 1 : [1, 1 : 800, 800 : u]\}$ and $X_2 :: \{2 : 1600 : 1600 : 1 : [1, 1 : 800, 800 : v]\}$, respectively

Program	**S**ource	**N**umber of Arrays	**M**emory (MB)	**T**ime (ms)
adi	Livermore	6	137	3120
apsi	Perfect Club	17	27.3	42
eflux	Perfect Club	5	14	410
tomcatv	Spec	9	0.23	6.5

Table 2.1: Benchmarks

Table 2.2: Compressed formats for the input arrays of our benchmarks.

Program	**C**ompressed Formats
adi	au1::{3:2000:2000:2:1:[100,100,1:601,601,1:v1]} au2::{3:2000:2000:2:1:[100,100,1:601,601,1:v2]} au3::{3:2000:2000:2:1:[100,100,1:601,601,1:v3]}
apsi	DKZH::{3:700:700:700:2:[1,700,700:51,700,700:a];[200,700,700:301,700,700:b]} WZ::{3:700:700:700:2:[2,700,700:50,700,700:a];[201,700,700:300,700,700:b]} HVAR{1:700:2:[1:51:e];[200,301:f]} DTM{1:700:2:[1:51:g];[200,301:h]} DKM{1:700:2:[1:51:i];[200,301:j]}
eflux	W::{3:1000:334:4:4:[400,50,1:800,250,1:p];[400,50,2:800,250,2:q];[400,50,3:800,250,3:r];[400,50,4:800,250,4:s]} X::{3:1000:334:4:2:[400,50,1:800,250,1:t];[400,50,2:800,250,2:u]} P::{2:1000:334:1:[400,50:800,250:v]}
tomcatv	X::{2:100:100:1:[31,31:80,80,p]} Y::{2:100:100:1:[31,31:80,80,q]}

Table 2.3: Different compressed formats for the adi benchmark.

ID	**C**ompressed Formats
1	au1::3:2000:2000:2:1:[100,100,1:601,601,1:v1] au2::3:2000:2000:2:1:[100,100,1:601,601,1:v2] au3::3:2000:2000:2:1:[100,100,1:601,601,1:v3]
2	au1::3:2000:2000:2:1:[100,100,1:900,900,1:v1] au2::3:2000:2000:2:1:[100,100,1:900,900,1:v2] au3::3:2000:2000:2:1:[100,100,1:900,900,1:v3]
3	au1::3:2000:2000:2:1:[100,100,1:1201,1200,1:v1] au2::3:2000:2000:2:1:[100,100,1:1201,1200,1:v2] au3::3:2000:2000:2:1:[100,100,1:1201,1200,1:v3]
4	au1::3:2000:2000:2:2:[100,100,1:700,700,1:v1];[1000,1000,1:1500,1500,1:v4] au2::3:2000:2000:2:2:[100,100,1:700,700,1:v2];[1000,1000,1:1500,1500,1:v5] au3::3:2000:2000:2:2:[100,100,1:700,700,1:v3];[1000,1000,1:1500,1500,1:v6]

2.5 Experiments

We implemented a source-to-source translator to map a given application code into a transformed code that uses the compressed formats for arrays and modified loops that take advantage of the value locality in the application.

2.5.1 Setup

For the first experiment, we used a set of four array-based benchmark codes to test our approach. Table 2.1 gives the important information about these four benchmarks. The second column of this table gives the number of arrays manipulated by each benchmark. The third column gives the memory space requirement of each benchmark and the last column shows the execution cycles obtained. The values in these last two columns are for the unmodified (original) benchmarks without any particular optimization for memory space saving.

We applied compression in each of these cases, and measured times on a single processor and 8 processor CMP, both with regular and balanced distribution of loops. The applied array compression saves 19% memory space on the average for the patterns used.

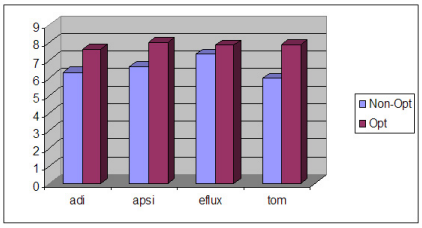


Figure 2.6: Obtained speedups for different benchmarks.

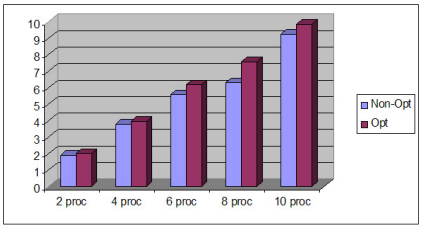


Figure 2.7: Obtained speedups for the adi benchmark for varying number of processors.

Figure 2.6 gives the speedup obtained by using CMP. For each pair of columns, the first denotes regular distribution, and the second denotes balanced distribution. These results indicate that the proposed balancing approach increases the speedup by 20% on the average. To obtain these results, we used the (value) patterns shown in Table 2.2. Note that these patterns define the compressed formats for the input arrays in our application.

For the second and third experiments, we focus on one of our applications (adi). In the second experiment, we used the same pattern for adi that is given in Table 2.2, changing the number of processors to measure speedup values. The results are given in Figure 2.7. Again, each pair of columns shows the speedup obtained by regular and balanced distributions, respectively.

In our third experiment, we measure the behavior of our approach when the input arrays have different value locality (and thus different compressed formats). For this purpose, we give the speedups under different value patterns (compressed format) in Figure 2.8. The compressed format patterns for adi used in obtaining the results in Figure 2.8 are listed in Table 2.3.

To recap, the results in Figures 2.6, 2.7 and 2.8 clearly demonstrate that the proposed balanced loop distribution approach for compressed arrays is successful in providing better speedup.

2.5.2 Results

While the results presented so far are encouraging, one may also want to see how they compare to memory savings that could be obtained from alternate schemes such as those based on lifetime analysis. In a lifetime

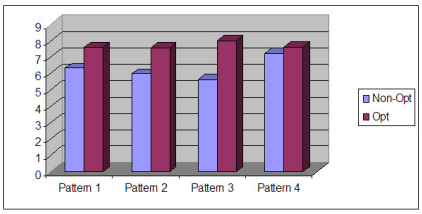


Figure 2.8: Obtained speedups for the adi benchmark with different uniform block sizes in the input arrays.

analysis based approach, the dead memory locations (i.e., the locations that hold data that will not be referenced in the rest of the execution) are recycled, and this helps reduce both average and maximum memory requirements. For the applications in our experimental suite, we found that such a lifetime-based approach can reduce memory space requirement by 13.3% on the average (compared to 19% average space savings our approach achieves). Therefore, from the memory space saving angle, our approach is competitive with the lifetime-based approach. In addition, we need to mention that, while a pure lifetime-based approach does not bring any performance benefits in general, as has been shown above, our approach reduces execution times as well, since it exploits value locality by transforming loop nests.

Implementing compressed format generally increases the code size, but since the code sizes of these benchmarks are much smaller than the sizes of the data arrays they manipulate,[§] the impact of the code size increase is not significant. Note that the impact of this code size increase on the execution cycles has already been included in the execution time results presented above.

2.6 Conclusion

Reducing data memory requirements of embedded applications is beneficial from cost, area, and power perspectives. However, most of the prior approaches to memory space reduction incur certain performance penalties. Our goal is to explore how one can reduce memory space requirements and improve performance at the same time. We propose a compiler-directed approach to achieve this objective. Specifically, in the proposed approach, the compiler exploits the value locality in array applications and determines compressed formats for arrays with value locality. This helps reduce the memory space demand of the loop nests in the application code. The compiler also restructures the application code to take advantage of value locality and this enhances performance. Our experiments with 4 array-intensive applications show that the proposed approach reduces the memory space requirements by 19% on average and execution time by 14% on average,

[§]While the code sizes are in KB ranges, the data size are in MB ranges.

and performs better than a lifetime-based memory space saving strategy.

The focus of this approach is on data compression. It works on both single and multi processor systems, and does load balancing on CMPs. It does not assume the existence of any particular memory subsystem, like a cache or SPM. In the next chapter we will focus on CMP systems with a shared cache.

Chapter 3

Code Scheduling for Optimizing Parallelism and Data Locality

3.1 Introduction

In this chapter, we propose and evaluate a static code scheduling scheme, which considers both parallelism and data locality at the same time. We target a shared memory based CMP architecture, where multiple CPUs share an on-chip cache space. A simplified view of this architecture is illustrated in Figure 1.1. We also assume the existence of a large off-chip memory space, shared by all processors in the system.

As chip multiprocessors are finding their ways into commercial market in embedded domain, programming support for these devices is becoming increasingly critical. This support includes language, compiler, and debugging related issues and is likely to receive a lot of attention in the near future.

In a chip multiprocessor based execution environment, two issues are critical to address: *parallelism* and *data locality.* The first of these indicates how well an execution exploits available computation resources. Ideally, one wants to use all available processors at each step of computation if doing so improves performance.* Data locality, on the other hand, captures how well an execution exercises available memory hierarchy. The concept of data locality is particularly important in the context of chip multiprocessors as the gap between latencies of the on-chip and off-chip accesses is huge. Clearly, one wants to satisfy majority of data references from the higher levels of memory, i.e., those components that are close to processor. In order to achieve good

*In some cases, increasing the number of processors used beyond a certain point can degrade performance due to increased inter-processor communication.

performance in a chip multiprocessor based embedded system, an optimizing compiler has to exploit *both* parallelism and locality in a synergistic fashion.

Unfortunately, most of the published work in the literature focuses only on one of these problems, and this can prevent one from achieving the best possible performance. For example, if locality remains unoptimized, one can expect poor performance at runtime even if all available parallelism is extracted from the application. Similarly, a locality-optimized program that is not parallelized appropriately can result in poor runtime behavior.

Therefore, optimizing for both parallelism and locality is very important in this CMP architecture. In particular, in order to attain good performance, one has to use all available CPUs to the maximum extent allowed by intrinsic data dependencies in the code, and the reused data elements should be caught in the on-chip cache space as much as possible (instead of going off chip). In this chapter, we demonstrate how data scheduling can be used for this purpose.

We propose and evaluate a compiler-directed code partitioning/scheduling scheme, which considers both parallelism and data locality at the same time. Our target application domain is data-intensive codes that use arrays as the primary data structures. These arrays have affine subscript expressions and are operated using loops with affine bounds. Our compiler captures the inherent parallelism and data reuse in the application code being analyzed using a novel representation called the *locality-parallelism graph*, or LPG for short. It then executes a partitioning/scheduling algorithm on this graph, which assigns the nodes of this graph to the processors in the parallel architecture (a chip multiprocessor). An important characteristic of this algorithm is that it has a global view of the computations in the application code. That is, during scheduling, it considers all loop nests and data access patterns before assigning a scheduling slot to a computation. In contrast, most current multi-core scheduling efforts are local, i.e., focus on a single loop at a time. We implemented this algorithm and evaluated its effectiveness using a set of four benchmark codes. The results collected so far indicate that our approach improves execution latency significantly.

We also present an ILP (Integer Linear Programming) based formulation of the combined parallelization and data-locality optimization problem. We implemented this ILP solver based solution and compared the results it generated to those obtained using our heuristic approach. The collected experimental results indicate that our approach gets within 4% of the ILP based solution.

The rest of this chapter is organized as follows. Related work is discussed and compared to our work in Section 3.2. Section 3.3 describes our compiler representation, LPG, and Section 3.4 presents the details of our partitioning/scheduling algorithm. The ILP formulation of the problem is discussed in Section 3.5. An experimental evaluation of the proposed approach is given in Section 3.6, and the chapter is concluded in Section 3.7 with a summary.

3.2 Related Work

In this section, we first evaluate the previous work on parallelization then we revise the efforts on data locality. Parallelism can be obtained at different levels of abstraction. Instruction-level parallelism is exploited by high-performance microprocessors, whereas data-level parallelism is utilized in nested loops using compilers. Similarly, task-level parallelism can be found in many embedded applications. To exploit the data-level parallelism, Kadayif et al [50] proposed to use different number of processor cores for each loop nest to obtain energy savings. This way idle processors are switched to a low-power mode to increase the energy savings. Mei et al [81] propose a modulo scheduling algorithm for coarse-grained reconfigurable architectures by exploiting loop-level parallelism. To parallelize loops, Bondalapati [14] exploits the distributed memory available in the digital signal processing domain. More specifically, he exploits the reconfigurable architecture by implementing a data context switching technique. Goumas et al [37] try to generate parallel code for tiled nested loops through different loop transformations using MPI. Hogstedt et al [42] predict the execution time of tiled loop nests and use this prediction to automatically determine the tiling parameters that minimizes the execution time. Arenaz et al [7] exploit coarse-grain parallelism by a gated single assignment (GSA) based approach with complex computations. Yu and D'Hollander [130] construct an iteration space dependency graph to visualize a 3D iteration space. Beletskyy et al [12] adopt a hyperplane-based representation to apply on transformation matrices with both uniform and non-uniform dependences. Lim et al [73] employ affine partitioning to maximize parallelism with minimum communication overhead. Ozturk et al [97] focus on optimizing parallelism in chip multiprocessors using constraint networks. In [104], authors propose an abstract interpretation to analyze needed loop parallelization.

Two major techniques to exploit locality are loop transformations and data transformations. Wolf and Lam [116] define reuse vectors and reuse spaces. Moreover, they use these concepts to implement an iteration space optimization technique. Similarly, Li [70] uses reuse vectors to detect the dimensions of loop nest that carry reuse. In [19], authors analyze data dependences by a variable renaming technique to break anti and output dependences along with a technique to resolve recurrences in a nested loop. Navarro et al [87] represent the locality using a locality graph, and mixed integer nonlinear programming is used on this graph to minimize the communication cost and load imbalance. Carr et al [17] re-order the computation by using a simple locality criterion to enhance data locality. Tiling [26, 60, 62] is another loop based locality enhancing technique. On the data transformation side, in [92], authors generate the code with a given data transformation matrix. Kandemir et al [53] implement an explicit layout representation, whereas [66] focuses more on memory consumption reduction due to a layout transformation. There are also efforts to combine data and loop transformations. Among these is one of the first papers [24] that offers a scheme which unifies

loop and data transformations. On the other hand, Anderson et al [6] propose a transformation technique that accesses contiguous data elements. Chen et al [21] employ a constraint network based solution to the combined data/loop optimization problem.

Hwu et al [82] present a parallel programming model for many-core microprocessors, and provide initial technical approaches towards this goal. Leverich et al [67] compare hardware-managed coherent caches and software-managed streaming memory under the same set of assumptions in terms of technology, area, and computational capabilities. Xue et al [123] propose a memory-conscious loop parallelization strategy which is formulated as a branch-and-bound problem. Hughes et al [45] examine parallelization potential of physics-based simulation and characterize its behavior on a chip multiprocessor.

As compared to these prior studies, we target chip multiprocessors where processors share an on-chip cache and propose a scheduling scheme for improving both data reuse and parallelism in a synergistic manner.

3.3 Locality-Parallelism Graph

Our scheduling algorithm, which targets loop based applications, operates with a *locality-parallelism graph* (*LPG*) of code blocks. This graph captures the dependencies among code blocks and locality among the blocks. An LPG is an acyclic graph $G(V, E_{dep}, E_{loc})$, where V is a set of nodes and E_{dep} and E_{loc} are sets of edges. Each v_i in V represents a *code block* (which will be explained in detail shortly). A directed edge $e_{i,j}$ in E_{dep} from v_i in V to v_j in V means there is a dependency between v_i and v_j (i.e., data produced by v_i is used by v_j). In this case, v_i is an immediate predecessor of v_j, and v_j is an immediate successor of v_i. We denote the set of immediate predecessors of a node v as $Pred_v$, and the set of immediate successors of v as $Succ_v$. A directed edge $e'_{i,j}$ in E_{loc} from v_i to v_j means there is a data reuse, i.e., v_i and v_j share some data between them. The weight $W_{e_{i,j}}$ of edge $e'_{i,j}$ captures the amount of data shared by the two nodes (code blocks). To make our problem formulation simpler, all non-existing edges in an LPG are assumed to have a weight of 0. The set of nodes that share a locality edge with a node v is denoted as Loc_v.

When we have a loop with a large number of iterations, we can rewrite the same loop as a set of loops of fewer iterations, which have the same loop body as the original. For example, if the original loop has n iterations, we can break it into k blocks, each block having roughly n/k iterations. In this context, we call each one of these smaller loops a *code block*. Notice that a given loop (i.e., the set of iterations in it) can be divided into multiple code blocks (unit of scheduling in our approach) and each code block contains a subset of the iterations in that loop. These code blocks can then be executed in parallel, based on the data dependency constraints. The number and size (i.e., the number of iterations) of the code blocks can be

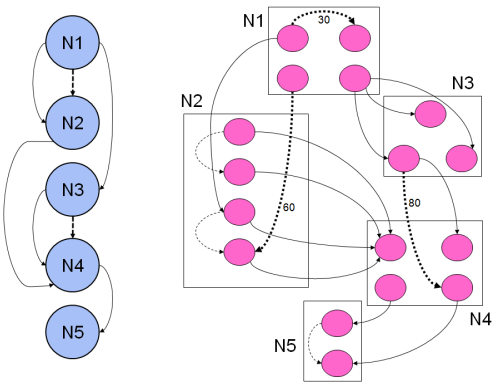


Figure 3.1: An example illustrating the concept of LPG.

arranged to achieve the desired level of granularity. In our case, this code block generation and process of extracting data dependencies among code blocks is carried out by the compiler.

As an example, in the graph in Figure 3.1, we have five separate loop nests in our code (shown on the left.) The solid lines represent the data dependencies, whereas the dotted lines capture the data reuse edges.[†] We partition each of these loops into smaller loops of reasonable size (i.e., into code blocks). On the right part of this figure, we see the code blocks that are generated by the partitioning of loop nests. Note that the number of edges has increased, since it now shows the dependencies and data reuse between small code blocks, instead of larger nests. This graph on the right (which is our LPG in this case) shows dependencies and data localities at a finer granularity, and is the main compiler based data structure on which our partitioning/scheduling scheme operates. When there is no confusion, in the remainder of this chapter, when we mention ”block”, we mean ”code block”.

3.4 Our Approach

3.4.1 Theory

As mentioned earlier, our target domain is data intensive computations that operate primarily on array data using nested loops. We assume that the loop bounds and array subscript expressions are affine functions of enclosing loops and loop-independent variables/constants.

We can think of a *schedule* as a two dimensional matrix, where the rows represent scheduling steps (also

[†]Note that, while a data dependency edge between two nodes means that there is also a data reuse edge between them, the opposite may not always be true. This is because if two code blocks only read the same data, this does not introduce a data dependency but we still have a data reuse between them.

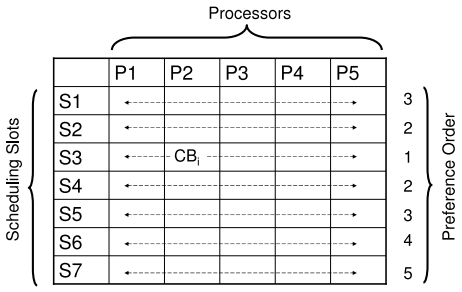


Figure 3.2: An example scheduling matrix.

referred to as the execution steps in this work) and the columns correspond to available CPUs. At the end of scheduling (which is explained below), we fill the entries of this matrix such that both parallelism and data locality are improved.

Our goal is to schedule, considering the CPUs we have in our given CMP, code blocks that have data reuse between them as close together as possible (in time), while respecting data dependencies. This is because, if two or more blocks that access the same data are scheduled in the same (or close by) execution steps, then the data will be loaded into the on-chip cache once, and used by all of them, instead of each block loading the same data into the cache separately at different times. This will hopefully result in an increase in cache hit ratio and enhance overall performance. [‡]

Consider for example the scheduling matrix shown in Figure 3.2. From the parallelism perspective, we want to fill all the entries in a row (scheduling slots) with code blocks. However, as mentioned earlier, data dependencies among code blocks may not allow such full utilization of scheduling slots. From the data locality perspective on the other hand, we want the code blocks that share data among them to be scheduled in close by scheduling slots. For example, Figure 3.2 shows the preferable scheduling slots (1 being the most preferable and 5 being the least) for a code block CB_j that has data reuse with code block CB_i if the latter has already been scheduled as shown in the figure.

To accomplish this goal, we designed a heuristic algorithm for resource-constrained scheduling. In this section, the key parts of our algorithm are given as a pseudo-code along with short explanations of the functions implemented. In our approach (which is fully automated), certain data structures are used throughout our algorithm, and are considered global. These are given in Table 3.1.

Before discussing the technical details of our scheduling strategy, let us informally state what it does. Our

[‡]Ideally, we want to load each data element from the off-chip memory to the shared on-chip cache only once. However, this may not always be realizable due to data dependencies in the code and other potential scheduling constraints.

Table 3.1: Global variables.

Variable	Definition
$Step_v$	Execution step that node v is scheduled. The value is 0 for unscheduled nodes.
Sch_i	the set of nodes scheduled in execution step i.
$sReady$	the set of nodes that are ready to be scheduled.
$sManda$	the set of nodes that are ready and have to be scheduled as soon as possible.
$sRemain$	the remaining set of nodes that are not ready.
cap	the remaining capacity in the current step.

approach consists of arranging nodes of a given LPG into an execution schedule depending on the amount of data reuse they exhibit and parallelism they have. At each step, a node is scheduled for the current execution step (scheduling slot); the node is selected such that it has the largest amount of data reuse with the nodes already selected for that step, as well as nodes scheduled for previous steps in a weighted fashion (i.e., data reuse within the same step has more weight than that of previous steps). While choosing the first node in a step, its amount of data reuse with ready-to-be-scheduled but unscheduled nodes is also taken into account.

In more technical terms, our approach starts by computing the ASAP (*as soon as possible*) and ALAP (*as late as possible*) values for the nodes of the LPG at hand. An ASAP value for a node v gives the earliest step of execution that v can be scheduled. The ASAP algorithm assigns an ASAP label S_v to each node v. Similarly, in ALAP scheduling, each block is scheduled to start at the latest possible step. An ALAP value for a node v represents the latest step of execution that v can be scheduled. The ALAP algorithm assigns an ALAP label L_v (step index) to each node v. T represents an upper bound on the number of steps. We omit the pseudo codes for the $ASAP()$ and $ALAP()$ procedures since they are well known in literature [83]. Each step of execution has a capacity of p, that is, the number of processors in the system. In other words, at most p code blocks can be scheduled in an execution step.

The procedure $calcSchedule(p, \alpha, \beta, flag)$ in Algorithm 1 calculates the schedule for a system with p processors. It first initializes several variables, populates $sRemain$, and then assigns $ASAP/ALAP$ labels to each node. $insertReadyNodes(1)$ adds the possible starting nodes (nodes that are not dependent on any other nodes) to $sReady$.

The rest of the code is the main $while$ loop, which iterates as long as there are ready nodes (code blocks) to be scheduled. At each iteration, it first goes on to next execution step, initializes variables, and calls $doMandatory$ (schedules nodes that would otherwise increase the total number of execution steps.) At each iteration of the inner $while$ loop, the ready node with the highest score is selected and scheduled, as long as there are ready nodes and there is room in the current execution step to accommodate new code block(s).

When either condition is $false$, $insertReadyNodes(cs + 1)$ loads $sReady$ with nodes ready for the next execution step and goes back to the next iteration of the main $while$ loop.

The selection of the nodes from $sReady$ is performed according to the score calculated by $calcScore\_Sch$. The boolean $flag$ enables contribution to the score by $calcScore\_sReady$, in the case of current step being empty. If the flag is down, the first node of the step is chosen, solely based on data reuses it exhibits with nodes scheduled in previous steps. Otherwise, a constant (β) times the $calcScore\_sReady$ score (a node's locality with $p - 1$ other nodes in $sReady$) is added to the total score. Note that the use of the parameters $flag$, α and β enables us to generate multiple heuristics and fine-tune our algorithm.

The procedure $insertReadyNodes(s)$ in Algorithm 2 scans the set of nodes $sRemain$ and determines if any of the nodes are ready for step s (i.e., all its predecessors have already been scheduled, and step s lies between its ASAP and ALAP labels). It then puts all ready nodes in the set of ready nodes, $sReady$, and deletes them from $sRemain$.

The procedure $doMandatory(s)$ in Algorithm 3 iterates through the nodes in $sReady$ to find the nodes that have an $ALAP$ label L_v such that $L_v \leq s$ (the given step) and adds them to $sManda$. It then schedules the nodes in $sManda$ in the order of non-decreasing $ALAP$ values as long as there are nodes in $sManda$ and there is available capacity in step s.

The procedure $scheduleNode(v, s)$ given as Algorithm 4 below schedules node v in execution step s. It updates Sch_s and $Step_v$ accordingly, deletes the node from $sReady$, and decreases the remaining capacity of the execution step.

The function $calcScore\_sReady(v_i, c)$ in Algorithm 5 is optionally used when it is time to pick the first node for a step, since the amount of data reuse (i.e., the number of data elements shared) with other nodes already scheduled on the same step is not sufficiently high. Instead, it calculates a total weight value for v_i, based on its amount of data reuse with other unscheduled ready nodes. The parameter c is used as an upper bound on the number of nodes considered, since there can be at most p nodes scheduled in a step. Basically, the function returns the sum of highest t locality weights that v_i shares with other ready nodes. The actual number of nodes is given by $t = min(c, \mid Loc_{v_i} \cap sReady \mid)$.

The weight value returned by $calcScore\_sReady(v_i, c)$ is given by the formula:

$$\sum_{v_j \in (Loc_{v_i} \cap sReady)} W_{e_{i,j}},$$

where v_j is one of the t nodes in $sReady$ with the highest data reuse with respect to v_i.

The function $calcScore\_Sch(v_i, \alpha, s)$ given in Algorithm 6 calculates the amount of weighted data reuse between v_i and already scheduled nodes both in the same step (s) and previous steps. The weight value

Algorithm 1 $calcSchedule(p, \alpha, \beta, flag)$

```
procedure CALCSCHEDULE(p, α, β, flag)
    for all v ∈ V do
        Step_v ← 0
        sRemain ← sRemain ∪ {v}
    end for
    cs ← 0                                              ▷ current step
    ASAP()
    ALAP()                                  ▷ Assign ASAP/ALAP values to each node
    sReady ← ∅                                          ▷ initialize ready nodes set
    insertReadyNodes(1)
    while (sReady ≠ ∅) do
        cs ← cs + 1                                     ▷ next execution step
        Sch_cs ← ∅
        cap ← p                                         ▷ remaining capacity in current step
        doMandatory(cs)
        while (cap > 0) and (sReady ≠ ∅) do
            myMax ← −1
            myNode ← null
            for all v ∈ sReady do
                ss ← 0                                  ▷ sReady score
                if (flag and (Sch_cs = ∅)) then
                    ss ← calcScore_sReady(v, p − 1)
                end if
                ts ← calcScore_Sch(v, α, cs) + β ∗ ss
                if ts > myMax then                      ▷ total score
                    myMax ← ts
                    myNode ← v
                end if
            end for
            scheduleNode(myNode, p − cap + 1, cs)
        end while
        insertReadyNodes(cs + 1)
    end while
end procedure
```

Algorithm 2 $insertReadyNodes(s)$

procedure INSERTREADYNODES(s)
  for all node v in $sRemain$ **do**
    if $((S_v \leq s)$ and $(L_v \geq s)$ and $(\forall v_i \in Pred_v.\ Step_{v_i} > 0))$ **then**
      $sReady \leftarrow sReady \cup \{v\}$
      $sRemain \leftarrow sRemain - \{v\}$
    end if
  end for
end procedure

Algorithm 3 $doMandatory(s)$

procedure DOMANDATORY(s)
  for all $v \in sReady$ **do**
    if $(L_v \leq s)$ and $(v \notin sManda)$ **then**
      $sManda \leftarrow sManda \cup \{v\}$
    end if
  end for
  while $(cap > 0)$ and $(sManda \neq \emptyset)$ **do**
    $sManda_0 \leftarrow v \in sManda$ with lowest L_v
    $scheduleNode(sManda_0, p - cap + 1, s)$
    $sManda \leftarrow sManda - \{sManda_0\}$
  end while
end procedure

Algorithm 4 $scheduleNode(v, s)$

procedure SCHEDULENODE(v, s)
  $Sch_s \leftarrow Sch_s \cup \{v\}$
  $Step_v \leftarrow s$
  $sReady \leftarrow sReady - \{v\}$
  $cap \leftarrow cap - 1$
end procedure

Algorithm 5 $calcScore\_sReady(v_i, c)$

```
function CALCSCORE_SREADY(v_i, c)
    k ← 0
    for all (v_j ∈ Loc_{v_i}) do
        if v_j ∈ sReady then
            k ← k + 1
            sReadyLocs[k] ← W_{e_{i,j}}
        end if
    end for
    sort sReadyLocs
    wt ← 0
    for l ← 1, Min(c, k) do
        wt ← wt + sReadyLocs[k − l + 1]
    end for
    return wt
end function
```

returned by $calcScore\_Sch(v_i, \alpha, s)$ is given by the formula:

$$\sum_{v_j \in Loc_{v_i}} [sgn(Step_{v_j}) * W_{e_{i,j}}/(\alpha * (s - Step_{v_j}) + 1)]$$

The signum function in the formula prevents contribution from unscheduled nodes, and is defined as follows:

$$sgn(x) = \begin{cases} -1 & : \quad x < 0 \\ 0 & : \quad x = 0 \\ 1 & : \quad x > 0 \end{cases}$$

Different α values change the effect of step difference (e.g., α=0 ignores execution steps and uses the locality value directly, whereas a large α value concentrates on the current execution step).

As explained earlier, in our CMP architecture, we assume that the highest level of memory for data is the on-chip shared cache, and there are no private on-chip data caches. However, we can also accommodate the case where the processors also have private on-chip data caches by making a small change in our algorithm. Currently, the algorithm considers which step to schedule a block in, but does not care for which particular processor the block is assigned. If there are private on-chip data caches attached to processors, the modified algorithm will also try to assign blocks with data reuse to the same processor in consecutive steps, so that the majority of data requests can be satisfied from the on-chip private data cache.

Algorithm 6 $calcScore\_Sch(v_i, \alpha, s)$

function CALCSCORE_SCH(v_i, α, s) ▷ Check weights with all scheduled items and consider the step distance
  $wt \leftarrow 0$
  for all $v_j \in Loc_{v_i}$ **do**
    if $Step_{v_j} > 0$ **then**
      $wt \leftarrow wt + W_{e_{i,j}}/(\alpha * (s - Step_{v_j}) + 1)$
    end if
  end for
  return wt
end function

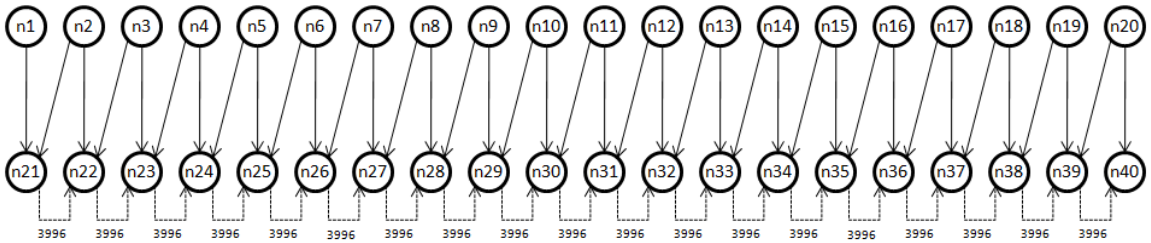


Figure 3.3: LPG of the adi benchmark.

3.4.2 Example

As an example we consider the case of adi benchmark (one of our benchmarks used in this study) when using 4 processors. This benchmark implements alternate directed integration, a frequently used approach. In this case, the adi benchmark code was broken into 40 code blocks, and Figure 3.3 illustrates the corresponding LPG. The schedules obtained by the heuristic solution and the ILP solution (to be presented shortly) are given in Table 3.2. Each line shows the nodes executed in that execution step. The weighted data reuse score for these schedules are 16.33 for the schedule returned by our heuristic approach, and 17 for the schedule returned by the ILP solver (which represents the optimal case). These scores are calculated using the formula given in Algorithm 6 with α=1. The heuristic schedule executed in 51.4 seconds, while the ILP schedule executed in 47.2 seconds. The details of the ILP based formulation will be given shortly.

3.4.3 Discussion

We want to re-iterate here is that in our approach LPG is built by the compiler. That is, using the compiler (gcc in our case) we extract data dependencies among code blocks and construct the LPG for the application code. As stated earlier, our scheduling strategy operates on this LPG. Note however that our proposed scheduling strategy is actually independent of how the input LPG is built. This brings us to the other point we want to

Table 3.2: adi schedule with 4 processors.

Heuristic Approach	**I**LP Based Approach
n11—n07—n10—n06	n01—n02—n03—n04
n09—n13—n14—n08	n05—n06—n07—n08
n16—n02—n12—n01	n09—n10—n11—n12
n17—n15—n20—n05	n13—n14—n15—n16
n19—n04—n18—n03	n17—n18—n19—n20
n30—n29—n28—n27	n21—n22—n23—n24
n26—n25—n24—n23	n25—n26—n27—n28
n22—n21—n31—n32	n29—n30—n31—n32
n33—n34—n35—n36	n33—n34—n35—n36
n39—n38—n40—n37	n37—n38—n39—n40

Table 3.3: Constants used in our ILP formulation.

Constant	**Definition**
p	number of processors
$W_{e_{i,j}}$	The weight of edge $e_{i,j}$
S_{v_i}	ASAP value of operation v_i
L_{v_i}	ALAP value of operation v_i

make. In case an application code is not fully analyzable by the compiler, the user can build an LPG (and in fact each node of this LPG can be an arbitrary computation as long as the computation workloads across different code blocks are more or less balanced) if she can extract data dependencies based on application-level information she has. The resulting LPG can then be fed to our scheduler for schedule generation.

3.5 ILP Formulation of the Problem

In order to see how close our heuristic comes to the optimal, we also implemented an ILP (Integer Linear Programming) based solution to the problem, and performed experiments with it. This section gives the details of our ILP based solution. Table 3.3 lists the constant terms used in our ILP formulation. We used LPSolve [76], a public-domain ILP tool, to formulate and solve our 0-1 ILP problem. Although ILP generates an optimal result (under the assumptions made), the time complexity of ILP prohibits practical usage in most cases. The computation of the solutions for the ILP problems mentioned below took days on average and more than a week in one case. Therefore, it is very important to explore heuristic solutions for this combined parallelism-data locality problem.

A 0-1 ILP problem is a special kind of ILP problem, where each variable can only take the value of 0 or 1. In our case, we have only one type of 0-1 solution variable, $X_{i,l}$, which indicate whether v_i is scheduled on step l. To make our presentation clear, we use the expression $Step_{v_i}$ to represent the execution step that the code block v_i is scheduled. $Step_{v_i}$ is expressed in terms of the $X_{i,l}$ variables and ILP constants as follows:

$$Step_{v_i} = \sum_{l=S_{v_i}}^{L_{v_i}} (l * X_{i,l}).$$

3.5.1 Objective Function

Our objective is to find the execution step that each node is to be scheduled, such that, nodes that exhibit high data reuse with each other are scheduled as close to each other as possible. In our formulation, the 0-1 variables $X_{i,l}$ are used to capture this information. In other words, we want to minimize the scheduling step distance between nodes with data locality. Therefore, we can express our objective function as follows:

$$\text{Minimize} \sum_{e_{i,j}} (|Step_{v_i} - Step_{v_j}| * W_{e_{i,j}}),$$

where $e_{i,j} \in E_{loc}$ (e.g. $e_{i,j}$ is a locality edge).

3.5.2 Constraints

We have three types of constraints in our problem.

1. The execution step $Step_{v_i}$ for code block v_i is unique for all $i \in \{1, ..., n\}$. As a result, for a given i, only one of the $X_{i,l}$ variables will take a value of 1, and the rest will be 0, which can be formulated as follows:

$$\sum_{l=S_{v_i}}^{L_{v_i}} (X_{i,l}) = 1.$$

   For example, if S_{v_1} = 3 and L_{v_1} = 5, then $X_{1,3} + X_{1,4} + X_{1,5} = 1$ is a constraint in our ILP formulation.

2. Sequencing relations must be satisfied. If a code block v_j depends on v_i, then v_j should be scheduled at a later step than v_i. This constraint can be expressed as follows: $Step_{v_j} > Step_{v_i}$, where $e_{i,j} \in E_{dep}$ (e.g., $e_{i,j}$ is a dependency edge).

3. Since we have p processors, at most p code blocks can be scheduled at an execution step. In other words, for any given step l, the sum of $X_{i,l}$ values cannot exceed p. Note that the actual number of nodes scheduled at a step can be less than p due to data dependencies between code blocks. Therefore,

Table 3.4: Benchmark codes used in our experimental evaluation.

Benchmark Name	Number of Nodes
adi	40
bmcm	23
tsf	26
vpenta	28

for each step l, we include the following constraint in our formulation:

$$[\sum_{i \in \{j | S_{v_j} \leq l \leq L_{v_j}\}} X_{i,l}] \leq p.$$

The above mentioned constraints and objective function constitute our ILP formulation of the problem. In this formulation, the nodes that do not exhibit data reuse are not part of the objective function, and are therefore scheduled based solely on data dependencies, or arbitrarily if they have none.

3.6 Experiments

We implemented the algorithm explained in this chapter as a software tool which takes an LPG, as well as α, β and the number of processors in the CMP as input parameters. The tool parses the graph and applies our heuristic algorithm with the given parameters to obtain the data locality-optimized parallel execution schedule.

3.6.1 Setup

We used four data-intensive, array-dominated benchmarks to test our algorithm, and performed experiments on three hardware platforms (with 2, 4 and 8 processors.) Table 3.4 lists the benchmark codes we used and the number of nodes in their LPGs. Table 3.5 on the other hand lists the key properties of the hardware platforms used in our tests. For our tests, we simulated the hardware and OS using Simics [108]. Simics is a simulation toolset for multi-processor systems and allows building a binary-compatible instance of the target hardware, which operates completely within a virtualized environment running on standard PCs.

For each of the hardware platforms, the following operations were performed. We applied our algorithm to the LPG of each benchmark. The above mentioned ILP formulation of each of these problems produced an alternate schedule. We used the Intel C++ Compiler 10 [48] and OpenMP API [95] to compile the codes

Table 3.5: Key parameters of our experimental platform

Number of Processors	Cache Size	Memory Capacity
2	128KB	256MB
4	256KB	256MB
8	512KB	256MB

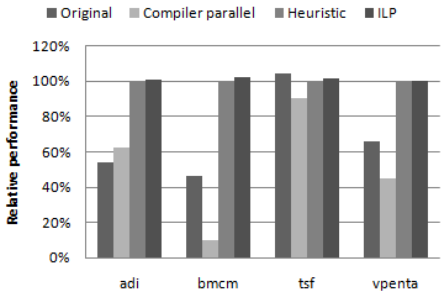


Figure 3.4: Results for the 2 processor case.

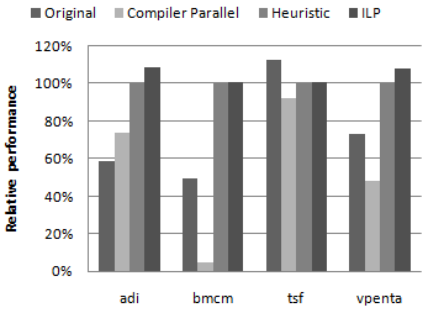


Figure 3.5: Results for the 4 processor case.

resulting from these schedules (for both heuristic scheme and ILP solver based scheme). We also compiled the original code both without parallelization and using the Intel compiler's own parallelization mechanism. For our tests, the default values of $\alpha = \beta = 1$ were used. We also made experiments with other values of the α and β parameters, but the results were very close to those shown here for the $\alpha = \beta = 1$ case. More specifically, the difference between the result obtained with different α, β values were within 4%.

3.6.2 Results

On each platform, we obtained four results for each benchmark. Figures 3.4, 3.5 and 3.6 show the results of the experiments with 2, 4 and 8 processors, respectively. The bars show the speeds *normalized* with respect to the result given by our heuristic algorithm. The normalized value for each version is computed as the ratio

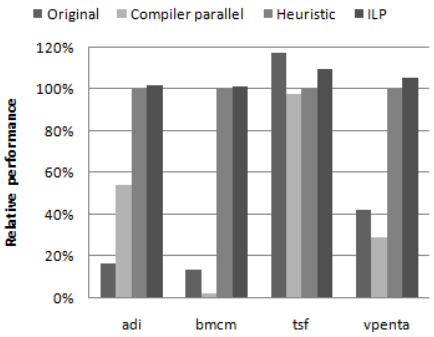


Figure 3.6: Results for the 8 processor case.

of the heuristic result's execution time to that version's execution time. The four bars for each benchmark in the figures represent the performances achieved by the following scheduling schemes:

1. **Original**: Original program with no scheduling. This represents the sequential code without exploiting loop level parallelism.
2. **Compiler Parallel**: The original code compiled by using the compiler's [48] own parallelization mechanism. While we obviously do not know the details of this parallelization strategy, it is reasonable to assume that it represents state-of-the-art in industry.
3. **Heuristic Parallel**: The code scheduled by our heuristic scheduling approach (Section 4).
4. **ILP**: The code scheduled based on the solution returned by the ILP based formulation (Section 5).

Our first observation is that, our algorithm performs better than the original in all but one of the benchmarks. This exception is due to that benchmark having a small body of code, which makes the synchronization overhead brought by parallelization significant. We also note that our scheduling algorithm performs better than the compiler's own parallelization mechanism in all cases, and is very close to the result achieved by the ILP solver. The overall speedups achieved by our algorithm with respect to the original code are 1.62, 1.50 and 4.21 for the 2, 4 and 8 processor cases, respectively. By comparison, the overall speedups achieved by the ILP based solution with respect to the original code are 1.63, 1.54 and 4.29 for the 2, 4 and 8 processor cases, respectively. As stated earlier however, ILP based scheduling may not be a viable option in some cases due to enormous solution times it requires.

3.7 Conclusion

Increasing use of chip multiprocessors in embedded computing domain makes automated software support a primary concern for programmers. In particular, compiler plays an important role since it shapes the code

behavior as well as data access pattern. Targeting chip multiprocessors and loop-intensive computations, we propose a novel compiler-based loop scheduling scheme with the goal of exploiting both parallelism and locality. In this chapter, we describe our strategy and evaluate it using a set of four application codes. The schedules generated by our algorithm are compared to those obtained by an ILP based scheduler and the original codes. The experimental results we collected are promising and indicate that our approach achieves better results than the commercial compiler and the improvements we obtain are close to those obtained using the ILP solver based scheduler.

In this chapter, our focus was on CMP architectures with a shared cache. We will be focusing on CMP systems that have a shared SPM in the next chapter.

Chapter 4

SPM Conscious Static Loop Scheduling

4.1 Introduction

Developing loop parallelization strategies that take into account memory behavior can be very useful in practice. In this context, parallelization is distribution of loop iterations across available parallel processors. Motivated by this observation, in this chapter we propose a novel static loop scheduling strategy for array/loop based applications. We target CMP architectures with a shared SPM (i.e. an SPM that is shared by all processors). Figure 1.2 shows the high level architecture of such a system.

This strategy tries to achieve two objectives. First, the sets of loop iterations assigned to different processors should approximately take the same amount of time to finish. Second, the set of iterations assigned to a processor should exhibit high data reuse. Satisfying these two objectives help us to minimize parallel execution time of the application at hand. The specific method adopted by our scheduling strategy to achieve these objectives is to distribute loop iterations across parallel processors in an SPM conscious manner. In this strategy, the compiler analyzes the loop, identifies the potential SPM hits and misses, and distributes loop iterations over processors such that the processors have more or less the same execution time. The technique presented here is not bound to any particular SPM management scheme.

We implemented the proposed approach using an optimizing compiler and conducted experiments with six array/loop based embedded applications. Our results so far indicate that the proposed approach generates much better results than existing loop schedulers. Specifically, it brings 18.9%, 22.4%, and 11.1% improve-

ments in parallel execution time (with a chip multiprocessor of 8 cores) over a previously proposed static scheduler, dynamic scheduler, and locality-conscious scheduler, respectively.

The rest of this chapter is organized as follows. The next section discusses related work. Section 4.3 presents the details of our loop scheduling strategy. An experimental evaluation of the proposed scheduler as well as its comparison against several other schedulers are given in Section 4.4. Finally, we conclude in Section 4.5 by a summary.

4.2 Related Work

While there have been a plethora of studies in the context of code and data optimization for loop based applications, in this section we discuss only the most related work on loop level scheduling. Loop-level scheduling can be performed in two different ways: *static* or *dynamic*. In static scheduling, the iteration space is partitioned across the processor prior to execution. Prior static scheduling algorithms include [63, 79]. Note that static scheduling can be made locality aware by ensuring that the set of iterations assigned to a processor exhibit data reuse, as in the case [63]. While static scheduling is easy to implement, it may suffer from load imbalance when the processors complete their assignments at different times. To remedy this problem, dynamic scheduling algorithms [100, 111, 125, 55, 68] have been proposed. The main idea behind these algorithms is to perform workload assignment at runtime. In this way, a processor that finishes its current assignment early can get a new assignment. To our knowledge, all of the implemented dynamic algorithms in the literature are centralized (i.e., master based). That is, a single master performs workload distribution at run- time across the processors. While these algorithms solve the load imbalance problem and can be made locality aware (as shown in [55, 68]), they can incur performance bottle- necks at runtime. Also, master processor can be a point of contention and entire execution fails if master becomes dysfunctional. In contrast to the prior work on loop level scheduling, in this chapter, we focus on an SPM based system and propose an SPM conscious scheduling algorithm that is locality aware. The proposed approach is different from the prior static algorithms since it allows dynamic workload distribution. It is also different from the previous dynamic scheduling techniques since it exploit the SPM contents as much as possible.

4.3 Our Approach

4.3.1 Parallel and Sequential Loops

Our focus is on array-based loop-intensive parallel applications. Such an application typically consists of a set of loop nests accessing a set of data arrays. In this chapter, we consider two types of loops: parallel loops and sequential loops. All the iterations of a parallel loop can be executed in parallel, while the iterations of a sequential loop must be executed sequentially. Sequential loops are used where the loop iterations cannot be executed in parallel due to data and control dependencies. We denote a parallel loop as:

forall $i \in [L, U] : \{\mathcal{B}\}$,

where i is the index variable, $[L, U]$ is the iteration region (iteration space), and B is the body of the loop. On the other hand, a sequential loop is denoted using the following form:

for $i \in [L, U] : \{\mathcal{B}\}$.

We assume that all the sequential loops have been normalized, i.e., index variable i iterates from the lower bound L to the upper bound U with increment 1 at each step.

The iterations of a parallel loop are distributed among a set of processors so that they can be executed in parallel. This is performed by a parallelizing compiler. Specifically, the compiler partitions the iteration space (i.e., the set of loop iterations) of each parallel loop into a group of sub- spaces; the loop iterations that belong to different subspaces are assigned to different processors. In order to guarantee the correctness of the execution, the compiler implicitly inserts a barrier to the end of each parallelized loop such that the code following this loop cannot be executed until all the iterations of this loop, which are distributed over multiple processors, are completed. Due to the overheads incurred by barriers, parallelizing a loop at a very small granularity is usually not desirable. A loop nest can contain multiple loops; and each loop of a given nest can be either parallel or sequential. While all the parallel loops in a loop nest can be parallelized if desired, in order to minimize the synchronization overheads due to the barriers, for each loop nest, we parallelize only the parallel loop that is not enclosed by any other parallel loops. All the iterations of a parallel loop that is not parallelized are executed on the same processor. Since there are no data dependencies among the iterations of a parallel loop, the compiler can reorder the execution of these loop iterations to improve data locality or for any other target optimization metric.

4.3.2 Example

A given parallel loop can be partitioned across processors in different ways and each of these potential partitionings typically yields a different performance. In particular, assigning the same number of loop iterations to all processors does not guarantee load balance. **Figure 4.1** illustrates this point using an example. **Figure 4.1(a)** shows a code fragment that consists of two parallel loops $\mathcal{L}_1$ and $\mathcal{L}_2$. We assume, for illustrative purpose, that the system contains two processors, p_1 and p_2. A parallelizing compiler partitions loop $\mathcal{L}_1$ into $\mathcal{L}_{1,1}$ and $\mathcal{L}_{2,1}$, and loop $\mathcal{L}_2$ into $\mathcal{L}_{1,2}$ and $\mathcal{L}_{2,2}$, as shown in **Figure 4.1(b)**. Let us further assume that loops $\mathcal{L}_{1,1}$ and $\mathcal{L}_{1,2}$ are executed on processor p_1, and loops $\mathcal{L}_{2,1}$ and $\mathcal{L}_{2,2}$ are executed on processor p_2. Before entering loops $\mathcal{L}_{1,1}$ and $\mathcal{L}_{2,1}$, processors p_1 and p_2 load, respectively, B[0..49] and B[50..99] - the regions of array B that will be respectively accessed by loops $\mathcal{L}_{1,1}$ and $\mathcal{L}_{2,1}$ - into the SPM. Since we do not remove data from the SPM during the execution of loop nests $\mathcal{L}_{1,1}$ and $\mathcal{L}_{2,1}$, at the synchronization point (marked as "barrier") following loops $\mathcal{L}_{1,1}$ and $\mathcal{L}_{2,1}$, the SPM contains array elements in the region B[0..99].

The executions of loops $\mathcal{L}_{1,2}$ and $\mathcal{L}_{2,2}$ follow those of loops $\mathcal{L}_{1,1}$ and $\mathcal{L}_{2,1}$. Since the array elements in B[0..99] are used by loop $\mathcal{L}_{1,2}$, we keep them in the SPM. Let us assume that the SPM can hold only 100 elements of array B, and thus, we cannot load any additional elements of array B into the SPM. As a result, loop $\mathcal{L}_{2,2}$ has to access elements of array B from the off-chip memory. On the other hand, all the array elements that are used by loop $\mathcal{L}_{1,2}$ are already in the SPM. Since on-chip SPM can be accessed much faster than the off-chip memory, the execution time of $\mathcal{L}_{1,2}$ is much shorter than that of $\mathcal{L}_{2,2}$. When processor p_1 completes its execution of $\mathcal{L}_{1,2}$, it has to wait in idle until processor p_2 finishes with the iterations in $\mathcal{L}_{2,2}$. This example clearly shows that a load imbalance across processors can easily hurt parallel execution time.

Let us assume that the average per iteration execution time of loop $\mathcal{L}_{i,j}$ (i =1, 2, and j =1, 2) is $T_{i,j}$. Since $\mathcal{L}_{1,2}$ accesses array B from the SPM while $\mathcal{L}_{2,2}$ accesses array B from the off-chip memory, we have $T_{1,2} < T_{2,2}$. The difference between $T_{1,2}$ and $T_{2,2}$ is determined by the difference in access latencies of the SPM and the off-chip memory. The overall execution time of the code shown in **Figure 4.1(b)** can be estimated as:

$T = \max\{50T_{1,1}, 50T_{2,1}\} + \max\{100T_{1,2}, 100T_{2,2}\}$

$= 50\max\{T_{1,1}, T_{2,1}\} + 100T_{2,2}$

In order to minimize the overall execution time of the application, we need to balance the execution time of processors p_1 and p_2. That is, after the execution of loop $\mathcal{L}_{1,2}$, processor p_1 can execute some iterations of loop $\mathcal{L}_{2,2}$, instead of waiting in idle for the completion of p_2. The number of the iterations of loop $\mathcal{L}_{2,2}$

that need to be moved from p_2 to p_1 to achieve the best balance can be computed as:

$N = 100(T_{2,2} - T_{1,2})/(2T_{2,2})$.

Based on this observation, we can rewrite the code of loops $\mathcal{L}_{1,2}$ and $\mathcal{L}_{2,2}$, as shown in **Figure 4.1(c)**. After this code rewriting, the overall parallel execution time can be estimated as:

$T^* = 50 \max\{T_{1,1}, T_{2,1}\} + 50(T_{1,2} + T_{2,2})$.

Since $T_{1,2} < T_{2,2}$,we have:

$T^* < 50 \max\{T_{1,1}, T_{2,1}\} + 50(T_{2,2} + T_{2,2}) = T$.

That is, through redistribution of the iterations of parallel loop $\mathcal{L}_2$, we can reduce the overall execution time of this loop.

4.3.3 Problem Formulation

Our goal is to determine a partitioning of a given parallel loop such that the overall parallel execution time of the loop is minimized. This problem can be formulated as follows. Given the number of processors, p, and a parallel loop with iteration space $\mathcal{I}$ of the following form:

forall $i \in \mathcal{I} : \{\mathcal{B}\}$,

determine a partitioning for $\mathcal{I}$ as:

$P(\mathcal{I}) = \{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_p\}$,

where $\mathcal{I}_1 \cup \mathcal{I}_2 \cup ... \cup \mathcal{I}_p = \mathcal{I}$ and $\mathcal{I}_i \cap \mathcal{I}_j = \emptyset$ for all $i \neq j$, such that the value of T , which can be computed using the following expression, is minimized:

$T = \max\limits_{\mathcal{J} \in P(\mathcal{I})} T(\mathcal{J}, \mathcal{B})$,

where $T(\mathcal{J}, \mathcal{B})$ is the execution time of the following loop:

for $i \in \mathcal{J} : \{\mathcal{B}\}$.

Since the total time taken by all the loop iterations combined, i.e., the value of $\sum_{\mathcal{J} \in P(\mathcal{I})} T(\mathcal{J}, \mathcal{B})$, is a constant, it is not difficult to prove that, when the execution times of the p processors are balanced, that is, when

load B[0..99] into SPM;
$\mathcal{L}_1$:forall $i \in [0, 99]\{\ldots B[i] \ldots\}$
$\mathcal{L}_2$:forall $i \in [100, 199]\{\ldots B[i] \ldots\}$

(a) A code fragment with two loops.

Processor p_1	**Processor** p_2
load B[0..49] into SPM;	load B[50..99] into SPM;
$\mathcal{L}_{1,1}$:for $i \in$ [0, 49]	$\mathcal{L}_{2,1}$:for $i \in$ [50, 99]
$\ldots B[i] \ldots$	$\ldots B[i] \ldots$
— barrier—	
$\mathcal{L}_{1,2}$:for $i \in$ [0, 99]	$\mathcal{L}_{2,2}$:for $i \in$ [100, 199]
$\ldots B[i] \ldots$	$\ldots B[i] \ldots$
— barrier—	

(b) Unbalanced parallelization over processors p_1 and p_2.

Processor p_1	**Processor** p_2
load B[0..49] into SPM;	load B[50..99] into SPM;
$\mathcal{L}_{1,1}$:for $i \in$ [0, 49]	$\mathcal{L}_{2,1}$:for $i \in$ [50, 99]
$\ldots B[i] \ldots$	$\ldots B[i] \ldots$
— barrier—	
$\mathcal{L}'_{1,2}$:for $i \in$ [0, 49 + N]	$\mathcal{L}'_{2,2}$:for $i \in$ [50 + N, 199]
$\ldots B[i] \ldots$	$\ldots B[i] \ldots$
— barrier—	

(c) Balanced parallelization over processors p_1 and p_2.

Figure 4.1: Motivation Example

$T(\mathcal{I}_1, \mathcal{B})$ = $T(\mathcal{I}_2, \mathcal{B})$ = ... = $T(\mathcal{I}_p, \mathcal{B})$,

we achieve the minimum value of T as:

$$T^* = \frac{1}{p} \sum_{\mathcal{J} \in P(\mathcal{I})} T(\mathcal{J}, \mathcal{B}).$$

In other words, in order to minimize the execution time of a parallel loop, we need to find a balanced partitioning of the iteration space of this loop.

4.3.4 Estimating Loop Execution Time

It should be noted that the body of a parallel loop may contain multiple sequential loop nests. For clarity of discussion, let us assume that $\mathcal{B}$, the body of the parallel loop in question, has the a structure of the following form:

for $i_1 \in [l_1, u_1]$
  for $i_2 \in [l_2, u_2]$
    ......
      for $i_m \in [l_m, u_m]\{$
        $\ldots X_1[f_1(i_0, i_1, i_m)] \ldots;$
        $\ldots X_2[f_2(i_0, i_1, i_m)] \ldots;$
        ......
        $\ldots X_n[f_n(i_0, i_1, i_m)] \ldots;$
      $\},$

where i_0 is the index variable for the parallel loop, X_r (where $1 \leq r \leq n$) is the name of an array, and function f_r maps the iteration vector $(i_0, i_1, ..., i_n)^T$ to the subscript vector of array X_r. We require $f_r(i_0, i_1, ..., i_n)$ be an affine function of iteration vector $(i_0, i_1, ..., i_n)^T$. For processor k, we have $i_0 \in \mathcal{I}_k$,where $\mathcal{I}_k$ is the set of the iterations of the parallel loop in question that are assigned to processor k. Based on this assumption, $T(\mathcal{I}_k, \mathcal{B})$, the total execution time due to the loop iterations that are assigned to processor k can be computed as:

$$T(\mathcal{I}_k, \mathcal{B}) = N_1 C_{datapath} + N_2 C_{SPM} + N_3 C_{memory},$$

where N_1 is the number of times that the body of innermost loop is executed, and N_2 is the number of accesses to the SPM, N_3 is the number of accesses to the off-chip main memory, $C_{datapath}$ is the number computation cycles spent by the processor core, C_{SPM} is the access latency for the SPM, and C_{memory} is the access latency for the main memory. For a typical array-intensive application, memory accesses dominate the execution time of the entire loop; therefore, we can omit the time due to the computation performed by the processor core. Consequently, we have:

$$T(\mathcal{I}_k, \mathcal{B}) \approx N_2 C_{SPM} + N_3 C_{memory}.$$

The values of C_{memory} and C_{SPM} are determined by the implementation of the system hardware; and the values of N_2 and N_3 can be computed as discussed below.

Let us first compute N, the number of total memory (including both the SPM and the main memory) accesses as:

$$N = M(\mathcal{B})|\mathcal{I}_k| \prod_{r=1}^{m}(u_r - l_r + 1),$$

where $M(\mathcal{B})$ is the number of memory access instructions in loop body $\mathcal{B}$; and $|\mathcal{I}_k|$, the size of set $\mathcal{I}_k$, gives the number of the iterations of the parallel loop that are assigned to processor k. In this equation, $|\mathcal{I}_k| \prod_{r=1}^{m}(u_r - l_r + 1)$ gives the number of times that loop body $\mathcal{B}$ is executed.

The number of SPM accesses, N_2, can be computed as:

$$N_2 = \sum_{r=1}^{n} |S_r|,$$

where S_r is the set of the iterations of the loop nest where the memory accesses made by $X_r[f_r(i_0, i_1, \ldots, i_m)]$ can be satisfied by the contents of SPM. S_r can be computed as:

$$\begin{aligned} S_r = \{(i_0, i_1, \ldots, i_n)^T | i_0 \in \mathcal{I}_k \text{ and} \\ \forall q = 1, 2, \ldots, n : l_q \leq i_q \leq u_q \text{ and} \\ X_r[f_r(i_0, i_1, \ldots, i_m)] \in D_{SPM}\}, \end{aligned}$$

where D_{SPM} is the set of array elements that are stored in the SPM, which is determined by the SPM management strategy. The size of set S_r can be computed using the techniques presented in [25, 59]. Finally, we can compute the number of the main memory accesses as:

$$N_3 = N - N_2.$$

4.3.5 Loop Partitioning Algorithm

In this section, we discuss our loop partitioning algorithm. Specifically, given the number of processors p and a parallel loop of the following form:

forall $i \in [0, M] : \{\mathcal{B}\}$,

our algorithm determines the partitioning points $P_1, P_2, ...P_{p-1}$ such that:

Input:
p: number of processors;
Parallel loop "forall $i \in [0, M] : \{\mathcal{B}\}$"
Output:
Partitioning points $P_1, P_2, \ldots, P_{p-1}$ such that
$T([0, P_1 - 1], \mathcal{B}) \approx T([P_1, P_2 - 1], \mathcal{B}) \approx \cdots \approx T([P_{p-1}, M], \mathcal{B})$.
$T^* = T([0, M], \mathcal{B})/p$.
$P_0 = 0$;
for i =1 to $p - 1$
P_i = partition($[P_{i-1}, M], T^*$)

function partition($[L, U], T^*$) {
$a = L; b = U$;
while ($a < b$) {
$t = (a + b)/2$;
if ($T([L, t], \mathcal{B}) < T^*$)$a = t$;
else if ($T([L, t], \mathcal{B}) > T^*$)$b = t$;
else return t;
}
return a;
}

Figure 4.2: Loop partitioning algorithm.

$$0 < P_1 < P_2 < ... < P_{p-1} < M,$$

and

$$T([0, P_1 - 1], \mathcal{B}) \approx T([P_1, P_2 - 1], \mathcal{B}) \approx T([P_2, P_3 - 1], \mathcal{B}) \approx \ldots$$
$$\cdots \approx T([P_{p-1}, M], \mathcal{B}) \approx \frac{T([0,M],\mathcal{B})}{p}.$$

Figure 4.3.5 shows the details of our algorithm. The algorithm first estimates T, the execution time of the given parallel loop if the entire load is distributed to the p processors evenly, as follows:

$$T^* = T([0, M], \mathcal{B})/p.$$

After that, it determines the first partitioning point P_1 by binary searching $[L, U]$ - the entire iteration space of the given parallel loop. Partitioning point P_i,where $2 \leq i \leq q$, however, is determined by binary searching the iteration space $[P_{i-1}, U]$. The computational complexity of our algorithm is $O(p \log_2 M f(\mathcal{B}))$,where $f(\mathcal{B})$ is the computational complexity of the static analysis algorithm used for estimating $T(\mathcal{J}, \mathcal{B})$.

4.4 Experiments

Our goal in this section is to experimentally evaluate the effectiveness of our SPM conscious loop scheduling strategy. While the primary metric of evaluation we employ is execution cycles, we also expect our approach to bring (leakage) energy benefits as well, as a side effect of reducing the parallel execution time. We collected our results using a simulation platform based on Simics [108], a multiprocessor simulator. Simics is a system-level instruction set simulator, capable of simulating target (uniprocessor and multiprocessor) systems with sufficient fidelity and speed to boot, and run operating systems and commercial workloads. It models several processor types and associated peripheral devices. The simulated chip multiprocessor has 8 embedded cores; each can issue and execute two instructions at each cycle. Processors are assumed to share an on-chip SPM space, which is 32KB in our default configuration. The off-chip memory access latency is assumed to be 90 cycles.

We implemented and simulated five different scheduling schemes:

- Default: This is a conventional static loop scheduling scheme that partitions the set of iterations across processors statically at compile time. Each processor is assigned a consecutive set of Q/P iterations, where Q is the number of loop iterations in the nest and P is the number of processors.

- Dynamic: This is a dynamic loop scheduling strategy that works with fixed chunk size. This approach starts by assigning a set of iterations to processors. When a processor finishes its current set, it is given a new set, in an attempt to fix load imbalance.

- Locality Aware: This is a locality-aware static scheduling technique proposed by Markatos et al [79]. This algorithm, like ours, attempts to balance the workload, minimize the number of synchronization operations, and exploit processor affinity. The idea is to ensure that a loop iteration is always assigned to the same processor. After the first execution of the iteration, the processor will contain the required data, so subsequent executions of the iteration will not need to load data into the SPM.

- SPM-Conscious: This is the SPM conscious loop scheduling strategy we propose.

We collected six different real-life embedded applications to evaluate the loop scheduling schemes described above. Table 4.1 lists the important characteristic of these applications. The main reason that we focus on these applications is that they represent a range of opportunities for addressing data locality, load

Benchmark	Description	Data Size	Execution Time
AC	Adjoint Convolution Computation	0.82 MB	147.5 ms
GE	Gaussian Elimination	1.13 MB	218.9 ms
FLT	Particle Filtering for Wireless Communication	1.89 MB	305.7 ms
SEG	Parallel Image Segregation	1.52 MB	273.3 ms
LRR	Link Reversal Routing	0.67 MB	121.8 ms
AGM	Adaptive Graph Matching	1.36 MB	257.3 ms

Table 4.1: Benchmarks used in this study.

imbalance and synchronizations overheads. The third column of Table 4.1 gives the amount data manipulated by each application in our experimental suite, and the last column gives the total parallel execution time for each application under the default scheme explained above.

Let us first present the maximum potential benefits (in terms of parallel execution time) that could be obtained by being SPM-conscious during loop scheduling. The bar-chart in Figure 4.3 presents three bars for each benchmark code in our experimental suite. The last two bars are normalized with respect to the first one, which represents the default static scheduling where each processor is assigned the same number of iterations and the iterations assigned to a processor are consecutive in the iteration space (recall that these absolute parallel execution times under the default scheme are listed in the last column of Table 4.1). The second bar captures the hypothetical case when all data accesses result in hits in the SPM. Clearly, this is not achievable in practice; but, it is interesting to observe the large savings this could bring. The third bar, which is achievable in principle, represents the perfect load balance among parallel processors. That is, in this case, the hits and misses are equally distributed across processors and for each processor data locality has been exploited as much as possible. Clearly, the third bar for each benchmark represents the lower bound for any SPM latency conscious loop scheduling scheme. We see that such a perfect hit/miss balancing can reduce total parallel execution time by 22.96% on average. Our goal in developing an SPM latency loop scheduling scheme is to approach to this third bar as much as possible.

Figure 4.4 presents the normalized parallel execution times for four different loop scheduling strategies. The first strategy, against which all other schemes are normalized, is the default static one explained above. The second bar on the other hand represents a dynamic loop scheduling scheme (proposed in [111]), where loop iterations are distributed to processors at runtime in an attempt to minimize the load imbalance across processors. When comparing the first two bars, we see that the dynamic scheduling strategy does not help too much, mainly because the load balancing benefits it brings are offset by the overhead of performing partitioning at runtime. These overheads include executing the semaphore-protected loop iteration distribution code executed by the master and restarting the suspended processor with new workload. The third scheme

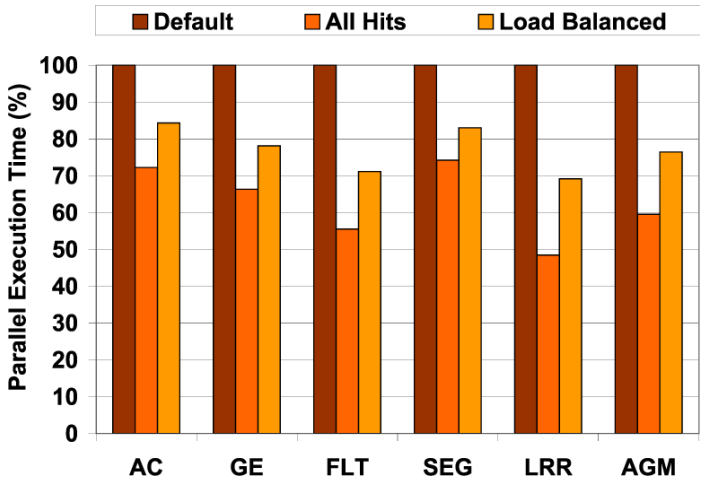


Figure 4.3: The maximum potential benefits that could be obtained by being SPM conscious during loop scheduling.

represents a prior loop scheduling scheme [79] that tries to maintain data locality for a processor as we move from one loop nest to another. Finally, the last bar corresponds to the scheduling strategy proposed in this work. Our first observation is that the average execution time savings brought by the dynamic, locality aware and SPM conscious scheduling schemes are -4.54%, 8.77% and 18.87%, respectively. These results emphasize the importance of balancing hits and misses across parallel processors. Our second, and maybe more important, observation is that the difference between our approach and the perfect load balancing (see the last bar in Figure 4.3 for each benchmark) very small (specifically, 18.87% savings versus 22.96% savings over the default scheduling scheme), meaning that our approach is very successful in exploiting SPM/off-chip latencies during loop scheduling.

We next study the impact of the number of processors used on our savings. In Figure 4.5, we have two bars per benchmark. The first bar (titled LC/LA) gives results of our approach normalized against the results of the prior locality aware scheduling scheme in [79]. The second bar (titled LB/LC) on the other hand gives the results with the perfect load balancing scheme normalized against our scheme. We see from this bar-chart that our approach tracks the optimal load balancing scheme very well; and, the difference between it and the prior locality aware scheme slightly increases as we increase the number of processors (processor count).

Our final set of results investigates the influence of the SPM capacity on the performance improvements we achieve. As in the previous case, we give only the values when averaged over all applications in our suite. An important observation from the results shown in Figure 4.6 is that the difference between our SPM-conscious scheduling strategy and the locality-aware method slightly increases as the SPM capacity is increased. The main reason for this is the fact that a large SPM enables us keep more data elements on-chip,

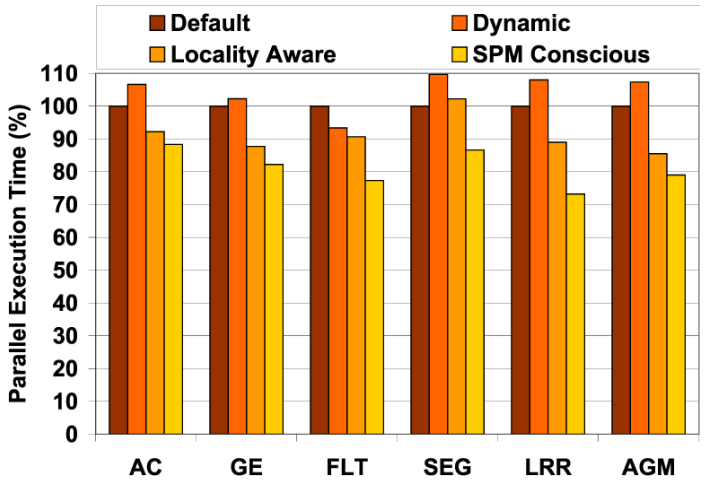


Figure 4.4: Normalized parallel execution times for four different loop scheduling strategies.

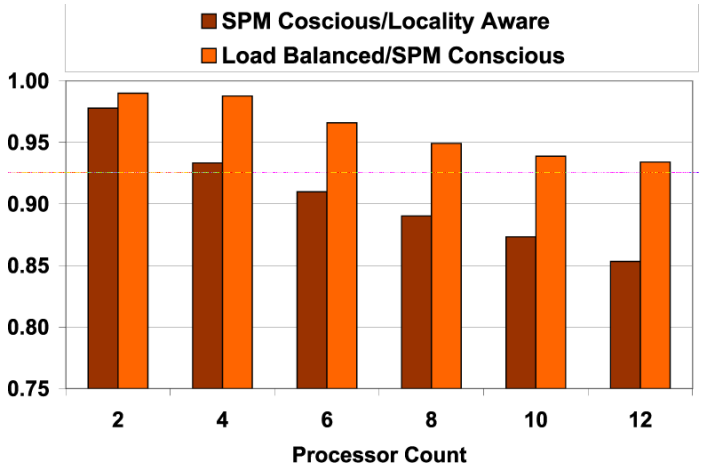


Figure 4.5: Impact of the number of processors.

and this in turn reduces the number of off-chip references and make it even more important to balance the distribution of SPM misses across the parallel processors.

4.5 Conclusion

Increasing use of chip multiprocessor in embedded computing makes automated software support a primary concern for programmers. In particular, compiler plays an important role since it shapes the code behavior as well as data access pattern. We propose a novel compiler-based loop scheduling scheme with the goal of capturing the fact that the different loop iterations can exhibit entirely different SPM (scratch-pad memory)

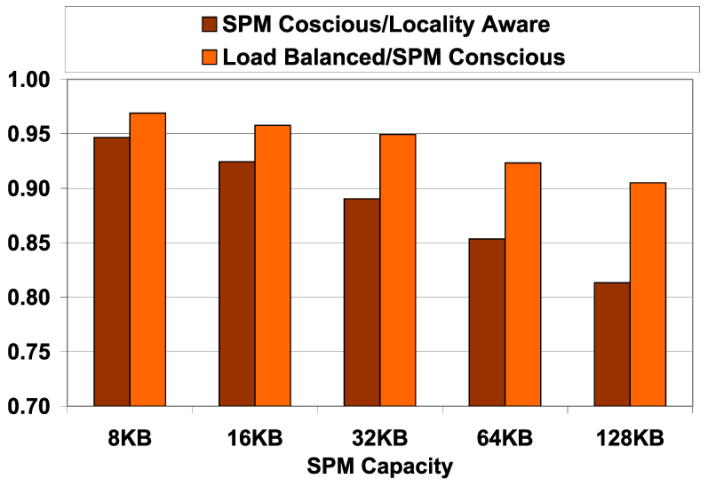


Figure 4.6: Impact of SPM capacity.

behaviors and, consequently, take different number of cycles to finish. Therefore, an SPM conscious loop scheduling, as opposed to just allocating the same number of iterations to each and every processor, can be very useful in practice. We describe our compiler implementation of such a scheduling strategy and compare it to several previously-proposed schedulers using a set of six embedded applications. The experimental results we collected clearly emphasize the importance of SPM conscious loop scheduling.

In this chapter, our focus was on loop scheduling on CMP architectures with a shared SPM. While our approach took into account the presence of the SPM while performing loop scheduling, it did not actually perform SPM management. In the next chapter we will be focusing on SPM management on the same architecture.

Chapter 5

SPM Management Using Markov Chain Based Data Access Prediction

5.1 Introduction

In this chapter, we present a data access prediction scheme and a code restructuring scheme for SPM management. Like the previous chapter, we target CMP architectures with a shared SPM. Figure 1.2 shows the high level architecture of such a system.

While there are numerous publications ([49, 52, 54, 69, 90, 114]) that focus on SPM management for programs with regular array accesses, only a few prior studies have considered irregular accesses. What we mean by ”irregular accesses” are data accesses that cannot be statically resolved at compile time. Two examples of such irregular accesses are illustrated in Figure 5.1. In (a), a pointer is used to access data elements within a loop. Since in general it may not be possible to completely resolve pointer accesses statically, the compiler may not be able to determine which data elements will be accessed at runtime. Similarly, in (b), the set of elements accessed from array A depends on the contents of index array X, which may not be known in general until runtime. In both these cases, it is not possible at compile time to determine the best set of elements to place into the SPM.

However, we want to point out that the lack of static analyzability does *not* necessarily mean lack of locality in data accesses. Consider, for example, Figure 5.1(b) again. Within the main loop of this code fragment (loop t), the same array elements may be reused over and over again. Consequently, based on the contents of this index array, accesses to array A can also exhibit high levels of data reuse, although this is not evident at compile time. To be more specific, assuming N is 20 for illustrative purposes, if the contents

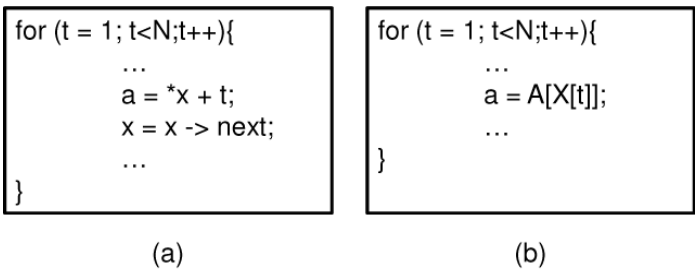


Figure 5.1: Sample irregular access patterns. (a) Irregular access to data by pointers (b) Irregular access to data by indexed array expression.

of array X happen to be {8, 3, 6, 3, 3, 17, 18, 3, 3, 3, 6, 8, 18, 18, 17, 6, 8, 8, 6, 18}, the same five elements of array A ({A[3], A[6], A[8], A[17], A[18]}) are accessed repeatedly by loop t. Therefore, if somehow this pattern can be captured dynamically (during the course of execution), via a compiler inserted code, significant performance gains can be achieved. We present and evaluate a novel approach to this problem. Specifically, targeting data-intensive applications with irregular memory access patterns, we makes the following contributions:

- We propose a Markov Chain (MC) based data access pattern prediction scheme. The goal of this scheme is to predict the next data block to be accessed by execution, given the current data block access.
- We present a compiler-based code restructuring scheme that employs this MC based approach. This scheme transforms a loop into two sub-loops. The first sub-loop forms the training part and is responsible for constructing a MC based memory access pattern prediction model. The second sub-loop is the prefetching part where data is prefetched into the SPM based on the MC based prediction model constructed in the training part.
- We quantify the benefits of this approach using seven data- intensive applications. Five of these applications have irregular data accesses and two have regular data accesses. Our experimental results show that the proposed MC based scheme is very successful in reducing execution time for all seven applications. We also present the results from our sensitivity experiments, and compare our approach to several previously proposed SPM management schemes.

The rest of this chapter is structured as follows. The next section discusses related work. Section 5.3 presents the details of our approach. An experimental evaluation is given in Section 5.4. Finally, we conclude in Section 5.5 by a summary.

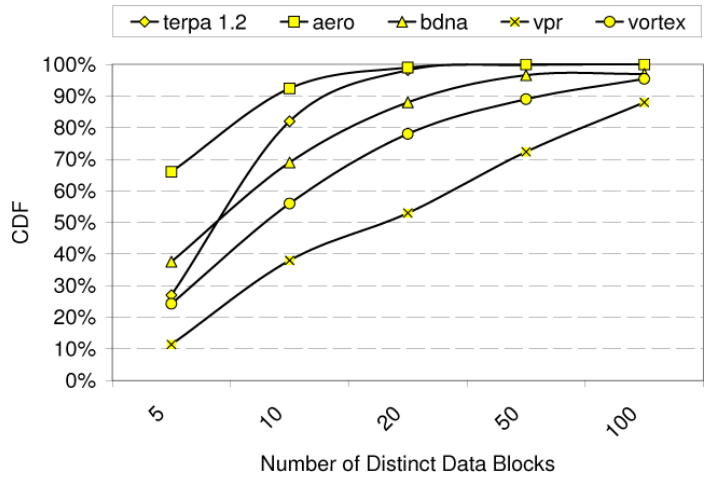


Figure 5.2: CDF of the number of distinct data elements accessed by a reference.

5.2 Related Work

Scratch-pad memories (SPMs) have been widely used in both research and industry, focusing mainly on the management strategies such as static versus dynamic and instruction SPM versus data SPM [99, 10]. Egger et al [32] present a dynamic SPM allocation strategy targeting a horizontally partitioned memory subsystem for processors in the embedded system domain. In [31], authors propose a fully automatic dynamic SPM management technique for instructions, where required code segments are being loaded into the SPM on demand at runtime. Puaut and Pais [103] present an algorithm for off-line selection of the contents of on-chip memories. Li et al [69] employ a compiler-based memory coloring technique to allocate the arrays of a program onto an SPM. Golubeva et al [35] tackle the SPM management problem from a leakage energy perspective. Nguyen et al [90] present an SPM allocation scheme that does not require any compiler support for interpreted-language based applications such as Java. In [30], authors present a compile-time method for allocating heap data to SPM. Nguyen et al [89] discuss an SPM allocation scheme targeting a scenario where SPM capacity is unknown at compile time. This compiler method provides portability to different processor implementations with different SPM sizes.

Some embedded array-intensive applications do not have regular access patterns that can easily be analyzed by static techniques. For such applications, conventional SPM management schemes will fail to produce the best results and will prevent allocating the SPM efficiently [2, 20, 23]. To tackle this problem, Absar et al [2] propose a compiler-based technique for analyzing irregular array-access, and mapping such arrays to the SPM. On the other hand, Chen et al [20] present an approach for data SPMs, where the task of optimization is divided between compiler and runtime. Cho et al [23] present a profiling based technique that generates a memory access trace. This trace, then, is used to identify the data placement within the SPMs.

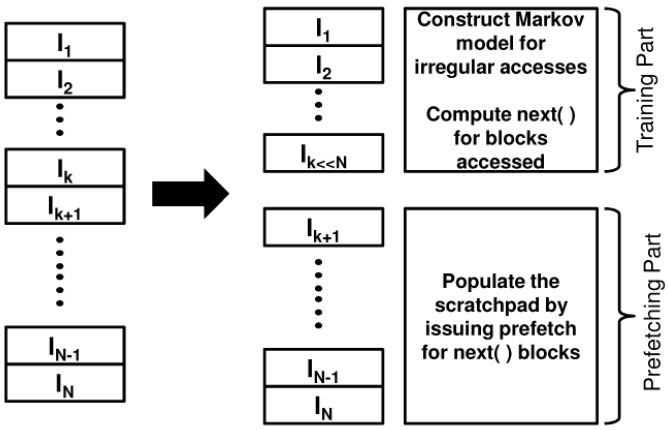


Figure 5.3: High level view of our approach depicting the division of iterations into training part and prefetching part and associated transformation procedures.

While [2] and [20] can handle only irregular accesses due to indexed array expressions, our approach can handle pointer codes as well. Also, as against [23], we do not use profile data, and instead use compiler support to capture runtime behavior and exploit it. Since [2], [20] and [23] are the most relevant prior works, in our experiments we compare our approach to these three approaches.

5.3 Our Approach

5.3.1 Hidden Data Reuse in Irregular Accesses

As stated earlier, the main motivation for our work is the fact that the lack of compile-time analyzability does not necessarily mean lack of locality in data accesses. To quantify this, we collected statistics on five data-intensive applications that are hard to analyze using compile-time techniques alone. The graph in Figure 5.2 plots the CDF of the number of distinct data elements accessed by a reference (when all references are considered). A point (x, y) in this plot indicates that $y\%$ of the accesses made by the reference are to x or fewer distinct data elements. For example, for application vpr, 11.4% of the memory accesses made by a reference are to only 5 distinct data elements, that is, there is significant data reuse per reference. Unfortunately, due to irregular data accesses (i.e., because of the way the code is written), this data reuse cannot be captured and exploited at compile-time.

We propose to use Markov Chains (MC) to capture and optimize such data accesses at runtime. Figure 5.3 shows a high-level view of our approach. For each loop nest of interest, the first few loop iterations are

used to build a Markov Model, which is used to fill a compiler-generated data structure, so that the remaining iterations can take advantage of the available SPM. In the remainder of this section, we present the details of our MC based approach.

We can think of MC as a finite state machine such that if the machine is in state q_i at time i, then the probability that it moves to state q_{i+1} at time $i+1$ depends solely on the current state. In our MC based formulation of the SPM optimization problem for irregular data accesses, each state corresponds to an access to a data block, i.e., a set of consecutive data elements that belong to the same data structure. The weight associated with edge (i, j), i.e., the edge that connects states q_i and q_j , is the probability with which the execution touches block q_j , right after touching data block q_i.

5.3.2 Different Versions

Figure 5.4 gives an example that shows the code transformation performed by our proposed approach. Our approach operates at a loop nest granularity, that is, it is given one loop nest at a time. It divides the given loop nest in two parts (sub-loops). The first part is the *training part* and its main job is to fill a compiler-generated data structure, which is subsequently used in the second part. This data structure represents the MC based model of data accesses encountered in the training part. The second part, called the *prefetching part*, uses this model to issue prefetch requests. Each prefetch request brings a new block to the SPM ahead of time, i.e., before it is actually needed. Therefore, at the time of access, the execution finds that block in the SPM and this helps improve performance and power, though in this work only performance benefits have been evaluated. We can see from Figure 5.4 that the first k iterations ($k << N$) are used for the training part. The remaining iterations are tiled into tiles of t iterations each, and prefetching for the each tile is performed at the beginning of the tile. Selection of t is done such that off-chip memory access latency can be hidden.

We now want to discuss the functionality of next(.). For a given data block B_i, next(B_i) gives the set of blocks that are to be prefetched within the prefetching part. Clearly, there are many different potential implementations of next(.). Below, we summarize the implementations evaluated in this work, using the sample Markov Model illustrated in Figure 5.5:

- A1: It returns only one block which corresponds to the edge in the Markov Model with the highest weight (transition of probability). For example, in Figure 5.5, if the current data block being accessed is B_0, next(.) will return B_2. While this implementation is simple and can be effective in many cases, it may not perform well in every case, as even the highest weight may not be very high. For example, in the same transition diagram, if the current block being accessed is B_6, the prefetched block will be B_9, but, the corresponding probability is only 22%.

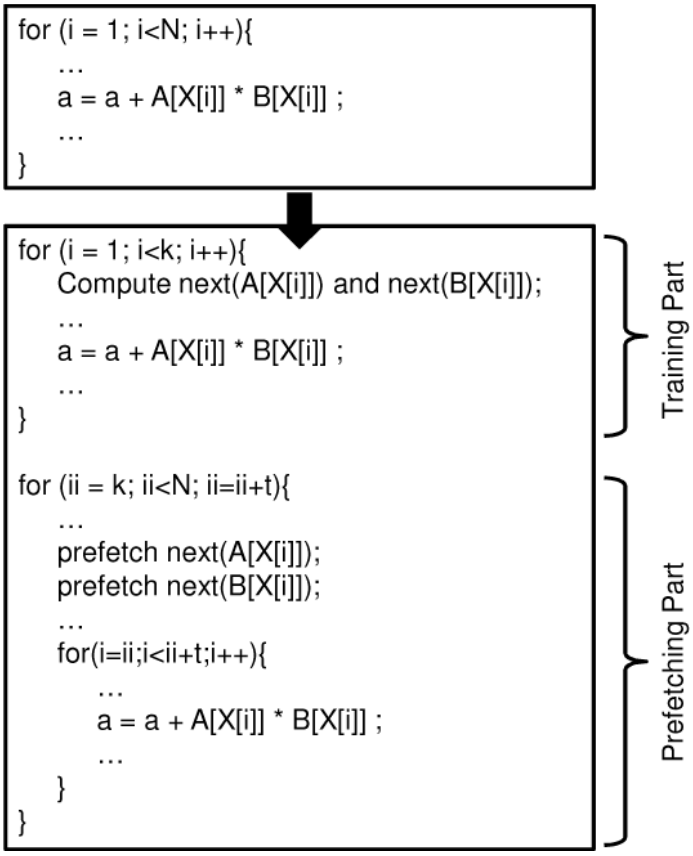


Figure 5.4: Code transformation depicting the training and prefetching parts in our approach.

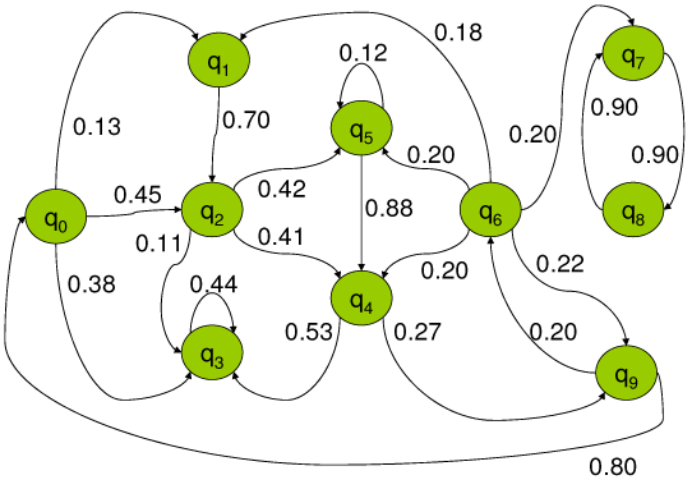


Figure 5.5: Sample Markov Model. Note that this figure shows only high transition probabilities; low ones are omitted for clarity. Each state q_i denotes an access of block B_i, and the weight associated with edge (i, j), between states q_i and q_j corresponds to the probability with which the execution touches block q_j , right after touching data block q_i.

Name	Data Size (MB)	Dominant Access Type
terpa 1.2	3.88	index arrays
aero	5.27	index arrays
bdna	5.9	index arrays
vpr	4.43	pointer based
vortex	2.71	pointer based
oa_filter	2.86	regular
swim	3.76	regular

Table 5.1: Benchmarks and their characteristics.

- A2: This alternative is a variant of the previous one and returns a block only if the corresponding transition probability is the largest among all blocks and above a preset threshold value (δ). In this way, we guarantee that the likelihood of the prefetched block being accessed by execution is high. Again, in the example of Figure 5.5, if the current data block is B_0 and δ is 50%, no block is prefetched under the A2 scheme. As another example, if the threshold value is 50%, next(B_4) is B_3.
- A3: The third alternative prefetches k blocks with the largest transition probabilities. In our example of Figure 5.5, next(B_2) would be $\{B_4, B_5\}$ if k is set to 2.
- A4: The last alternative we experiment with selects k blocks to prefetch such that the cumulative sum of the transition probabilities of these blocks is larger than a preset threshold value (δ). As an example, if δ is set to 80%, next(B_4) would be $\{B_3, B_9\}$ under this alternative. Notice that under this scheme next(.) set can contain any number of blocks.

It is to be noted that some of these alternatives work with parameters, the values of which may be critical to their success. More specifically, schemes A2 and A4 use a threshold parameter (δ), whereas A3 operates with a k parameter. When there are multiple options (combination) that lead to the same threshold value of δ, the A4 alternative selects the combination with minimum number of blocks. The important point to note is that the code shape shown in Figure 5.4 does not change much with the particular scheme (alternative) adopted; the different schemes change only the contents of next(.).

After determining the next(.) blocks in the training part, it is important to efficiently insert the prefetch instructions to the scratch-pad memory for each next block to be used in the successive iterations of the prefetching part. We use an algorithm similar to [85] in order to insert prefetch instructions in the code to prefetch data into the SPM. The prefetch distance (the time difference between time of prefetch and time of first use of a data block) is an important parameter that is determined using the approach in [85], which can be computed as a simple function of the estimated time for a single prefetch and the estimated cycle of each loop iteration. Note that, although this compiler prefetch algorithm is efficient, the choice of the compiler

CPU	2-issue embedded core
SPM Capacity	64KB
Block Size	1KB
SPM latency	2 cycles
Off-chip memory latency	200 cycles

Table 5.2: Simulation parameters.

algorithm for prefetching is orthogonal to the problem of predicting the next(.) blocks. It is also important to note that, the next(.) set of each block could potentially consist of more than one block (depending on whether A1, A2, A3 or A4 is being used), and in such a case, we conservatively insert prefetch for each block in the next(.) set.

A fully adaptive scheme that selects these parameters dynamically can be expensive to implement. Therefore, we fix the values of these parameters at compile time. Obviously, a programmer can experiment with different values of parameters in a given alternative, and select the best performing one for the application at hand.

Another potential issue is what happens when our approach is applied to code with regular data access patterns. While our approach works with such codes as well, the results may not be as good as those that could be obtained using a conventional (static) SPM management scheme. This is due to the overheads incurred by our approach (mainly within the training part) at runtime. In order to quantify this behavior, we also applied our approach to two codes with regular data access patterns, and reported the results in Section 5.4. Note that a compiler implementation can select between our approach and a conventional static scheme, depending on the application code at hand. This is possible because a compiler can infer that a given reference is irregular, though it cannot fully analyze the irregularity it detects.

5.4 Experiments

We implemented our proposed approach using the SUIF compiler [39], and performed simulations with the schemes (A1 through A4) above as well as three SPM management schemes. To perform our simulations, we enhanced SimpleScalar [9]. The important simulation parameters and their default values are listed in Table 5.2. The set of applications used in this study are given in Table 5.1. In the following discussion, Static, Alt-I and Alt-II represent the schemes explained in [54], [20], and [2], respectively. The results of our schemes include both the training and prefetching parts, i.e., all overheads of our schemes are captured. All the performance improvement results presented below are with respect to a version that uses a conventional

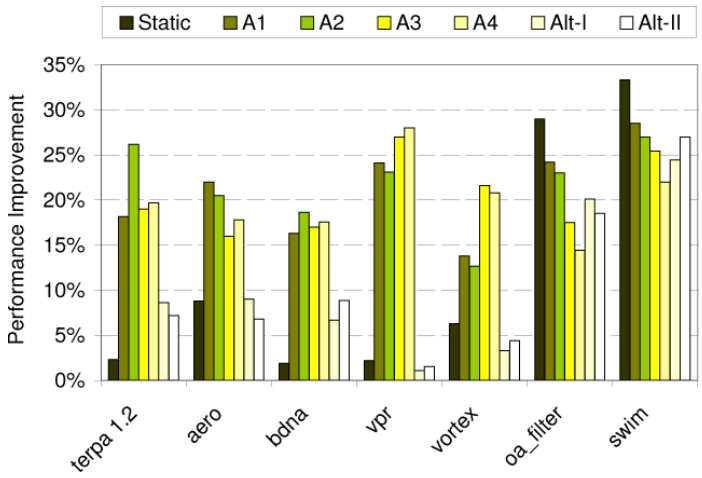


Figure 5.6: Percentage improvement in execution cycles under different schemes.

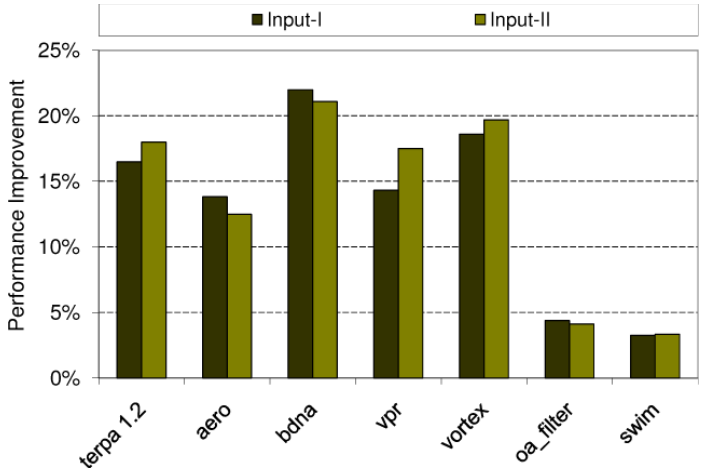


Figure 5.7: Additional performance improvements our approach brings over the approach in [23].

(hardware managed) cache of the same size as the SPM capacity used in other schemes.

Our first set of results are present in Figure 5.6, and give the percentage improvement (reduction) in execution cycles under the different schemes explained above. Our first observation is that the average performance improvements brought by schemes Static, A1, A2, A3, A4, Alt-I, and Alt-II are 11.9%, 21.0%, 21.5%, 20.5%, 20.0%, 10.4% and 10.6%, respectively. We also note that our dynamic schemes (A1 through A4) generate much better savings than the static scheme for all five applications with irregular access patterns. This is expected as the static SPM management scheme in [54] can only optimize a few loop nests in these applications, namely, the nests with compile-time analyzable data access patterns, and the remaining loop nests remain unoptimized. In contrast, our approach, using the explained MC based model, successfully optimizes these applications. We also observe that our dynamic scheme improves performance for our two

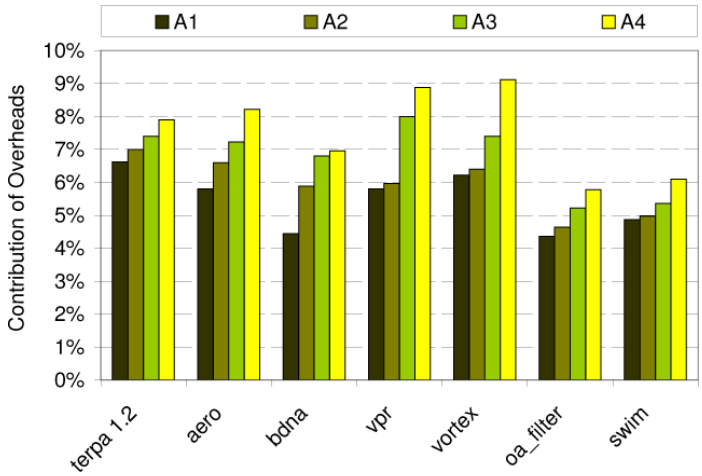


Figure 5.8: Contribution of overheads to the overall execution cycles.

regular applications (oa filter and swim) as well, though the results (savings) are not as good as those brought by the static scheme. This difference is mainly due to the runtime overheads incurred by our scheme as discussed earlier. However, as explained earlier in Section 5.3.2, an optimizing compiler may choose between the static and dynamic schemes depending on the application code at hand.

Among our schemes, we observe that A2 generates better results than the rest in terpa 1.2. This is because the transition diagram for terpa 1.2 is very dense, and as a result, given a node, transition probabilities are almost equally distributed in many cases. This behavior in turn favors A2 over A1, as A2 is more selective in prefetching and does not perform useless prefetches. On the other hand, A3 and A4 issue too many prefetches in this application, and this contributes to the runtime overheads. In applications vpr and vortex, the extra overheads brought by A3 and A4 are compensated by their benefits (the increase in SPM hit rate as a result of more prefetches), and the overall performance is improved.

We now discuss how our approach compares against two previously-proposed schemes that try to address irregular data ac- cesses. Alt-I tracks the statements that make assignments to index arrays and use these values to determine the minimum and maximum bounds of the data arrays. Since this scheme targets irregularity that arises from indexed array accesses, it does not offer a solution for pointer based applications, and consequently, it performs no better than the static scheme for our pointer applications (vpr and vortex). In fact, due to the overheads involved, Alt-I performs worse than the static scheme [54] in these two applications. The same observation goes for Alt-II as well, which also targets exclusively indexed array accesses. When the index array applications (terpa 1.2, aero, and bdna) are considered, our schemes are better than both Alt-I and Alt-II, thanks to the inherent locality exhibited by the indexed array based data accesses.

We also compared our approach (version A1) to the SPM management scheme in [23], which uses profile

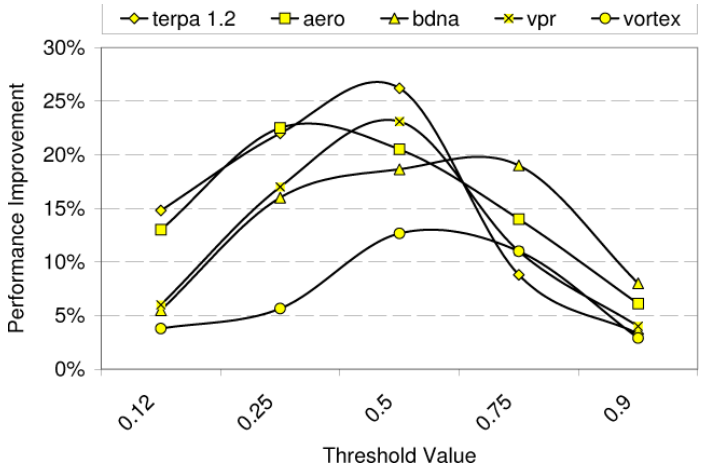


Figure 5.9: Sensitivity of the A2 scheme to the threshold value (δ).

data to place data into the SPM. To do this, we profiled each application using an input set (Input-0) and then executed the same application using two different input sets, Input-I and Input-II, both of which are different from Input-0. The bar-chart in Figure 5.7 gives the additional performance benefits our approach brings over the scheme in [23]. The average improvement when considering all benchmarks is around 13.5%. The reason for this is that in irregular applications the input data used for execution can change the behavior of the application significantly. Therefore, any profile based method will have difficulty in optimizing irregular codes, unless the profile input is the same as the input used to execute the application.

Since our schemes (A1 through A4) incur runtime overheads, it is also important to quantify these overheads. Figure 5.8 gives the contribution of these overheads to the overall execution cycles in our applications. We observe that the overheads range between 4.4% and 9.1%, depending on the particular alternative. As expected, most overheads are incurred by the A4 alternative.

As noted earlier, different versions (A1 through A4) work with different parameters. Now, we quantify the impact of these parameters. Due to space constraints, we focus on A2 and A3 versions only. First, in Figure 5.9, we present the sensitivity of the A2 version to the threshold value (δ), for our irregular applications. It is easy to see from these curves that, for each application, there is an optimum threshold value (among those tested). Working with a smaller threshold value causes unnecessary prefetches to the SPM, while employing a larger threshold value suppresses a lot of prefetches, some of which could have been useful. Similarly, Figure 5.10 plots the sensitivity of the A3 version to parameter k. It can be seen that the different applications reach differently to varying k. For example, terpa 1.2 and vpr take advantage of increasing k values, whereas aero's performance decreases as we increase k.

We now quantify, in Figure 5.11, the influence of the granularity of prefetch on our savings. Each curve

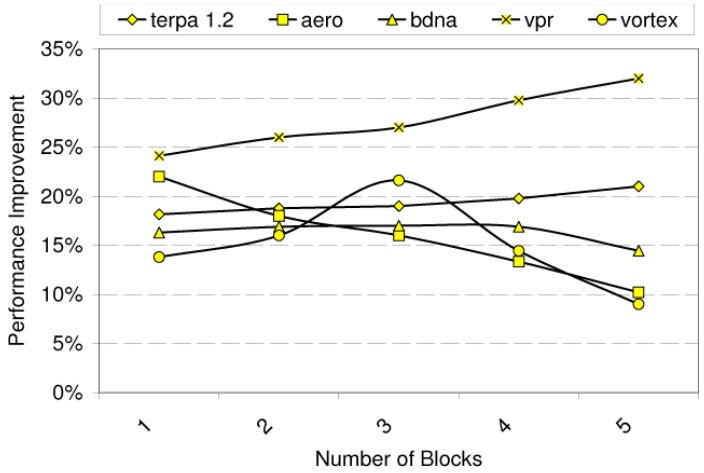


Figure 5.10: Sensitivity of the A3 scheme to parameter k.

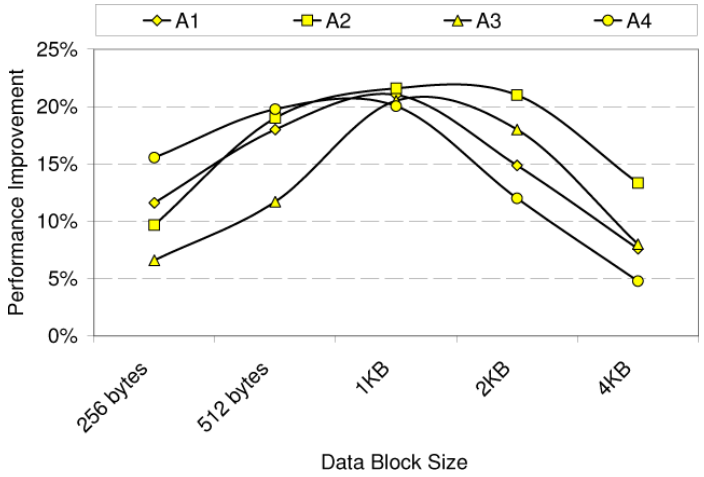


Figure 5.11: Sensitivity to the data block size.

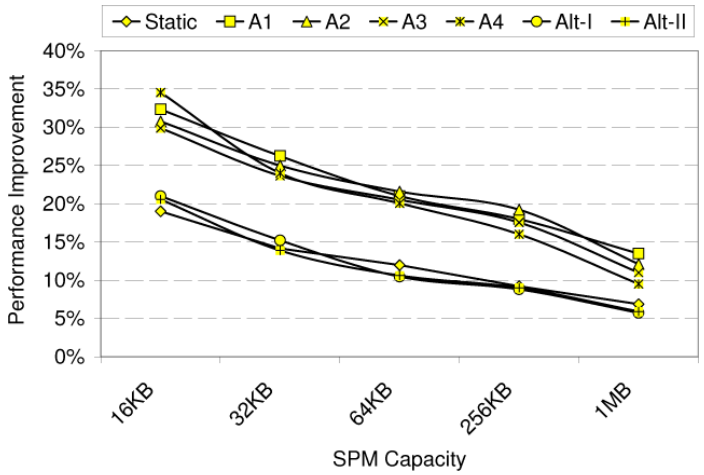


Figure 5.12: Average improvement values across all applications.

in this plot represents the *average improvement* value (across all applications) under varying data block sizes. the default block size used in our experiments so far was 1KB (Table 5.2). We see from these results that block size selection is a critical issue. For example, working with large blocks is not very useful as it causes frequent displacements from the SPM. While this argues for employing smaller blocks, doing so can lead to complex Markov models, which may be costly to maintain at runtime. In addition, small block sizes also increase the activity between the SPM and the off-chip memory, which can in turn affect overall performance. Considering these two factors, one has to make a careful choice for the block size.

It is also important to study the behavior of our scheme under different SPM capacities. The default SPM capacity used in our experiments is 64KB (Table 5.2). The results plotted in Figure 5.12, which represent *average performance improvement* values across all applications, show that our dynamic scheme is consistently better than the remaining schemes for all SPM capacities tested. As can be seen, our performance improvements reduce a bit with increasing SPM capacities. This is expected as the presented results are values normalized with respected to the original case, i.e., the case with conventional hardware-managed cache. As the on-chip memory capacity (SPM or cache) is increased, the difference between our scheme and the original case gets reduced. It should also be noted however that, as the increase in data set size is usually much higher than increase in on-chip memory capacities, we can expect higher savings from our scheme in future systems.

5.5 Conclusion

We proposed various schemes to predict irregular data accesses in data intensive applications using a Markov chain based model. Using such a data access pattern prediction model for prefetching data into scratch-pad memory helps improve the performance of applications with irregular data accesses to a large extent. We observe that scratch-pad memory management using our approaches produces 12.7% to 28.5% improvements in performance across a range of applications with both regular and irregular access patterns, with an average improvement of 20.8%.

In this and the previous chapters, our work targeted CMP systems with a shared cache and a shared SPM. In the next chapter, we will target a CMP architecture with on-chip banked memory.

Chapter 6

Memory Bank Aware Dynamic Loop Scheduling

6.1 Introduction

In this chapter, we present a dynamic loop scheduling scheme. We target CMP architecture with on-chip shared banked-memory that is shared by all the processors.

In a battery-operated embedded execution environment, power consumption is an important metric to consider during dynamic loop scheduling. In particular, in a banked memory system, the loop iteration-to-processor mapping can have substantial impact on memory system power consumption (since it determines the set of banks that will be exercised at any given period of time), which can be a significant portion of the overall energy consumption for the data-intensive embedded applications.

In this chapter, we present a *bank aware dynamic loop scheduling scheme* for array-intensive embedded media applications. The goal behind this new loop scheduling scheme is to minimize the number of memory banks that need to be used for executing the current working set (group of loop iterations) when all processors are considered together. The unused memory banks can then be held in a low power state, for longer periods of time, to conserve energy. The proposed scheduling approach represents both the current active/idle status of banks and the set of banks that may be accessed by a given loop iteration using bitmaps and uses these bitmaps at runtime in performing the iteration assignment. The goal is to minimize the number of banks that are active at any given period of time.

We implemented this memory bank-aware loop scheduling scheme and performed experiments with

several embedded application codes. In our experimental evaluation, we compare it, in terms of both energy consumption and performance, to a number of previously proposed loop scheduling strategies, including both static and dynamic techniques. Our experimental results show that the proposed scheduling scheme not only reduces the energy consumption significantly, but it also leads to much better energy savings when compared to these prior techniques and it is competitive with the loop scheduler that generates the best performance. To our knowledge, this is the first dynamic loop scheduling scheme that is memory bank aware.

The remainder of this chapter is structured as follows. Section 6.2 gives background on loop scheduling and banked memories. It also discusses the relevant prior work on loop scheduling. Section 6.3 presents the details of our proposed bank-aware loop scheduling scheme. This scheme is evaluated along with several previously proposed scheduling techniques in Section 6.4, and the results are discussed from both energy and performance angles. Section 6.5 concludes the chapter with a summary.

6.2 Related Work

As mentioned earlier, the prior work on loop scheduling considered both static and dynamic techniques. The static techniques perform iteration assignment to CPUs at compile time, and therefore, are easy to implement, as compared to their dynamic counterparts, which require some work to be performed at runtime, thereby contributing (as overhead) to the execution time. The basic static technique [101, 118] divides the iteration space of the parallel loop (i.e., the set of all iterations in the loop) to be scheduled into P equal (or almost equal) subsets where P is the number of processors at hand, and each processor is assigned a subset. As mentioned in [63], static scheduling can also assign iterations in a locality aware fashion by ensuring that the set of iterations assigned to a processor exhibit data reuse among themselves. However, such a purely static assignment of iterations can lead to load imbalances, which can be due to the different reasons:

- Variations due to conditional control flow. For example, the different branches of an IF or SWITCH statement can be taken by the different loop iterations, and consequently, there can be large variations across the execution times of the iterations that belong to the same parallel loop.
- Variances due to loop index dependent bounds. When the lower bound and/or upper bound of an inner (sequential) loop depends on the index of the parallel outer loop, the different iterations of this outer loop can experience different execution latencies.
- Variances due to data locality (cache behavior). The different loop iterations can produce different cache hit/miss counts, which can in turn lead to significant variations among their execution times.

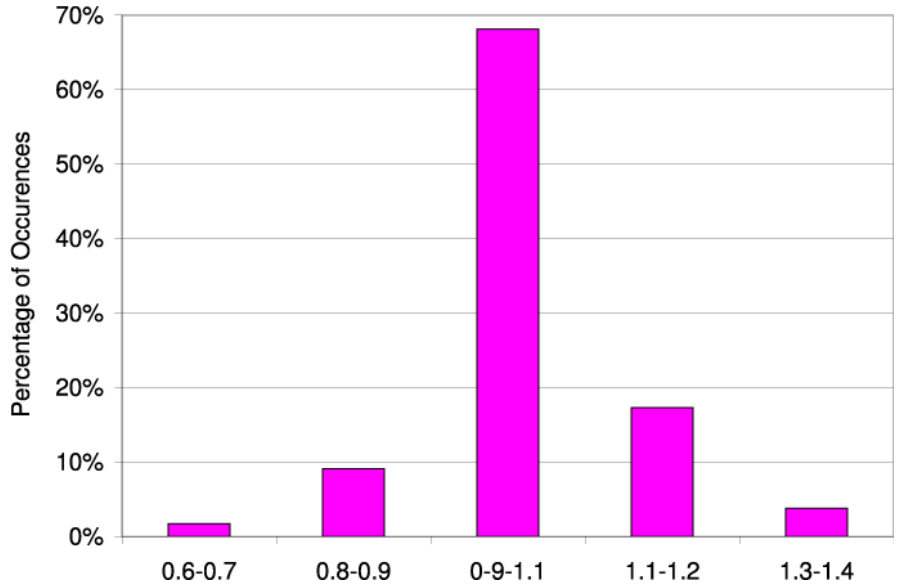


Figure 6.1: Distribution of the execution latencies of iterations for a typical parallel loop from one of our applications.

It needs to be noted that most of these variations are not possible to capture at compile time, and thus, a purely static scheduling technique can be very inefficient when these variations are really significant. As a result, due to these factors, partitioning loop iteration space across available processors evenly such that each processor receives more or less the same number of iterations can lead to large variances across the execution times of the CPUs. In addition, since the execution time of a parallel loop nest is typically determined by the execution time of the processor that finishes its group of iterations last, an unbalanced partitioning (in terms of execution cycles) can be detrimental to performance. To give an idea about the magnitude of this variance, Figure 6.1 shows the distribution of the execution latencies of iterations for one of the loops in baleen, one of our embedded applications (we will present the details of our embedded applications and the simulation platform used later). The y-axis in this figure captures the percentage of occurrences for each latency group. The value 1 on the x-axis represents the most frequently occurring latency across all iterations, and all other latencies are normalized with respect to that value. So, each bar gives the percentage of occurrences for a latency interval. It is easy to see that the largest variance across the executing latencies of the different loop iterations is around 40%, indicating the severity of the problem. We need to mention that this particular loop nest was not an extreme case; rather, it was exhibiting a typical behavior. In fact, we observed during our experimental evaluation that the largest variation across two different iterations of the same parallel loop varied between 1% and 82%, averaging on 37%.

In dynamic loop scheduling, a master CPU assigns works to slave (worker) processors at runtime. One of the earliest dynamic scheduling schemes is known as self scheduling [101], and addresses load imbalance by assigning a small load to slave processors initially, and allowing the slaves to ask for more work once they are done with their current assignments. Each work assignment takes place within a critical section, a piece of code for which entrance is granted to one CPU at a time. A variant of self scheduling is tapering, also known as guided self scheduling [100], where the loads assigned to processors are reduced gradually as the execution progresses, in an attempt to prevent potential imbalances towards the end of the iteration space. The other variants of dynamic scheduling include factoring [46] and trapezoid self scheduling [113]. Our initial experiments with these different performance overhead of dynamic schedulers found that the performance and energy behaviors of tapering, factoring, and trapezoid self scheduling were very similar to each other (within 1% of one another). Therefore, as far as dynamic schedulers are concerned, we present results with self scheduling and tapering only.

More recently, there have been several proposals for locality aware loop scheduling schemes. The distinguishing characteristic of these schemes is that they are specifically designed for optimizing the behavior of data cache by assigning iterations to processors carefully. Such techniques either take advantage of the data reuse across different iterations (as in [55]) or exploit the fact that the same loop can be visited multiple times during the execution of the program [80]. Since optimizing cache locality means exploiting temporal and/or spatial reuse in the innermost loop iterations, one can expect a locality aware loop scheduling scheme to reduce bank energy consumption as well. This is the main reason why we compare, during the experimental evaluation, our approach to the locality aware loop scheduling schemes as well, in addition to pure static and dynamic scheduling techniques. We discuss a static scheduling scheme, specifically designed for software managed on-chip memories. There also exist several efforts that consider dynamic scheduling in the operating system (OS) [18] or system levels [125]. Our approach is different from such approaches in that in our case the compiler dictates the scheduling decisions, and we target embedded chip multiprocessors.

As far as the banked memory architecture is concerned, we are focusing on an SRAM-based on-chip memory system, divided into multiple banks. In this system, each bank can be transitioned into a low leakage mode independently of the others to save energy. The specific policy we implemented in our simulations is based on [33], where the banks are placed into the low leakage mode periodically (using the suggested threshold values by [33]), and are activated when they are accessed (we assume the same reactivation penalties as in [33]). Consequently, working with a small set of banks at a given period of time increases the chances for the other (unused) banks to remain in the low leakage mode, thereby increasing energy savings. This is why our dynamic approach tries to schedule the loop iterations in such a fashion that the idle periods of the banks are lengthened as much as possible. We are not proposing a new bank power reduction strategy,

but readers interested on power saving in banked SRAM/DRAM memories can read the papers [57, 64] and the references therein.

6.3 Our Approach

As discussed in the previous section, in dynamic scheduling, when a worker CPU finishes its current work (assignment), it asks the master CPU to give it a new set of loop iterations. The important point to note is that the master has complete freedom in selecting this set of iterations, due to the fact that we consider parallelization of dependence-free loops only. In order to reduce the number of active banks at any given moment, our bank aware scheduler selects this set of iterations such that it *reuses* only the active banks, if it is possible to do so. Clearly, this complete bank reuse may not always be achievable, and when this is the case, our approach selects the iterations to be assigned to the requesting CPU such that the number of additional banks required is minimum. In mathematical terms, we represent the *current on/off status*[*] of the banked memory system using a bitmap ∇_c of the following form:

$$\nabla_c = m_1 \bullet m_2 \bullet m_3 \bullet \cdots \bullet m_s,$$

where s is the number of banks in the memory system and m_i captures the status of bank i. Specifically, if m_i is 1, bank i is currently active, i.e., there is at least one processor in the system that executes an iteration which accesses that bank. On the other hand, m_i = 0 indicates that the bank is currently unused (Such a bank can be in high leakage or low leakage mode, depending on how long it was unused.) At a particular step, when the scheduler is about to assign a workload to a processor, it selects the set of iterations (that constitute the workload to be assigned) such that only the currently active banks are used if it is possible. Mathematically, let $\zeta(\vec{I})$ be a function that gives the set of banks that may be accessed by iteration $\vec{I}$ (Note that $\vec{I}$ is a vector where each entry corresponds to a value of a loop, starting from the outermost loop, corresponding to the first entry). Note that this is a *conservative* estimate since it may not always be possible to determine the exact set of banks to be accessed by a particular loop iteration. As a consequence, in the worst case, $\zeta(\vec{I})$ can contain all the banks in the memory system. Observe that a $\zeta(\vec{I})$ can also be represented as a bitmap $\beta(\vec{I})$ as follows:

$$\beta(\vec{I}) = n_1 \bullet n_2 \bullet n_3 \bullet \cdots \bullet n_s,$$

[*]When there is no confusion, we use the terms such as low power mode, low leakage mode, and off mode interchangeably. It needs to be made clear however that, in our implementation, when a bank is placed into the low power mode, its contents are maintained, using an approach, similar to that discussed in [33].

where n_i is set to 1 if bank i belongs to the $\zeta(\vec{I})$ set; otherwise, it is set to 0. Now, we can also define a workload level bitmap that captures the bitmaps of all the iterations in a given workload collectively. That is,

$$\tau(W) = k_1 \bullet k_2 \bullet k_3 \bullet \cdots \bullet k_s,$$

such that k_i is set to 1 if there is at least a $\vec{I} \in W$ such that n_i of $\beta(\vec{I})$ is 1; otherwise, k_i is set to 0. In other words, let $\vec{I}_1, \vec{I}_2, \ldots, \vec{I}_l$ be the set of iterations in the workload assigned to a processor when it asks for more work, and $\beta(\vec{I}_j) = n_{j,1} \bullet n_{j,2} \bullet n_{j,3} \bullet \cdots \bullet n_{j,s}$, where $1 \leq l$, Then, we have

$$k_i = n_{1,i} \vee n_{2,i} \vee n_{3,i} \vee \cdots \vee n_{l,i},$$

where $\vee$ denotes the OR operator.

The main job of our bank aware dynamic scheduler is to build a workload W at runtime and assign it to an idle processor which asks for more work to do. Let us first make the following definition. Given bitmaps p and q, we write $p \triangleright q$, if the following two conditions are satisfied together:

- The number of 1s in q is equal to or larger than that in p, and
- If p has 1 in i^{th} position, q also has a 1 in its i^{th} position.

For example, we have $11001000 \triangleright 11101001$ and $0101 \triangleright 0101$, while $1010 \triangleright 0101$ and $111000 \triangleright 101000$ are not correct. Based on this definition, the iterations to put in set W should be selected in such a fashion that $\tau(W) \triangleright \nabla_c$ should be satisfied if it is possible to do so. Before giving the pseudo code for the algorithm that selects W, the set of iterations to be assigned to a worker CPU that requires work, we want to discuss a couple of important issues.

First, it can be costly to compute a W set at runtime such that $\tau(W) \triangleright \nabla_c$ even if such a set that satisfies this condition does actually exist. Therefore, our approach does some extra work at compile time to reduce this potential runtime overhead. Let us use Z to denote the set of iterations to be executed for the current parallel loop (i.e., Z represents the iteration space of this loop). We divide at compile time this set into 2^s bins, where s is the number of banks in the memory system. Each bin holds the set of iterations that have the same $\beta(\vec{I})$ bitmap (in our implementation, each bin is represented by the constraints that give the iterations which belong to that bin). Clearly, some of these bins can be empty (i.e., there may not exist any iteration that accesses a particular set of banks). As the iterations are assigned to processors, we update the contents of these bins accordingly, a process during which some (originally full) bins can become empty. It is important to note that each bin can be represented using a bitmap, similar to those used for representing the on/off status of the banks (∇_c) and the bank access patterns of iterations ($\beta(\vec{I})$). Let $\mu(q)$ represent the bitmap of

For $i = 1, 2^s$
  Generate the contents of bin q_i
  Compute, $C(q_i)$, the size of q_i
  Select, if possible, two bins q_j and q_k such that $\mu(q_j) \vee \mu(q_k) = \mu(q_i)$
    and connect q_j and q_k to q_i
EndFor

Figure 6.2: Compile time component of our approach.

Obtain ∇_c
Determine the bin q_i to check
If $C(q_i) \geq K$ then
  Return the first K iterations in q_i to the requesting processor
  Update the contents of bin q_i
Else
  Take M iterations from q_i, where $M < K$
  Update the contents of bin q_i
  Determine bins q_j and q_k that are connected to bin q_i
  If $C(q_j) + C(q_k) \geq K - M$ then
    Return $K - M$ iterations from these bins and update their contents
  Else
    Return the remaining iterations from any subset of bins and update contents

Figure 6.3: Runtime component of our approach.

bin q. Suppose now that we are to assign a workload W (which contains K iterations to be selected from the iterations contained in Z) to a processor which asks for work. Using ∇_c, we first check the whether the bin q where $\mu(q) = \nabla_c$ has at least K iterations. If this is the case, we give K iterations to the requesting processor and we are done. If this is not the case (i.e., when bin q can provide only $M(< K)$ iterations, we next search to find a set of bins $q_1, q_2, \ldots, q_r$ such that $\mu(q_1) \vee \mu(q_2) \vee \cdots \vee \mu(q_r) \triangleright \nabla_c$ and can collectively provide $K - M$ iterations for the requesting processor, where $M(< K)$ is the number of iterations provided by $\mu(q)$. Informally, this means selecting a set of bins such that collectively they provide $K - M$ iterations and these $K - M$ iterations do not demand access to any of the unused banks. Since there are many ways of selecting these banks and we cannot try all the alternatives at runtime due to cost considerations, we mark at compile time which alternative to try if we cannot find a bin q where $\mu(q) = \nabla_c$ has at least K iterations. In our current implementation, we try only one alternative which consists of two banks, and if that alternative cannot provide the required number of iterations, we select the remaining iterations randomly. It is also important to explain why we first try to find a bin q such that $\mu(q) = \nabla_c$ has at least K iterations. This is due to the following observation. At the time the slave processor in question asks for a workload, the set of banks that are active is captured by ∇_c. Since these banks are active anyway, we may want to reuse all of them (while they are active). This is because such a reuse will also likely to help cluster the iterations that do not use some subsets of these banks, which will in turn help increase power savings in the rest of the execution of

	$\mu(q_i)$	Scenario I	Scenario II
q_1	1 0 0 0	8	7
q_2	0 1 0 0	8	27
q_3	1 0 1 0	20	2
q_4	0 0 1 0	5	6
q_5	0 1 1 0	5	4

Figure 6.4: Example application of our loop scheduling algorithm. The last two columns give the number of iterations in five bins; the other bins are assumed to be empty.

this parallel loop nest. Figure 6.2 shows the compile-time portion of our approach based on the explanation above.

Second, our bank aware loop scheduling approach can be used with any variant of self scheduling. This is because the different variants of self scheduling such as tapering [100] and factoring [46] differ only in the number of iterations they assign at runtime to a requesting CPU. Based on our discussion of the previous paragraph, this only affects the value of K (size of the workload W) and the rest of our approach can be used as it is. In our experimental evaluation however, we only implemented the bank aware version of the baseline self scheduling.

Third, it is important to understand why a purely static (bank aware) scheduling approach may not be as successful as our dynamic bank-aware scheduler. Notice that any purely static loop scheduling scheme that is to be bank aware needs to make conservation assumptions about the on/off status of the memory banks during each phase of the execution of the parallel loop. In other words, it needs to estimate the bitmap ∇_c conservatively (at compile time). However, such a conservative estimation can be far from reality due to at least two factors. First, dynamic cache behavior can affect the bank access pattern of a loop iteration completely. For example, for a given loop iteration, the compiler can conservatively deduce that it can access four banks; however, at runtime, all these four accesses can be captured and supplied by the data cache, resulting in no memory access. The second potential reason that can invalidate the compiler based estimation is the dynamic code behavior. For example, a procedure/function can have a lot of conditional branches. In order to make a conservative estimate, the compiler needs to consider the worst case scenario. However, in a given execution, only a subset of all possible branches can be taken, which means a much smaller number of bank accesses, compared to the conservative estimation made at compile time. Nevertheless, in our experiments, we also measured how much energy we would lose, had we adopted a static bank aware scheduling approach, instead of the dynamic approach presented in this chapter. The pseudo-code for the algorithm that selects the workload W to be given to a requesting CPU is shown in Figure 6.3. This algorithm is executed at run-time and constitutes the dynamic portion of our approach (the static portion of our approach

is given earlier in Figure 6.2).

We now give an example to illustrate how our approach works in practice using two scenarios presented in Figure 6.4. In this example, we assume that the memory system has four banks and at compile time we grouped the iterations into five different bins $(q_1, q_2, \ldots, q_5)$. Let us assume that the current bank on/off status, ∇_c, is 1010, the same as the bitmap of bin q_3. We further assume that the workload W we are to assign has 10 iterations. Note that each scenario in Figure 6.4 corresponds to a particular number of iterations at each bin. Under Scenario I, we first check q_3 to see whether it can supply the required number of iterations. Since this bin has currently 20 iterations and we need only 10 (i.e., K=10), we take the first 10 iterations from this bin, and reduce its contents to 10, and we are done. Under Scenario II, we also first check q_3. But, this time, this bin can supply only 2 iterations (i.e., M=2). So, we need $K - M$ = 10-2 = 8 more iterations. Assuming that q_1 and q_4 are identified as the backup bins for q_3 (since $\mu(q_1) \vee \mu(q_4) = \mu(q3)$), we next check q_1 and q_4. Since q_1 can give us 7 iterations, we need only 1 (=10-(2+7)) iteration from q_4. After that, the contents of q_1 and q_4 are updated accordingly.

6.4 Experiments

Using SIMICS [108], we simulated a chip multiprocessor architecture and evaluated the following loop scheduling schemes:

- **static:** This represents the well-known compiler based loop scheduling scheme. In this approach, the iterations of the loop to be executed in parallel are divided across the available parallel processors as evenly as possible. As noted earlier in the text, the main problem with this approach is that it cannot take into account the dynamic variances across the workloads of the different processors. In other words, distributing loop iterations evenly does not necessarily lead to evenly distributed workloads.
- **dynamic:** This is a well-known dynamic loop scheduling scheme (also known as self scheduling [118]). A master processor controls the loop distribution at runtime.
- **tapering:** This is a slight variant of the dynamic scheme, and we followed the specific implementation discussed in [100]. Our initial experiments that compared this scheduling scheme with trapezoid self-scheduling [113] and factoring [46] showed that all these three schemes exhibit similar behavior for our benchmark codes; therefore, we do not report separate results with the trapezoid self-scheduling or factoring.
- **locality aware-dynamic:** This is a dynamic locality aware scheduling scheme, explained in [55]. In this scheme, whenever a processor asks for a workload, it is given a set of loop iterations that exhibit

Table 6.1: Default values of our simulation parameters.

Parameter	Value
Number of Processors	8
IPC	2
L1 Cache (Per Processor)	8KB; 2-way; 32 byte line size
Shared On-Chip Memory	4MB; 8 banks of 512KB
L1 Access Latency	2 Cycles
On-Chip Memory Access Latency	8 Cycles
Bus Contention Cost	5 Cycles

Table 6.2: Benchmark codes used in our experiments. The numbers under the last two columns are for the static loop scheduling scheme. The energy numbers are calculated for 70 nm.

Benchmark Name	Explanation	Dataset Size	# of Cycles	Energy Consump.
Baleen	Segments an image into subimages	2.85MB	344.52M	171.48mJ
Demosaic	Interpolates a complete image from partial raw data	3.51MB	576.12M	247.18mJ
Imar ver2	Transforms different sets of data into one coordinate system	2.51MB	305.83M	148.17mJ
Zonography	A variant of linear tomography kernel	3.94MB	883.9M	481.92mJ
Poly 1.1	A complex form of tomography (poly tomography)	3.98MB	927.06M	515.64mJ
Cbd	Car barrier detection algorithm	1.73MB	290.46M	129.13mJ

high degree of data reuse among them. The goal of such an assignment is to improve the data cache locality. The reason that we make experiments with such a scheme as well is to demonstrate that a dynamic scheduling scheme that targets only cache locality may not be sufficient for maximizing bank energy savings.

- **locality aware-static:** This scheduling scheme is similar to the previous one, except that the assignment of loop iterations to processors are done statically (at compile time). Simply put, the iterations space of the parallel loop is divided at compile time into P subsets (P being the number of processors) such that the iterations in each subset reuse a lot of data elements among themselves. It generates similar results to the locality-aware static scheduling described in [80].

- **bank aware:** This is the scheduling scheme discussed in Section 6.3.

The code modifications required by these schemes are automated using the SUIF infrastructure [39]. As mentioned earlier, we used the SIMICS [108] platform to perform our experimental evaluation. SIMICS is

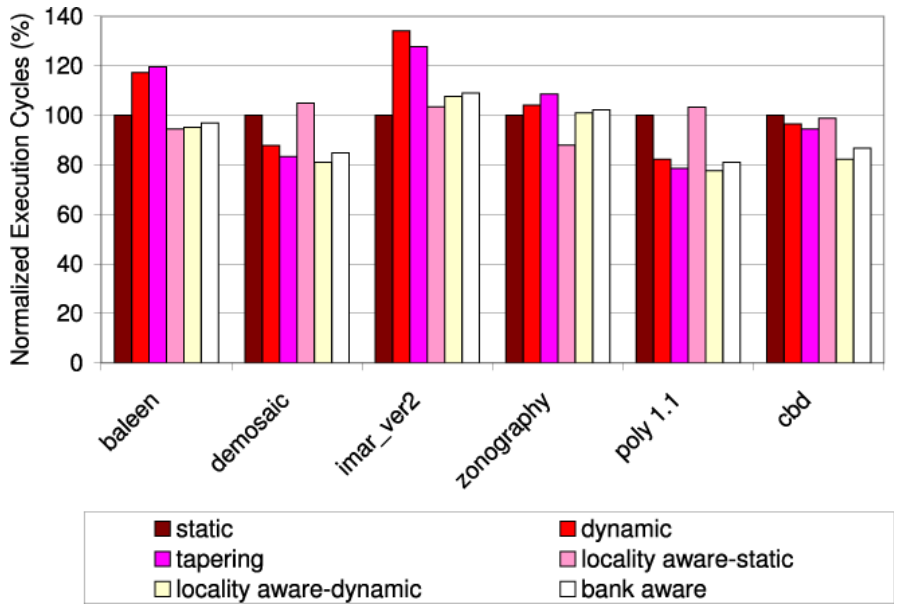


Figure 6.5: Execution cycles normalized with respect to the static scheduling scheme.

a functional simulator and runs unmodified operating systems, drivers, firmware, and application software on the simulated machines. As far as the software is concerned, there is no difference from a real machine. We used this platform to simulate an embedded chip multiprocessor system with private (on-chip) L1 and shared (on-chip) SRAM memory, which is banked. The default number of banks is 8 and each bank is 512KB (meaning that the total on-chip memory space is 4MB). The architecture has separate L1 instruction and data caches for each and every processor. The default simulation parameters used in most of our experiments are given in Table 6.1.

Table 6.2 present the important characteristics of the benchmarks used for evaluating our bank aware dynamic loop scheduling scheme. The third column of this table gives amount of data manipulated by each benchmark, and the fourth column shows the execution cycles taken by a pure static scheduling scheme (as explained above). The last column of the table gives the energy consumption in the memory system, again under the static loop scheduling scheme. The performance and energy numbers presented in the rest of this section are given as values, *normalized* with respect to the last two columns of this table.

Figure 6.5 presents the execution cycle results. One can easily see that the dynamic scheduling scheme (second bar) generates savings (over the static scheme) in three applications, namely demosaic, poly 1.1, and cbd. These are exactly the benchmarks with large workload variations across processors when the static workload assignment is employed. In the other three benchmarks, however, the dynamic scheme generates poor results. Overall, as compared to the static scheme, the dynamic scheme ends up with 3.7% performance degradation when averaged over all six benchmark codes. The behavior exhibited by the tapering scheme

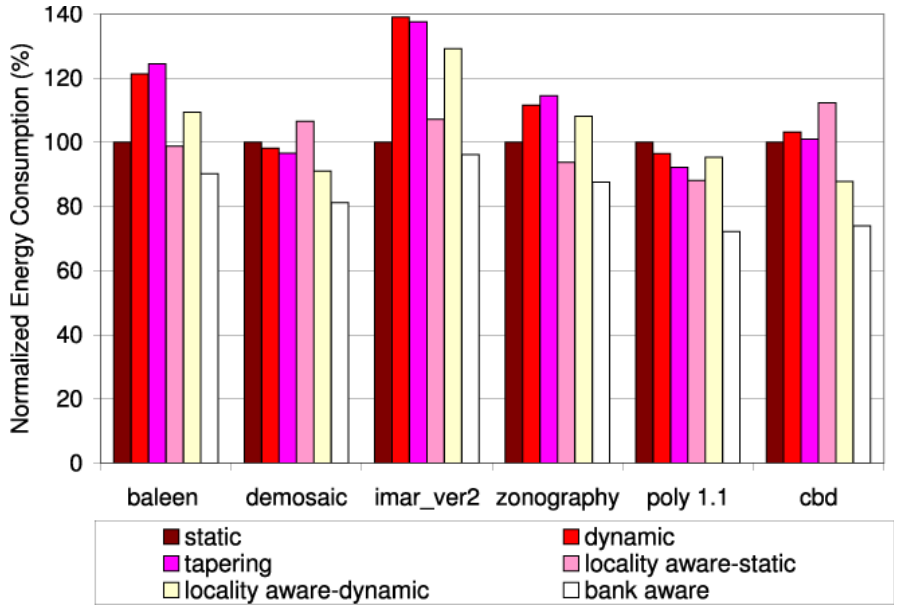


Figure 6.6: Energy consumptions normalized with respect to the static scheduling scheme.

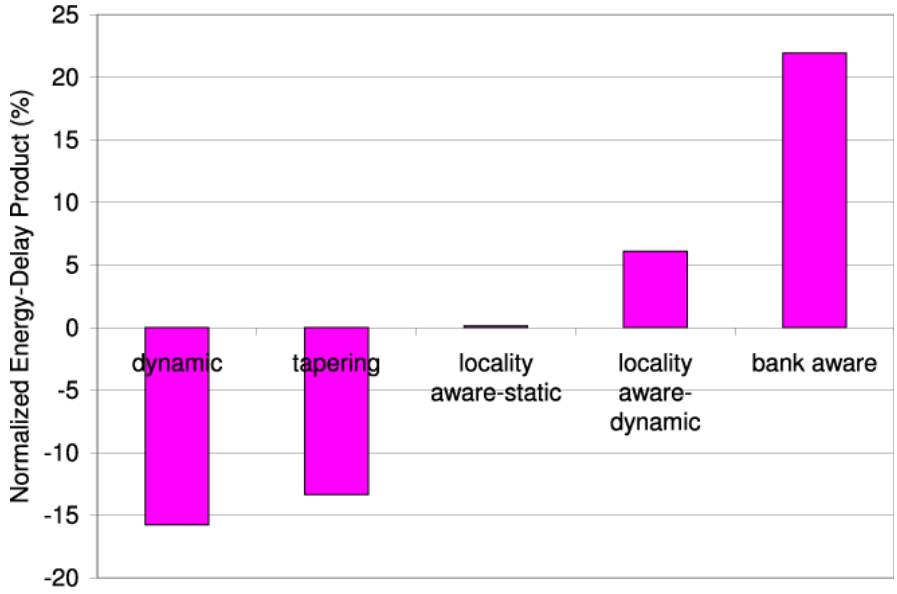


Figure 6.7: Average energy-delay products normalized with respect to the static scheduling scheme.

(third bar) is similar to that of the dynamic scheme, with an average performance degradation of about 2% over the static scheme. In comparison, locality aware-static scheme (fourth bar) performs better, bringing reasonable improvements in two benchmarks (baleen and zonography). The locality aware-dynamic scheme (fifth bar) is much more successful since it is able to both take advantage of data reuse and exploit load imbalances. The locality aware static and dynamic schemes bring average performance improvements of 1.2% and 9.2%, respectively, over the static loop scheduling scheme. Lastly, our bank-aware loop scheduling scheme (last bar) achieves 6.5% improvement over the static scheme. Although it is not as good as the locality aware-dynamic scheme (as the latter is pure performance oriented), it is not too far from it either. This is because of the fact that, most of the time, minimizing the number of accesses to a small set of banks (which is the main goal of our approach) also leads to good data cache behavior, as it tends to improve data reuse within a given time period.

The normalized energy consumption results are presented in Figure 6.6. The energy consumptions for on-chip memory components (L1 cache and on-chip memory) are calculated with the help of the CACTI toolset [107]. The energy consumption of the remaining components on the other hand are obtained using activity based energy models similar to those used in Wattch [15]. Maybe the most important observation one can make from the results in Figure 6.6 (which include energy consumption of both memory and non-memory components) is that the bank aware scheme performs much better than the remaining schemes, bringing an average energy saving of 16.4% over the static loop scheduling scheme. In fact, it reduces energy in all six benchmark codes. In comparison, the remaining scheduling schemes do not repeat the savings they achieve in execution cycles. For example, the dynamic, tapering, locality aware-static, and locality aware-dynamic scheduling schemes increase the energy consumption of the static scheduling scheme by 11.6%, 11.1%, 1,1%, and 3.4%, respectively, on average.

Since any loop scheduling scheme affects both execution cycle count and energy consumption, it is also important to quantify the energy-delay product values. Figure 6.7 gives the energy-delay products when *averaged* over all six application codes. Each bar in this figure is normalized with respect to the average energy-delay product of the static loop scheduling scheme. Since the bank aware scheduling approach improves both performance and energy consumption, it exhibits the best energy-delay product. We also see that, as well as energy-delay product is concerned, the locality aware-dynamic scheme is the only scheme (other than our bank aware approach) that brings some reasonable improvement.

Figure 6.8 presents the *average* performance and energy values with the different processor counts (4, 8, 12, and 16). Remember that our default processor count was 8. We see that, as we increase the number of processors, the differences among the different scheduling schemes get magnified (this is true for both performance and energy). The main reason for this is the fact that an increase in the processor count usually

leads to spread the bank accesses more in the memory space (i.e., more irregularity in the bank accesses). Considering the possibility that future chip multiprocessors will accommodate large number of CPUs, we believe that these results are encouraging. Figure 6.9 gives the energy results with different number of banks, keeping the total memory space at 4MB. All other parameters are set to their default values shown in Table 6.1. We see that the behavior of the locality aware scheduling schemes improve with smaller number of banks. This is because as the number of banks gets smaller, optimizing for cache locality generates similar results to those obtained by optimizing bank locality. However, when the number of banks is increased, these two locality concepts start to behave differently, and as a consequence, our bank aware scheduling scheme generates much better results than the others.

We next study an alternate bank aware scheduling scheme, which is a *static version* of the dynamic scheme discussed here. The only difference between this scheme and our dynamic scheme is how the ∇_c bitmap is obtained. In our dynamic scheme, it is obtained at run-time, while in this alternate bank aware scheme, it is computed at compile time. As discussed earlier in Section 6.3, the compile time computation of ∇_c can be overly pessimistic. The energy results captured by the first two bars (for each benchmark) in Figure 6.10 corroborate this expectation. We see that the average energy improvements by the dynamic and static scheduling schemes are 16.4% and 10.4%, respectively (the results for the dynamic scheduling scheme are reproduced from Figure 6.6). These results underline the importance of dynamically obtaining the ∇_c bitmap. To better understand the difference between the statically computed and dynamically obtained bitmaps, we also recorded during the experiments, the causes for misprediction (of the ∇_c bitmap) with the static scheme. The results are presented in Figure 6.11. As discussed earlier, cache behavior and dynamic control flow are two important reasons for the conservative estimation of the bitmap, also corroborated by the results in this graph. The third portion of each bar in this graph represents the mispredictions whose cause we could not identify.

Recall that in our current implementation we make two attempts to select the set of iterations that satisfy the requirement that no new banks are activated by the newly-assigned workload. In the first attempt, we try to find a bin q where $\mu(q) = \nabla_c$ has at least K iterations. If this fails, in the second attempt, we try to find a set of bins q_1 and q_2 such that $\mu(q1) \vee \mu(q2) \triangleright \nabla_c$ and can collectively provide the $K - M$ iterations, where M is the number of iterations provided by bin q. If this try also fails, then we select the remaining iterations required randomly. However, it is clear that, further improvements to this implementation are possible by increasing the number of attempts. In the general case, when the first attempt fails, we can search for a set of bins $q_1, q_2, \ldots, q_r$ such that $\mu(q_1) \vee \mu(q_2) \vee \cdots \vee \mu(q_r) \triangleright \nabla_c$ and these bins can collectively provide the required number of iterations. Note that in general there are many ways of selecting these bins. The last two bars for each benchmark in Figure 6.10 represent the energy consumption values with two enhanced

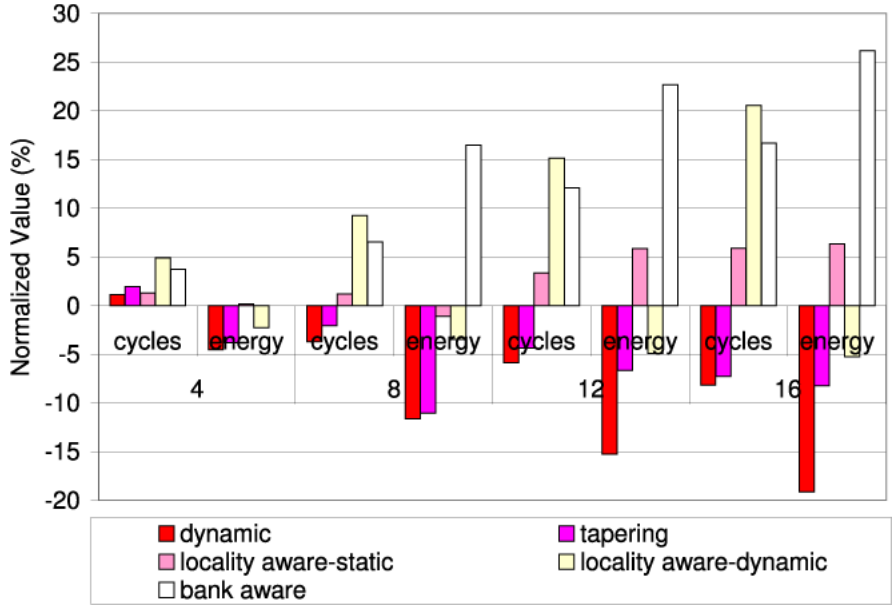


Figure 6.8: Average energy consumption and execution cycle results with different processor counts.

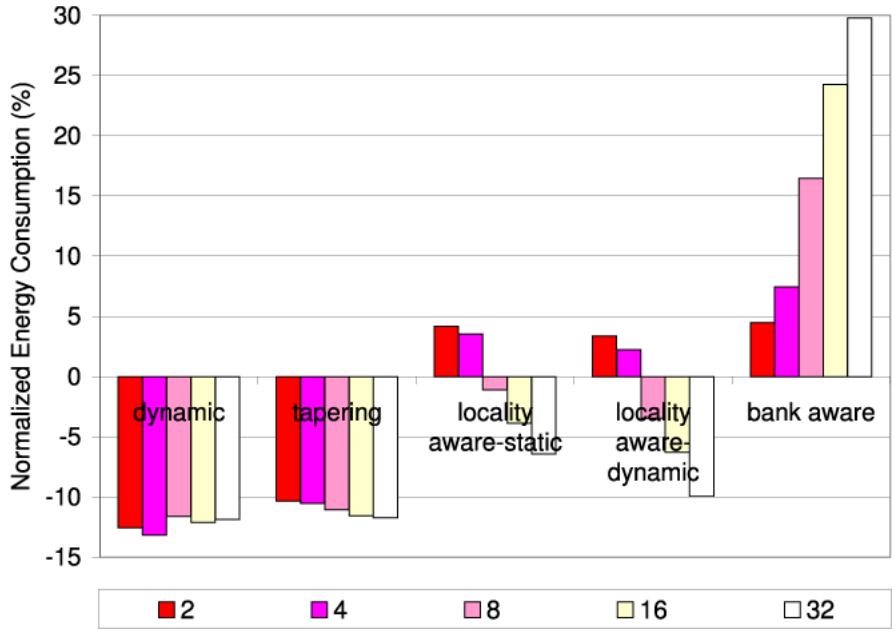


Figure 6.9: Average energy and execution cycle results with different bank counts. In each experiment, the total memory capacity is the same.

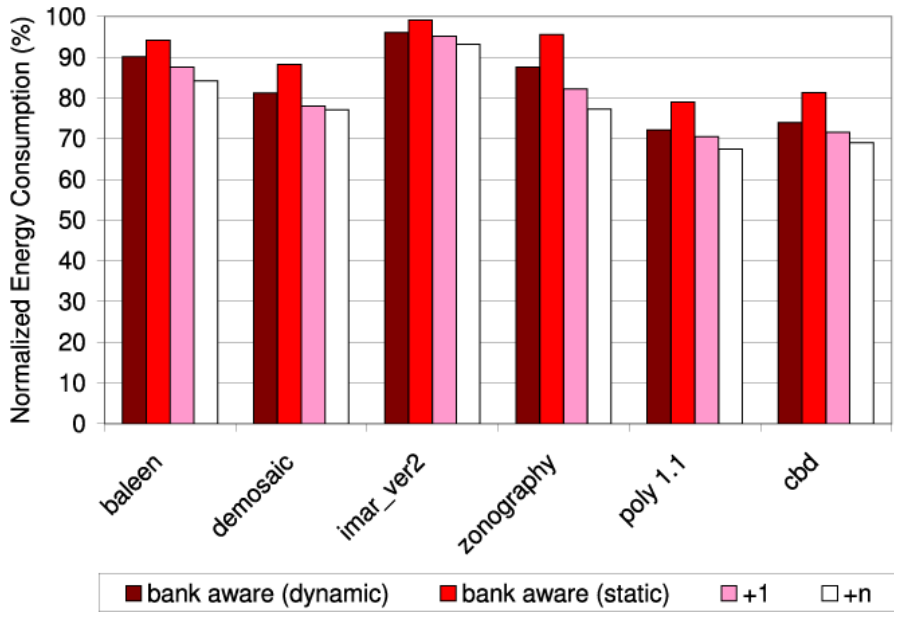


Figure 6.10: Energy comparison of different schemes.

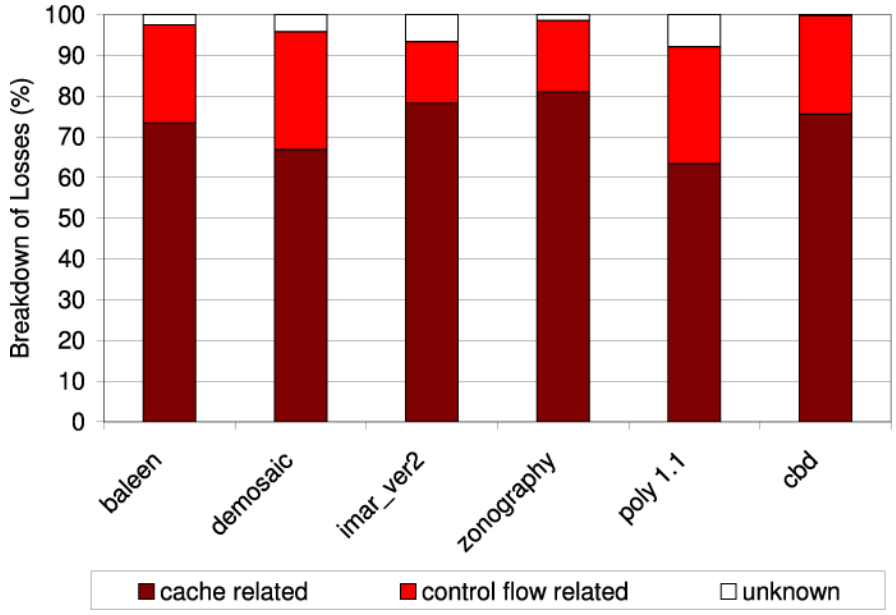


Figure 6.11: Breakdown of causes for mispredictions of the ∇_c bitmap when the static bank aware scheme is used.

implementations of our bank aware dynamic loop scheduling scheme. The bar marked using ”+1” tries one more alternative over our second attempt in the default implementation, whereas the bar marked using ”+n” tries all possible alternatives. We see that the average energy savings brought by the ”+1” scheme and ”+n” scheme are 19.1% and 21.9%, respectively. Considering these values with our default value (16.4%), we can conclude that our default implementation is not far from them, as far as energy consumption is concerned. In addition, although not presented here in detail, the ”+n” scheme increased the execution cycles by nearly 6% over our default implementation, making the latter even more promising option, when energy consumption and execution cycles are considered together.

6.5 Conclusion

In this chapter, we presented a memory bank-aware dynamic loop scheduling scheme. Our approach selects the set of iterations to assign to a requesting processor such that the currently active banks are reused if possible (without activating a new bank). We tested this approach and collected both performance and energy numbers using a SIMICS based simulation platform. In our evaluation, we also compared it to a number of previously published loop scheduling schemes, including the pure static and dynamic schemes, variants of dynamic scheme, as well as two locality oriented loop scheduling approaches. Our experimental results with six embedded applications clearly show that the proposed scheduling scheme not only reduces the energy consumption significantly, but it also leads to much better energy savings when compared to these prior techniques and it is competitive with the loop scheduler that generates the best performance.

Until now we have focused on CMP systems with different memory structures, including cache, SPM and on-chip banked memory. Although the memory subsystems were different, these systems were similar in that, they had a bus for communication. In the next chapter, we will focus on 2-D mesh based CMP systems, which use a grid-based Network-on-Chip architecture.

Chapter 7

Integrated Code and Data Placement in 2-D Mesh Based CMPs

7.1 Introduction

In this chapter, we present a static code and data placement scheme. We target two-dimensional mesh based chip multi-processor (CMP) architectures. Each node of this CMP contains a processor core, an on-chip memory component, and a network interface which connects the node to its neighbors. We assume that the compiler/programmer manages the on-chip memory space and thread-to-core assignments. In this architecture, cost of a data access depends on the distance between the requesting core and requested data.

As CPU design has become severely power limited, it is now commonly accepted that staying on the current performance trajectory will come about through the integration of multiple processors on a chip rather than through increases in the clock rate of single processors. Once the number of CPUs on one chip passes some threshold ($\sim$8 CPUs), these future chip multiprocessors (CMPs) will require an on-chip network (an NoC, Network-on-Chip) in order to be able to handle the required communications between the CPUs in a scalable, flexible, programmable, and reliable fashion. With this network-on-chip-based CMP (NoC-based CMP) as the computing platform, a very rich set of research challenges arise. Circuit and architectural challenges such as router design, IP placement, and sensor placement are currently being studied in the context of CMPs in both industry and academia, as is evident from recent publications [78, 43, 47]. In comparison, the work on programming and compiling for these architectures has received considerably less attention. Unfortunately, unless critical software issues such as programming language support, application mapping, data placement, and compiler support are adequately addressed, CMPs may not be able to deliver

promised performance levels.

Motivated by this observation, we propose a compiler directed code and data placement scheme for two-dimensional mesh based CMP architectures. The proposed approach uses a Code-Data Affinity Graph (CDAG) to represent the relationship between loop iterations and array data and then assigns the loop iterations to processing cores and sets of data blocks to on-chip memories. During the mapping process, the on-chip memory capacity and load imbalance across different cores as well as the topology of the NoC are also taken into account.

We assume that code parallelization has already been applied prior to our approach, using any known technique. In fact, the choice of the code parallelization scheme used is orthogonal to the main focus of our approach, which performs code and data placement. Therefore, in principle, our approach can work with any code parallelization strategy. Note that the data dependencies across loop iterations will be taken care of during code parallelization and, consequently we assume that the iteration blocks (used in our approach) do not have data dependencies among them.

We propose two variants of our integrated code-data mapping approach: Depth First Placement (DFP) and Breadth First Placement (BFP).The experimental evaluation shows that our CDAG based placement schemes are very successful in practice, achieving average performance improvements of 19.9% (DFP) and 16.8% (BFP), and average energy improvements of 29.7% (DFP) and 27.8% (BFP).

The rest of this chapter is structured as follows. The next section discusses related work. Section 7.3 explains the CDAG structure. Section 7.4 presents the details of our approach. An experimental evaluation is given in Section 7.5. Finally, we conclude in Section 7.6 by a summary.

7.2 Related Work

Automatic computation and data decomposition using compilers has not been effective with large codes, although there are myriad previous works attempting to achieve this [5, 91, 71, 110]. Mapping applications for NoC based architectures using compilers has also been a hot research issue in the recent past [44]. Kuijlman et al [61] present a compiler framework that takes a program with partial work and data placement information, and transforms it into an explicit parallel program optimized for the amount of communication. [16] compares performance impact of alternate code and data mapping strategies on a 64 node IBM RP3, but this technique is very architecture specific and does not extend easily to chip multiprocessors. Lowenthal and Andrews [75] describe an adaptive system that takes an initial data placement, and changes the placement whenever a monitor indicates that a different placement would perform better. Performance overheads of

such monitoring as well as modifying of placement is avoided by a good compiler based approach. Moreover, affinity between code and data has not been considered or its impact on the performance has not been studied well in the previous works.

7.3 Code-Data Affinity Graph

Our compiler based placement scheme employs a data structure called the *Code-Data Affinity Graph (CDAG)*. CDAG is essentially a bipartite graph $G(V_1, V_2, E)$, where V_1 represents iteration blocks, V_2 represents data blocks, and E captures the access relationship between iteration and data blocks. In this context, an *iteration block* is a set of consecutive loop iterations that belong to the same loop nest and a *data block* corresponds to a set of consecutive elements that belong to the same array. An edge $e \in E$ indicates that at least one of the iterations in V_1 accesses at least one of the data elements in V_2. This is referred to as an *access relationship*. Note that CDAG is built by the compiler and used for data code placement.

Clearly, an important question is how to determine the iteration and data blocks to use and how to extract the access relationships from the application code. Our approach is flexible in the sense that it can work with any iteration/data block size. Consider, for instance, that a given array is divided into data blocks of size L_d and an iteration space is also divided into blocks of size L_i. Note that, in general, L_d may be different from L_i. As an example, consider the following simple loop, written in a pseudo language, where two one-dimensional arrays are accessed:

```
for j = 2, N-1
        b[j] = a[j+1] + a[j-1] + a[j] ;
```

Figure 7.1(a) shows how the loop iterations (on the left) access data elements (on the right) in this loop (we focus only on array a for illustrative purposes). In this code fragment, we can identify a block with its first element; that is,

$$B_{i,p} = \{j | \max 2, p \leq j < \min\{p + L_i, N - 1\}\}$$

$$B_{d,q} = \{k | \max 1, q \leq k < \min\{p + L_d, N\}\},$$

where $B_{i,p}$ and $B_{d,q}$ correspond to p^{th} iteration block and q^{th} data block, respectively, assuming that array a has N elements. If the number of loop iterations is not divisible by L_i or L_d, the remaining elements are placed in a new block with fewer iterations or data elements; this would not affect the performance in any significant way as in general L_i and $L_d << N$.

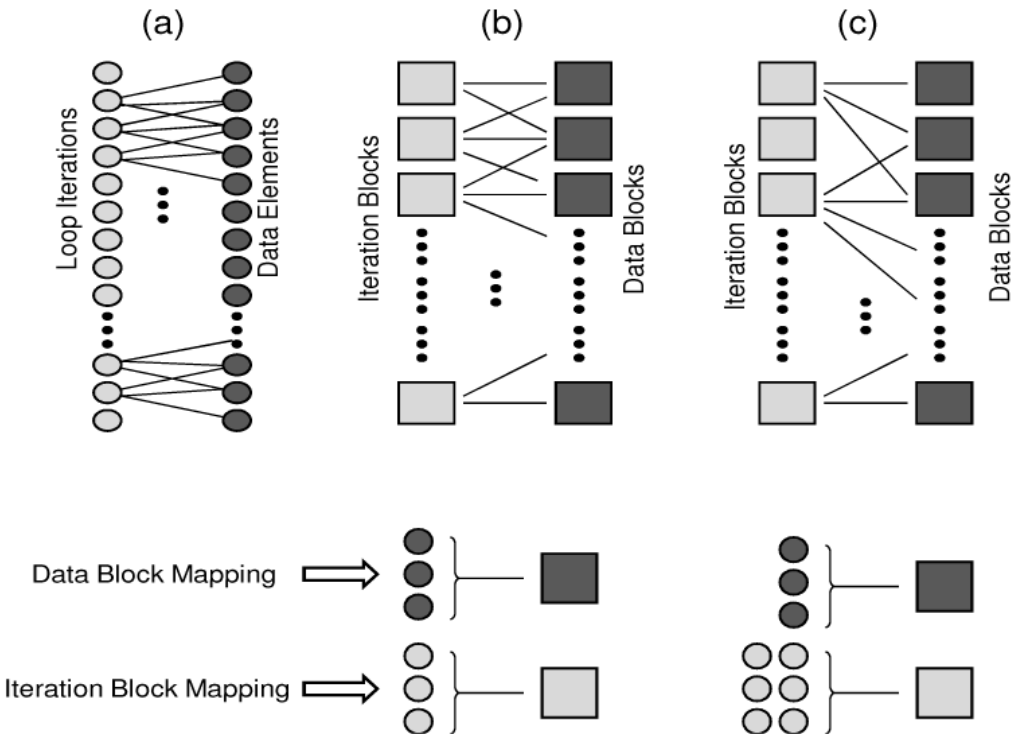


Figure 7.1: Pictorial representation of loop iteration elements, data elements and construction of iteration blocks, data blocks and access relationships.

In the loop shown above, each iteration j accesses three data elements of array a. Note that, the data elements accessed by a given j can belong to the same or different data blocks. In formal terms, we can define an access relationship $\Delta(B_{i,p}, B_{d,q})$ between data block $B_{d,q}$ and iteration block $B_{i,p}$ as follows:

$$\Delta(B_{i,p}, B_{d,q}) = \begin{cases} 1, & \text{if } \exists j, k, R \text{ such that } j \in B_{i,p} \ \& \\ & k \in B_{d,q} \ \& \ R \in \mathcal{R} \ \& \ R(j) = k \\ 0, & \text{else.} \end{cases}$$

where $\mathcal{R}$ refers to the set of array references in the code. Informally, $\Delta(B_{i,p}, B_{d,q})$ takes a value of 1 if there is an iteration in $B_{i,p}$ that accesses a data element in $B_{d,q}$. It should be noted that the access relationships between iteration blocks and data blocks can be represented using a bi-partite graph. For example, assuming L_i and L_d are each set to 3, the bi-partite graph in Figure 7.1(b) represents the access relationships implied by the code fragment above. This graph is referred to as the *Code-Data Affinity Graph (CDAG)*, and is the main data structure built and used by our compiler for optimizing code and data placement. For ease of representation, we use V_1 to represent the set of nodes that contain the iteration blocks, and V_2 for the set of nodes that contain the data blocks. Note that the two nodes, $B_{i,p}$ and $B_{d,q}$, of a CDAG have an edge between

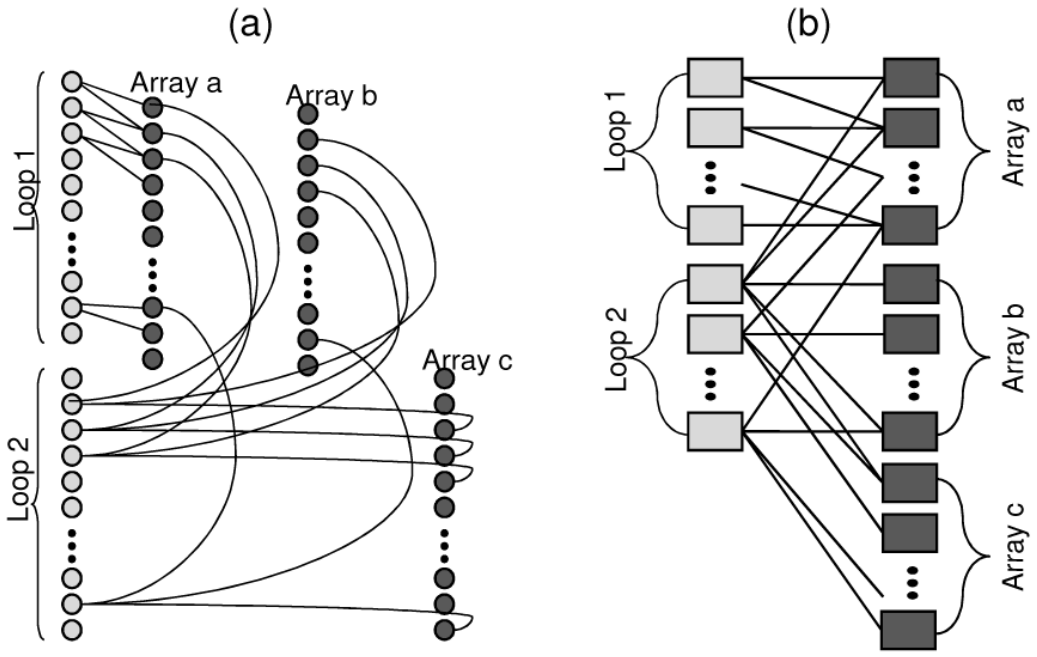


Figure 7.2: Pictorial representation of loop to data mapping (a) and corresponding CDAG for a case where two loops access three arrays (b).

them if and only if $\Delta(B_{i,p}, B_{d,q})$ is 1. We also associate a weight, $\omega_{i,d}$ with each edge $e_{i,d}$ (from iteration block represented by $v_i \in V_1$ to data block represented by $v_d \in V_2$) of a CDAG, and this weight captures the total number of elements from V_2 accessed by the iterations in V_1.

An important point about CDAG is that it can take a different shape when the value of L_i and/or L_d is changed. For example, Figure 7.1(c) shows another CDAG for the same code fragment above, this time with L_i and L_d values of 6 and 3, respectively. We renumber all loop iterations and data blocks in the application code such that the only ids that we use during the optimization (placement) process are $B_{i,p}$ and $B_{d,q}$. Figure 7.2(a) shows loop-iterations-to-data mapping and Figure 7.2 (b) gives the CDAG that corresponds to the code fragment below where two separate loops manipulate five different data arrays (again, for illustration, we focus only on arrays a, b, and c).

```
Loop1: for j = 1, N-1
       d[j] = a[j+1] + a[j+1] + a[j] ;
Loop2: for i = 2, N-1
       e[i] = a[i-1] + b[i] * c[i+1] ;
```

One can potentially build a CDAG for the entire application program or divide a program into disjoint code regions such that no two code regions share any data block between them and build a separate CDAG

for each such region to optimize code/data placement independently. This is the approach taken in our current implementation.

7.4 Our Approach

Our code and data placement algorithms for CMP architecture use CDAG. In our placement algorithms, iteration blocks are units of code placement, and similarly, data blocks are units of data placement. Integrated code-data placement decides the iterations blocks that will be mapped to CPUs for execution and data blocks that will be stored in the on-chip memories attached to the CPUs. In this section, we describe two algorithms for this integrated placement problem.

7.4.1 Depth First Placement

Our first algorithm, called the *depth-first placement (DFP)* algorithm, performs code and data placement for each core in turn. The DFP algorithm starts with a node which can be an iteration block node (that belongs to V_1) or data block node (that belongs to V_2). Without loss of generality, let us assume that it is an iteration block node ($v_x \in V_1$). In the next step, we select a node $v_y \in V_2$ such that $\omega_{x,y} \geq \omega_{x,z}$ for any $v_z \in V_2$ that is connected to v_x. In other words, we proceed from the iteration block node to a data block node whose associated edge has the highest weight among all alternatives. In the next step, we move from v_y to $v_n \in V_1$ such that $\omega_{n,y} \geq \omega_{m,y}$ for any $v_m \in V_1$ that is connected to v_y. That is, we proceed from the data block node to an iteration block node whose associated edge has the highest weight among all alternatives. This ping-pong style movements between V_1 and V_2 continue in this fashion. Each time we move to an iteration block node v_i, we add its size $|v_i|$ to C, and similarly, each time we move to a data block node v_j, we add its size $|v_j|$ to C' (both C and C' are initialized to 0). This process continues until either $C \geq C_{ideal}$, where C_{ideal} represents the maximum allowable load on the core or $C' \geq C'_{ideal}$, where C' ideal represents the on-chip memory capacity for a node. Once either of these is reached, the data blocks visited so far are assigned to be stored in a core's memory and the iteration blocks visited so far are assigned to the same core for execution. The important point to note here is that the loop iterations and data elements assigned to a core using this approach exhibit a certain degree of affinity, i.e., the iterations mostly use the data elements assigned to the memory attached to the same core.

After the code and data placements for the first core are complete, we move to a neighboring core and carry out its code and data assignment. However, in doing so, our approach takes into account both the topology of NoC in question and the code-data placements that have already been performed up to this point.

Algorithm 7 Depth First Placement Algorithm

Input: A mesh based CMP, a weighted bipartite graph $G = (V, E)$ and a vertex $v_i \in V$, where $V = \{V_1$ (iteration blocks) $\bigcup V_2$ (data blocks) $\}$.
Output: Placement of code and data on appropriate cores and local memories.

$V_c = V_c' = \phi;\ C = C' = 0;$
Select a free core, C_f , that is a neighbor of already processed cores (if any).
Create a Stack, S and $Push(S, v_x)$
while $C \leq C_{ideal}$ && $C' \leq C'_{ideal}$ && $\mathbf{not}(S.empty())$ **do**
    $v_x = Pop\_highest\_priority(S)$, Mark v_x as "$placed$"
    if $v_x \in V_1$ **then**$V_c = V_c \bigcup \{v_x\}; C = C + |v_x|;$
    else$V_c' = V'c \bigcup \{v_x\}; C' = C' + |v_x|;$
    end if
    for all v_y, such that $(v_x, v_y) \in E$ selected in reverse order of priority **do**
        if v_y is "$non - placed$" **then**
            $Push(S, v_y)$
        end if
    end for
end while
Place V_c on processor C_f and V_c' in memory of C_f
Select a "$non - placed$" node v_i such that, $\forall v_p$, such that v_p is marked "$placed$" and $(v_i, v_p) \in E$ and $\exists v_p$ marked "$placed$", $\omega_{p,i}$ is the maximum
if such a node is found **then**
    goto 1.
elseTerminate.
end if

More specifically, we always select the core to process next from among the ones that are neighbors to already processed cores, and the CDAG node to start (for the new core) is selected such that it is a neighbor of one of the traversed nodes (during processing the previous core) and the edge that connects it to the already traversed nodes has the highest weight among all alternatives. This ensures that we start our graph traversal for the second core with either (1) a data block node that is frequently accessed by an iteration block node already assigned to a neighboring node, or (2) an iteration block node that frequently accesses a data block node already assigned to a neighboring node. Then, we traverse the bi-partite graph for this second core using a similar ping-pong like strategy explained above. A sketch of our DFP algorithm is given in Algorithm 7.

7.4.2 Breadth First Placement

Like the DFP algorithm, our second algorithm, called the *breadth-first placement (BFP)* algorithm, also performs code and data placement for each core in turn. However, it traverses the bi-partite graph in a

breadth-first fashion. It starts with a node which can be an iteration block node (that belongs to V_1) or data block node (which belongs to V_2). Let us assume, that it is an iteration block node ($v_x \in V_1$). In the next step, we select a set of nodes $\nu \subseteq V_2$ such that, for all $v_y \in \nu$, (x, y) is an edge in the CDAG. That is, we proceed from the iteration block to *all* the data blocks that are connected to that iteration block. In the next step, we start with ν and include all iteration block nodes that are connected to the data block nodes in ν. In the next step, we consider all the data block nodes that can be reached from $\mathcal{I}$, the set of iteration block nodes. This process continues until we exceed the thresholds of C_{ideal} or C'_{ideal}. If this happens, instead of including all the nodes, we include only a subset of them (to maximize locality, we select the ones with the largest weights). Once we are done with the placements for the first core, we move to a neighboring core and repeat the procedure explained above. Algorithm 8 presents a sketch of the BFP algorithm.

Algorithm 8 Breadth First Placement Algorithm

Input: A mesh based CMP, a weighted bipartite graph $G = (V, E)$ and a vertex $v_i \in V$, where $V = \{V_1$ (iteration blocks) $\bigcup V_2$ (data blocks) $\}$.
Output: Placement of code and data on appropriate cores and local memories.

$V_c = V'_c = \phi;\ C = C' = 0;$
Select a free core, C_f , that is a neighbor of already processed cores (if any).
Create a Priority Queue, Q and $Enqueue(Q, v_x)$
while $C \leq C_{ideal}$ && $C' \leq C'_{ideal}$ && $\mathbf{not}(Q.empty())$ **do**
    $v_x = Dequeue(Q)$, Mark v_x as "$placed$"
    if $v_x \in V_1$ **then**$V_c = V_c \bigcup \{v_x\}; C = C + |v_x|;$
    else$V'_c = V'c \bigcup \{v_x\}; C' = C' + |v_x|;$
    end if
    for all v_y, such that $(v_x, v_y) \in E$ **do**
        if v_y is "$non - placed$" **then**
            $Enqueue(Q, v_y)$
        end if
    end for
end while
Place V_c on processor C_f and V'_c in memory of C_f
Select a "$non - placed$" node v_i such that, $\forall v_p$, such that v_p is marked "$placed$" and $(v_i, v_p) \in E$ and $\exists v_p$ marked "$placed$", $\omega_{p,i}$ is the maximum
if such a node is found **then**
    goto 1.
elseTerminate.
end if

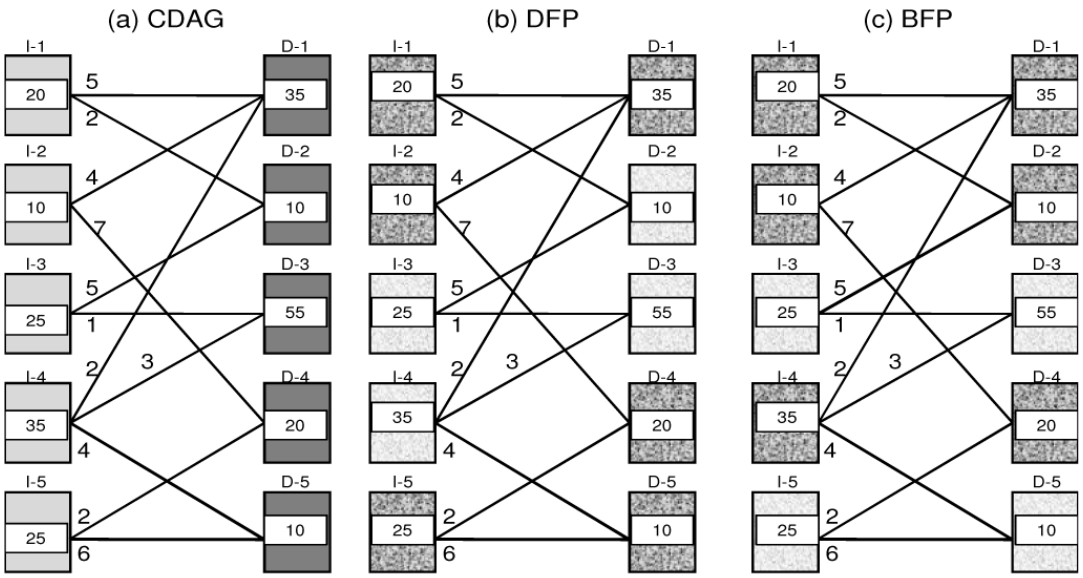


Figure 7.3: (a) Illustration of a simple CDAG. (b) Partitions corresponding to DFP. (c) Partitions corresponding to BFP. The target system consists of two processors, C_{ideal} = 75 and C'_{ideal} = 60.

7.4.3 Example

To better illustrate the difference between the DFP and BFP algorithms, we now go over a simple example. Figure 7.3 (a), depicts a simple CDAG with associated weights and sizes. Sizes of nodes are different in this example to illustrate the most general case. Our DFP scheme is depicted in Figure 7.3 (b). The algorithm starts from node I-1. The edge with the highest weight from I-1 is the edge to D-1 which is taken. Similarly, at every further node, the edge incident on the target node with the highest weight and is yet to be taken is selected (D-1 to I-2) . At each step, based on whether the node represents an iteration block or data block, corresponding sizes are accumulated in C and C'. In our example, when the edge I-5 to D-5 is taken, C' accumulates a total of 65 (data block sizes of 35, 20 and 10), which is greater than C'_{ideal}. This indicates the end of accumulation of affine code and data for one core. In order to move to the next core, we select a neighboring core to an already processed core. In this simple example, it happens to be the second core. The node from which the new traversal starts is determined as the node connected with the edge with the highest weight from any node that is already processed. In our example, it is the node connected with the highest weight to any node in the partition $\{$I-1, I-2, I-5, D- 1, D-4, D-5$\}$. I-4 is connected to D-5 with a weight of 4 which is the highest. Therefore, we start our next traversal from I-4 and accumulate the rest of the nodes to be placed in the second processor. Note that, in DFP, every partition is formed by alternating between an iteration block and a data block if a partition is completely selected without retracing any path. Therefore, in

Architecture	5 × 5 2D mesh
Core	two-issue
Data/Instr L1 Capacity	8KB (per node)
Local On-Chip Memory	512KB (per node)
Link Speed	1GHz
Link Activation Latency	1 μsec
Link Activation Energy	140 μjoule
Packet Header Size	3 flits
(L_i, L_d)	(20, 20)

Table 7.1: Default values of our simulation parameters

a given partition, the number of iteration blocks could differ from the number of data blocks by any degree in general but only one in case the entire partition is selected over a single trace.

On the other hand, the operation of our BFP scheme is depicted in Figure 7.3 (c). In this scheme, all the nodes connected to the node being processed are traversed first. In our example, starting from I-1, all nodes connected to I-1, namely D-1 and D-2 are traversed in order of corresponding edge weights. Accumulation of sizes happens as in depth first placement. The node dequeued from the priority queue is D-2. Therefore, all nodes that are not marked and are connected to D-2 are traversed in the next phase. This continues until D-4 is accumulated and C' becomes greater than C'_{ideal}. Note that the partition derived here is different from the DFP case and also, in general, that the partitions can have any number of iteration or data blocks and the scheme does not impose any relationship between them.

Before moving to our experimental evaluation of these two algorithms and their quantitative comparison to alternate placements, we want to emphasize that it is not clear which one of these two placement algorithms is better than the other. Depending on the program code to be analyzed, one of them may be better than the other. The advantage of BFP over DFP is that it covers all data blocks (resp. iteration block) accessed by an iteration block (resp. data block) in a single shot, whereas the DFP algorithm may not be able to include all those blocks. The drawback of BFP on the other hand is that some of such quickly-included blocks may not be the best candidates (for the current core being processed) in the long run as the corresponding edge weights may not be very high. In our experiments, we evaluate both these algorithms.

7.5 Experiments

In this section, we introduce our experimental setting and present the data collected from the evaluation of our proposed placement schemes. To conduct our experiments, we implemented a flit-level network-on-chip

simulator (built on top of Orion [115]) and connected it with SIMICS [108], a multi-processor simulator. The network is parameterized in a similar fashion to that in [28]. The link speed is set to 1Gb/sec. Each input port of switch has a buffer that can hold 64 flits, each of which is 128 bits wide (packet size is 16 flits).

7.5.1 Setup

We used all the benchmarks from the SPECOMP suite [8] to evaluate the proposed integrated code-data placement schemes. For each benchmark, after fast-forwarding 1 billion instructions, we collected statistics for the next 2 billion instructions. To compare different approaches proposed to alleviate the code-data placement problem, we implemented several schemes in our experimental framework and quantified their impact. The schemes tested in this work can be summarized as follows:

- Code-Only: In this scheme, the data are distributed across the on-chip memory components in a round-robin fashion, but the code distribution is carried out taking into account the data distribution. In other words, loop iterations are assigned to processing core to maximize data locality (for the round-robin data distribution).
- Data-Only: This is the dual of the previous scheme. The code assignment across cores is performed in a round-robin fashion. However, the data-to-memory assignment is done such that data locality is maximized as much as possible based on the round-robin iteration distribution.
- DFP and BFP: These are the implementations of the integrated code-data placement algorithms described in this chapter.
- Topology Agnostic: This is a previously proposed integrated code-data placement scheme [5]. The fundamental difference between our scheme and this work is that the latter does not take the network topology into account in performing code-data placement. However, to our knowledge, this scheme represents the state-of-the-art in compiler-directed code and data placement in multiprocessor machines.

Each loop in the application has been analyzed by the compiler (SUIF [4]) and transformed such that outermost loop parallelism is obtained to the extent allowed by data dependencies, i.e., from each loop nest, the compiler parallelizes the outermost loop that does not carry any data dependency. This strategy tends to minimize inter-processor synchronization and helps reduce execution time. It also represents, in our opinion, the state-of-the-art in loop-level code parallelization.

We make experiments with Code-Only and Data-Only schemes to show that, for the maximum performance and energy benefits, code and data placement should be carried out in a synergistic fashion. Also,

our experiments with the scheme proposed in [28] are aimed at revealing the importance of considering the CMP topology during code-data placement. All these versions are implemented using the SUIF compiler framework [4]. The additional compilation time increases brought by our DFP and BFP schemes over the Code-Only scheme were 14.7% and 12.8%, respectively, on average. The longest compilation time we witnessed during our experiments was 1.6 minutes, which is not too high in our opinion, considering that compilation is basically an off-line process. Also, the additional code size increases brought by DFP and BFP (again over the Code-Only scheme) were, on average, 6.6% and 4.7%, respectively. Consequently, we did not observe any degradation in the instruction cache performance.

Table 7.1 gives the major simulation parameters and their default values. Under these parameters, the execution times for our applications under the Code-Only scheme varied between 55.7 seconds and 1.6 minutes, and their energy consumptions varied between 118 mJ and 809 mJ. The performance and energy numbers presented in the remainder of this section are *normalized* with respect to the corresponding values obtained under the Code-Only scheme. Unless otherwise stated, the code and data block sizes are equal to 20 (iteration and data elements).

7.5.2 Results

Our first set of results give normalized execution latencies and are presented in Figure 7.4. We first observe that our placement schemes generate better results than the Code-Only and Data-Only schemes. In fact, the average latency improvements the DFP algorithm brings over Code-Only and Data-Only are 19.9% and 22.0%, respectively. In galgel, the default (round robin) code distribution performs reasonably well, and therefore, the results obtained using our algorithms and Data-Only are close to each other. Similarly, in mgrid, our approach and Code-Only generate very similar results. Secondly, in general, for this set of applications, DFP generates better results than BFP. The third major observation one can make from these results is that there is more than 10% difference between the average improvement brought by DFP and that obtained using the Topology

Agnostic scheme (19.9% versus 9.5%), emphasizing the importance of considering on-chip network topology during code-data placement. This difference between our placement schemes and the Topology Agnostic scheme can be explained as follows. The Topology Agnostic scheme distinguishes between only two types of data localities: local and remote. The former of these refers to the case when a thread accesses the data element it requires from its local on-chip memory. The latter on the other hand captures the remaining accesses, i.e., accesses to data that do not reside on the local memory. In an NoC based platform however, the exact location of the remote (non-local) data in the chip matters and this is where our schemes bring

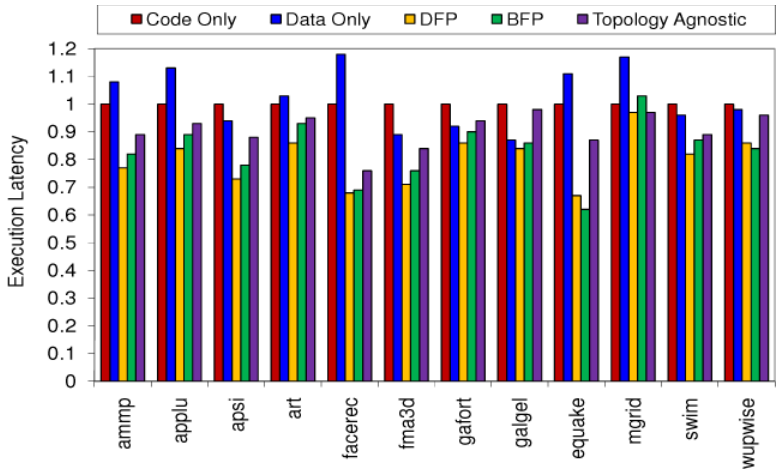


Figure 7.4: Normalized execution latencies.

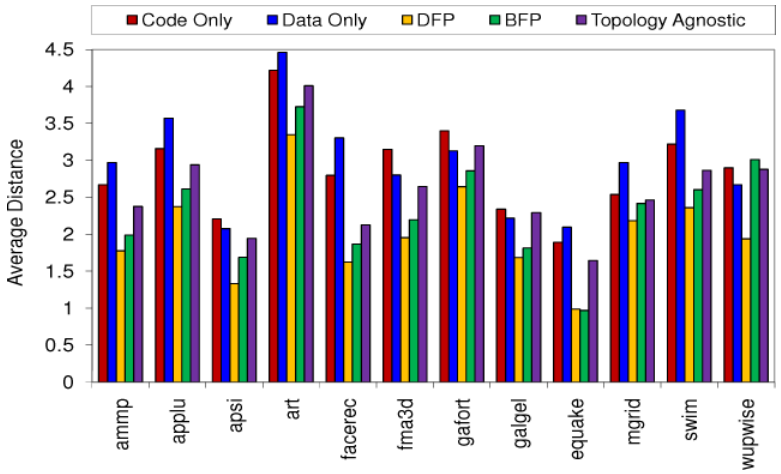


Figure 7.5: Thread-to-data access distances (average values).

additional benefits over the approach in [5].

To better explain these improvements in execution latencies, we present in Figure 7.5 the average thread-to-data access distances. What we mean by this is the number of NoC links for a thread to access a data element (when averaged over all data accesses executed by the application). We see that the lowest distances are achieved by our schemes (DFP and BFP). As a result, our schemes result in lower execution latencies than the other schemes tested.

Figure 7.6 gives the performance improvements brought by the DFP scheme under different on-chip memory sizes and core counts. Recall from Table I that the default mesh topology we use is 5 5 and the default on-chip memory capacity attached to each node is 512KB. The results presented in Figure 7.6 show that the effectiveness of our scheme improves with larger number of cores and smaller on-chip memory

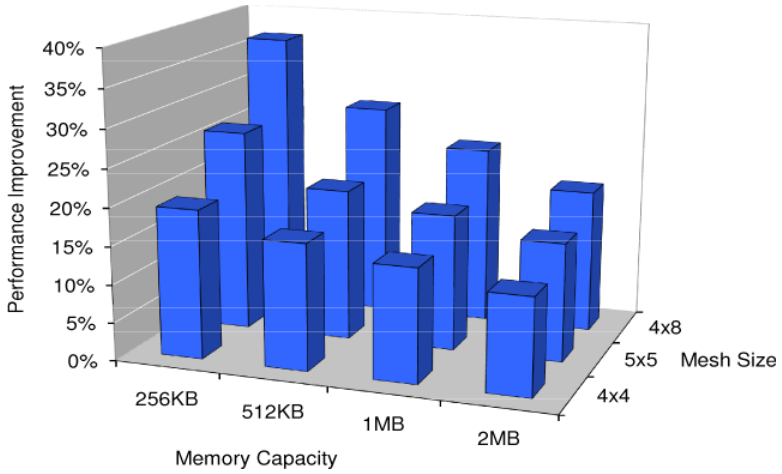


Figure 7.6: Normalized performance improvements under the different mesh sizes and on-chip memory capacities.

sizes. The reason for the first one is because a larger network leads in general to worse behavior as far as the Code-Only scheme is concerned. Since the results with our scheme are normalized with respect to those of the Code-Only scheme, we witness an improvement with larger networks. The reason for the observed improvement with smaller on-chip memories is that larger memories typically hide the inherent weaknesses of simple schemes (such as Code-Only). In fact, a very large on-chip memory space may even obviate the need for sophisticated locality optimizations. However, we want to mention that both of these observed trends from Figure 7.6 are encouraging. It is projected that, in the long run, both the number of cores and on-chip memory sizes in CMPs will increase. However, it is also the case that the increases in the amount of data processed by parallel applications outpace the increase in memory capacities. Therefore, schemes such as ours which favor smaller memory sizes (as compared to application dataset sizes) and large number of cores will be more popular in the future.

While these performance improvements brought by our approach are good, it is also important to evaluate the energy consumption of the different schemes. This is particularly important in battery-operated execution environments. We assume that the communication links in the NoC can be shutdown independently, using a time-out based mechanism as described in [109] (to be fair in our evaluation of different schemes, this mechanism is used in all the schemes tested). We set the time-out counter threshold for the hardware-based power reduction scheme to 1.5μsec based on some preliminary analysis. The time it takes to switch a link from the power-down state to the active state is set as 1μsec, and the energy overhead of this switching is assumed to be 140μJ based on the prior research. Since the network energy model employed is not a major contribution of our work, we do not elaborate on it any further. For modeling the energy consumption of memory components, we used CACTI [107], and for collecting energy data for core-related activities, we

enhanced SIMICS with accurate timing models and Wattch-like energy models [15]. All the energy numbers presented below include both dynamic energy and leakage energy consumed in computation, inter-thread communication, on-chip and off-chip memory accesses. Also, as in the case of performance, the results we report include all the overheads incurred by our placement schemes.

The energy consumption results for the different code-data placement schemes are presented in Figure 7.7 under the default values of the simulation parameters in Table I. As compared to the performance results given in Figure 7.4, we see that the energy savings are higher. More specifically, the DFP scheme achieves an average 29.8%, 28.5% and 16.8% improvement in energy consumption over the Code-Only, Data-Only and Topology Agnostic schemes, respectively. The corresponding average energy savings obtained using the BFP schemes are 27.7%, 26.4% and 14.3%. That is, our schemes are very effective in reducing total energy consumption. The reason why our energy savings are higher than our performance savings can be explained as follows. In parallel execution, some remote data accesses (within chip) can be hidden during parallel execution (that is, although its latency is not eliminated, it is hidden in parallel execution). However, while performance overheads can be hidden, energy overheads cannot be. As a result, our approach brings larger energy savings compared to its performance benefits (recall that all the results in Figures 7.4 and 7.7 are normalized with respect to the first bar (Code-Only) for each application).

Our code and data placement approach works with the given iteration block and code block sizes. Recall that the default values used in our experiments so far was 20. We also conducted experiments with different block sizes and the performance results are presented in Figure 7.8 (Energy results were similar and hence not shown explicitly). In this graph, each (L_i, L_d) point on the x-axis indicate the (iteration block size, code block size) pair used. Consequently, our default operating point is (20,20). Each bar in this plot represents the average values over all twelve benchmarks. We can make several interesting observations from these results. First, the results with (10,10), (20,20) and (50,50) are similar to each other. By comparison, when we move to (100,100) the savings we achieve start to get reduced, and (though not shown in this bar-chart), the larger blocks resulted in even worse behavior (power and performance). This result motivates for using smaller block sizes. This in our opinion makes sense because larger block sizes tend to blur the affinities between individual iterations and data elements, and this tends to generate results which are more on the sub-optimality side. Another interesting observation from Figure 7.8 is that (10,20) and (20,10) generate worse results than both (10,10) and (20,20). This result says that, unless there is a reason, we should not work with cases where the iteration block sizes and data block sizes differ.

We also made experiments to compare our performance and energy savings to the optimal savings that can be achieved under the given code and data block sizes. To obtain the optimal savings, we used a profile based approach. In this approach, we first profiled the application code to determine the relationship between

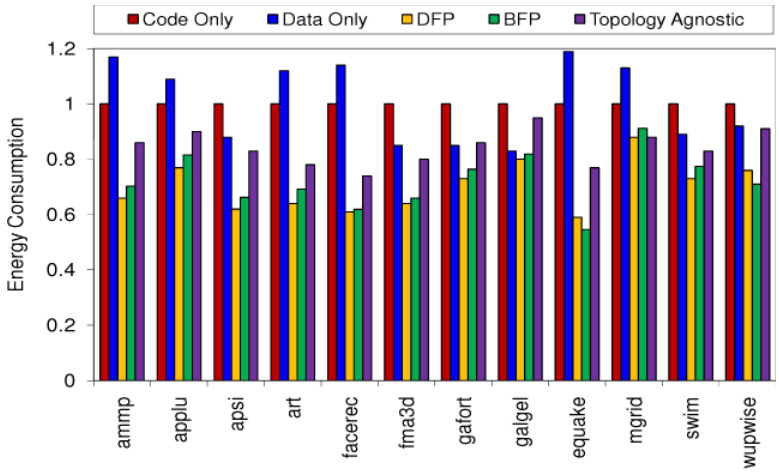


Figure 7.7: Normalized energy consumption values.

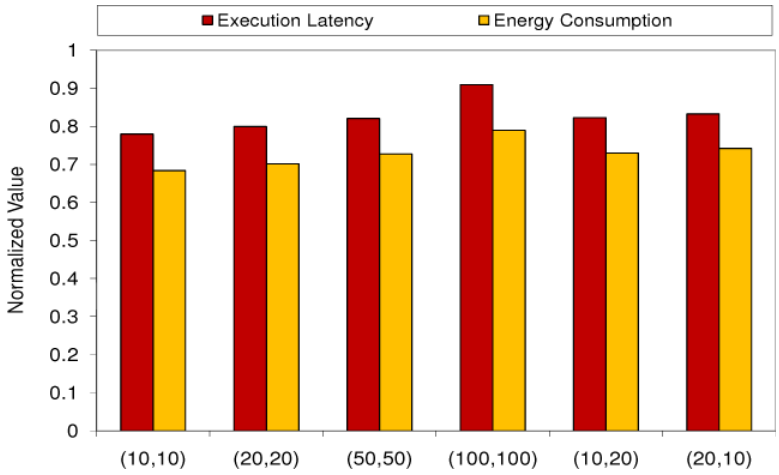


Figure 7.8: Sensitivity to the iteration block and data block sizes.

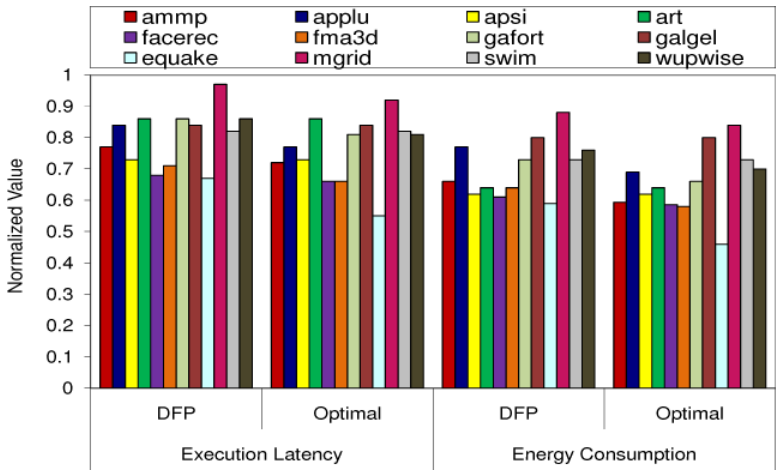


Figure 7.9: Comparison of the DFP scheme with the optimal code-data placement.

code and data blocks, and then formulated an ILP (integer linear programming) based solution, which gives the code and data placements for all cores. The results are shown in Figure 7.9 for both performance and energy. An interesting observation is that in four applications (apsi, art, galgel, and swim), DFP and the optimal placement scheme produced the same result, and in facerec, the difference between the two schemes was very low. When all twelve applications are considered, we see that the optimal placement is 4.8% better than our scheme from a performance viewpoint and 6.3% from an energy viewpoint. Therefore, we can conclude that our approach in general comes very close to the optimal code-data placement. Also, we want to mention that an ILP based solution may not be feasible in general due to the large number of variables and constraints involved in the code-data placement problem. In fact, we observed during the experiments that in some cases the solution time taken by the linear solver was as high as 18 hours on a 2 GHz machine. Therefore, when both solution time and solution quality are considered together, we believe that our approach strikes a good balance.

7.6 Conclusion

We present an integrated code and data placement scheme targeting two-dimensional mesh based CMPs. The proposed approach uses a novel (compiler based) data structure called the Code-Data Affinity Graph (CDAG) to represent the relationship between loop iterations and array data and then places the sets of iteration blocks to processing cores and sets of data blocks to on-chip memories. We evaluated two different variants of this approach and compared them against three alternate code and data mapping schemes. The results indicate that our CDAG based placement schemes achieve average performance improvements of 19.9% (DFP) and 16.8% (BFP), and average energy improvements of 29.7% (DFP) and 27.8% (BFP).

Chapter 8

Conclusion and Future Work

8.1 Conclusion

We used different algorithms to optimize performance, memory usage and energy consumption on different embedded system architectures utilizing data compression, data placement, code placement, code scheduling and SPM management approaches.

In Chapter 2, we discussed a compiler-based strategy that uses compressed arrays for saving memory. We saw in our experiments that it improved performance and reduced memory usage, and it increased the executable size due to code restructuring to work with compressed arrays. Though it requires a particular type of data locality to be effective, it provides impressive results with suitable applications.

In Chapter 3, we discussed a compiler-based code scheduling scheme for shared cache CMP systems, which considers both parallelism and data locality at the same time. Our experimental results indicated that our approach provides significant performance improvement, and gets very close to the performance of the ILP solution.

In Chapter 4 we discussed an SPM-aware static loop scheduling strategy for CMP systems with a shared SPM. Our experimental results indicate that the proposed approach brings a performance improvement over previous strategies.

In Chapter 5 we discussed an SPM management approach using Markov chain based data access prediction for irregular accesses that have hidden data reuse. Our experimental results indicate that our approach brings a significant performance improvement in a set of applications with both regular and irregular access patterns.

In Chapter 6 we discussed a memory bank aware dynamic loop scheduling scheme that minimizes mem-

ory energy consumption in CMP systems with banked memory. This leads to reduction of overall energy consumption, and this is important, especially in battery-operated embedded systems. Our experimental results show that the proposed scheme leads to much better energy results when compared to prior techniques and is also competitive in performance.

In Chapter 7 we discussed a compiler directed integrated code and data placement scheme for 2-D mesh based CMP architectures. Our approach assigns sets of loop iterations to processors and sets of data blocks to on-chip memories, taking into account the on-chip memory capacity and load imbalance, as well as the topology of the NoC. Our experimental results show that our CDAG based placement scheme brings improvements in both performance and energy consumption.

8.2 Future Work

There are two interesting avenues for future work. The first is to derive new methods by combining the discussed approaches wherever possible. We will discuss a possible application in section 8.3. The second is to try to apply our approaches in a different target platform, where the system shows similar characteristics to those discussed. We will discuss this in section 8.4.

8.3 Data Locality and SPMs

Our work mainly consists of data placement and code scheduling in multiprocessor systems. In chapter 3 we discussed a data locality based code scheduling method that makes no assumptions about the memory hardware. In chapters 4 and 5, we discussed SPM related approaches.

A combination of these approaches would be to transform the code scheduling algorithms of chapter 3 in such a way that they will work in an SPM-conscious manner; meaning, the approach will be aware of the embedded system's underlying memory hierarchy, and try to make use of this information to further speed up memory accesses. We see two applications for this approach.

The first is to determine the best schedule, given a memory hierarchy and code to optimize. This tool will take as input a memory hierarchy and code, and give as output a schedule and a mapping of data to the SPMs.

Of course, it makes sense to optimize an application for a system, but there are cases where the focus is on the software, and not the hardware. There are many situations where a system is built for a single application. An embedded system that controls the brake system in your car, or the hardware for a handheld

Tetris game are not concerned with providing overall good performance for general purpose applications; they run a single application and nothing else. The manufacturer of such a system has to make a decision on which hardware to choose for the mission critical application.

So the second application is to determine best memory hierarchy for a given application. This tool will take as input a code, and give as output a memory hierarchy, a schedule and a mapping of data to SPM. Of course, the second approaches makes use of and builds on the first.

8.3.1 Example Case

Let us elaborate on the details for a system with two levels of SPM structure.

p : number of processors

q : number of SPMs

$|L|$: Total on-chip memory capacity

$|L_1|$: Level 1 capacity

$|L_2|$: Level 2 capacity

$|S|$: Total data size manipulated by the program.

We can assume the following:

$|S| \geq |L| = |L_1| + |L_2|$

$|L_1| = 2^{k_1}$

$|L_2| = 2^{k_2}$

There will be costs associated with access to SPMs by the processor, as well as costs associated with moving a data block from one SPM to another. These costs are given by Tables 8.1 and 8.2. The goal would be to move a data block b from SPM$_i$ to SPM$_j$ when it is beneficial; that is, if

C_1 = the access cost to SPM$_i$ by the processors that use b

C_2 = the access cost to SPM$_j$ by the processors that use b

C_3 = the cost of moving b from SPM$_i$ to SPM$_j$

then $C_1 > C_2 + C_3$ is our criteria for moving b from SPM$_i$ to SPM$_j$.

For execution step s_i:

Table 8.1: Access Cost Matrix

	SPM_1	SPM_2	...	SPM_q
CPU_1	$ac_{1,1}$	$ac_{1,2}$	...	$ac_{1,q}$
CPU_2	$ac_{2,1}$	$ac_{2,2}$	...	$ac_{2,q}$
CPU_3	a	a	a	a
...		...		
CPU_p	$ac_{p,1}$	$ac_{p,2}$	...	$ac_{p,q}$

Table 8.2: Move Cost Matrix

	SPM_1	SPM_2	...	SPM_q
SPM_1	0	$mc_{1,2}$	...	$mc_{1,q}$
SPM_2	$mc_{2,1}$	0	...	$mc_{2,q}$
SPM_3	$mc_{3,1}$	$mc_{3,2}$	...	$mc_{3,q}$
...		...		
SPM_q	$mc_{q,1}$	$mc_{q,2}$	...	0

If the access cost associated with CPU_j at s_i is Cost(i, j), then the total access cost for s_i is $\max_{j=1}^{p}$Cost(i,j), since the accesses are in parallel.

Therefore, the cost function that has to be minimized is $\sum_{i=1}^{\#steps} \max_{j=1}^{p}$Cost(i,j).

8.4 General Purpose Computing on Graphics Processing Units

Another avenue for continuation of our work would be exploring the applicability of some of the approaches in other scenarios, such as General Purpose Computing on Graphics Processing Units (GPGPU).

Although state of the art GPUs are not available on embedded systems, the GPU subsystem itself has some similar characteristics to an embedded system, which makes the application of the discussed data placement approaches for GPU memory access optimization as a possible next project.

Due to developments in video and gaming technologies, the demand for graphics processing power is always increasing. The GPU manufacturers' response to that is continuously packing more processing power in their GPUs to be able to keep up with this demand.

As a result, most state of the art Graphics Processing Units (GPUs) have lots of processing power. Most people don't use graphics intensive applications, or do so at a fraction of the time. As a result, there is a lot of idle processing capacity in many desktop computers in the form of GPUs. General Purpose Computing on

Graphics Processing Units (GPGPU) is the concept of using the GPU processing power for general purpose computing.

Modern GPUs have multicore architectures that have hundreds of processors and their memory architectures have multiple levels that feature SPMs and constant (read-only) memory as well as RAM. So the general ideas and approaches discussed for embedded systems are applicable for GPUs, with some restrictions.

Open Computing Language (OpenCL) standard for GPGPU programming was recently released. OpenCL is not limited to GPGPU computing, but is an open standard for parallel programming of heterogeneous systems. All major GPU manufacturers support OpenCL, and some also have proprietary GPGPU software models, which only work with their products. In spite of its increasing popularity, software support for GPGPU is still in its infancy.

In all GPGPU programming models, a basic unit of code that runs on a GPU core is called a kernel. A kernel is executed on multiple cores simultaneously. The code for all instances is identical, but each instance is uniquely identified by its index.

As a result, to run a piece of code on GPUs, it has to be translated to a format suitable for GPU execution first. This task is currently not automated, so it is an error prone, manual task for the programmer.

GPGPU memory models provide access to memory at different levels, such as core-specific, core-group-specific, system-wide and constant memory, with varying access times. Therefore, because of this multi-level memory access model, a given piece of GPGPU code can be optimized for memory access by using data placement techniques.

Bibliography

[1] Bulent Abali, Mohammad Banikazemi, Xiawei Shen, Hubertus Franke, Dan E. Poff, and T. Basil Smith. Hardware compressed main memory: Operating system support and performance evaluation. *IEEE Transactions on Computers*, 50(11):1219–1233, 2001. [cited at p. 13]

[2] M. J. Absar and F. Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1162–1167, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 61, 62, 66]

[3] Edward Ahn, Seung-Moon Yoo, and Sung-Mo Steve Kang. Effective algorithms for cache-level compression. In *Proceedings of the 11th Great Lakes symposium on VLSI*, pages 89–92, 2001. [cited at p. 13]

[4] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Chau wen Tseng. An overview of the suif compiler for scalable parallel machines. In *In Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 662–667, February 1995. [cited at p. 99, 100]

[5] Jennifer-Ann Monique Anderson. *Automatic computation and data decomposition for multiprocessors*. PhD thesis, Stanford University, Stanford, CA, USA, March 1997. Adviser-Monica S. Lam. [cited at p. 90, 99, 101]

[6] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the 5th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 166–178, 1995. [cited at p. 30]

[7] Manuel Arenaz, Juan Tourino, and Ramon Doallo. A gsa-based compiler infrastructure to extract parallelism from complex loops. In *Proc. of the 17th Annual International Conference on Supercomputing*, pages 193–204, 2003. [cited at p. 29]

[8] Vishal Aslot, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *In Workshop on OpenMP Applications and Tools*, pages 1–10, 2001. [cited at p. 99]

[9] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002. [cited at p. 66]

[10] Rajeshwari Banakar, Stefan Steinke, Bo sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *In Tenth International Symposium on Hardware/Software Codesign (CODES), Estes Park*, pages 73–78. ACM, 2002. [cited at p. 61]

[11] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of International Symposium on Computer Architecture*, 2000. [cited at p. 13]

[12] Volodymyr Beletskyy, R. Drazkowski, and Marcin Liersz. An approach to parallelizing non-uniform loops with the omega calculator. In *Proc. of the International Conference on Parallel Computing in Electrical Engineering*, 2002. [cited at p. 29]

[13] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proceedings of the conference on Design, automation and test in Europe*, page 449, 2002. [cited at p. 12]

[14] Kiran Bondalapati. Parallelizing dsp nested loops on reconfigurable architectures using data context switching. In *Proc. of the 38th Design Automation Conference*, pages 273–276, 2001. [cited at p. 29]

[15] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94,, June 2000. [cited at p. 84, 103]

[16] Mats Brorsson. Performance impact of code and data placement on the ibm rp3. Technical report, IBM, June 1989. [cited at p. 90]

[17] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *SIGPLAN Not.*, 29(11):252–262, 1994. [cited at p. 29]

[18] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *ASPLOS-VI: Proceedings of the Sixth Inter-*

national Conference on Architectural Support for Programming Languages and Operating Systems, pages 12–24, New York, NY, USA, 1994. ACM. [cited at p. 75]

[19] Weng-Long Chang, Chih-Ping Chu, and Michael Ho. Exploitation of parallelism to nested loops with dependence cycles. *Journal on System Architecture*, 50(12):729–742, 2004. [cited at p. 29]

[20] G. Chen, O. Ozturk, M. Kandemir, and M. Karakoy. Dynamic scratch-pad memory management for irregular array access patterns. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 931–936, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. [cited at p. 61, 62, 66]

[21] Guilin Chen, Ozcan Ozturk, Mahmut Kandemir, and Ibrahim Kolcu. Integrating loop and data optimizations for locality within a constraint network based framework. In *Proc. of International Conference on Computer-Aided Design*, 2005. [cited at p. 30]

[22] M. Chen and M. L. Fowler. The importance of data compression for energy efficiency in sensor networks. In *Conference on Information Sciences and Systems*, 2003. [cited at p. 13]

[23] Doosan Cho, Ilya Issenin, Nikil Dutt, Jonghee W. Yoon, and Yunheung Paek. Software controlled memory layout reorganization for irregular array access patterns. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 179–188, New York, NY, USA, 2007. ACM. [cited at p. 61, 62, 67, 68, 69]

[24] Michal Cierniak and Wei Li. Unifying data and control transformations for distributed shared-memory machines. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 205–217, 1995. [cited at p. 29]

[25] Philippe Clauss, Vincent Loechner, V. Taylor, and K. Parhi. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19(2):179–194, 1998. [cited at p. 52]

[26] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290, 1995. [cited at p. 29]

[27] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded risc processors. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 139–149, 1999. [cited at p. 12]

[28] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 684–689, New York, NY, USA, June 2001. ACM. [cited at p. 99, 100]

[29] Saumya Debray and William Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, 2002. [cited at p. 12]

[30] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal Embedded Computing*, 1(4):521–540, 2005. [cited at p. 61]

[31] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233, New York, NY, USA, 2006. ACM. [cited at p. 61]

[32] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 321–330, New York, NY, USA, 2006. ACM. [cited at p. 61]

[33] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 148–157, Washington, DC, USA, May 2002. IEEE Computer Society. [cited at p. 75, 76]

[34] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 238–243, New York, NY, USA, 2004. ACM. [cited at p. 7]

[35] Olga Golubeva, Mirko Loghi, Massimo Poncino, and Enrico Macii. Architectural leakage-aware management of partitioned scratchpad memories. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1665–1670, San Jose, CA, USA, 2007. EDA Consortium. [cited at p. 61]

[36] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of International Symposium on Computer Architecture*, 2003. [cited at p. 13]

[37] Georgios Goumas, Nikolaos Drosinos, Maria Athanasaki, and Nectarios Koziris. Automatic parallel code generation for tiled nested loops. In *Proc. of the ACM Symposium on Applied Computing*, pages 1412–1419, 2004. [cited at p. 29]

[38] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel simd architecture for the cell heterogeneous chip-multiprocessor. In *Hot Chips 17*, August 2005. [cited at p. 13]

[39] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, December 1996. [cited at p. 66, 81]

[40] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer Special Issue on "Billion-Transistor Processors"*, September 1997. [cited at p. 13]

[41] R. Hetherington. *The UltraSPARC T1 Processor - Power Efficient Throughput Computing*, December 2005. [cited at p. 13]

[42] Karin Hogstedt, Larry Carter, and Jeanne Ferrante. On the parallel execution time of tiled loops. *IEEE Transactions on Parallel Distributed Systems*, 14(3):307–321, 2003. [cited at p. 29]

[43] Bo Hu and M. Marek-Sadowska. Multilevel expansion-based vlsi placement with blockages. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 558–564, Washington, DC, USA, 2004. IEEE Computer Society. [cited at p. 89]

[44] Jingcao Hu and Radu Marculescu. Energy- and performance-aware mapping for regular noc architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):551–562, April 2005. [cited at p. 90]

[45] Christopher J. Hughes, Radek Grzeszczuk, Eftychios Sifakis, Daehyun Kim, Sanjeev Kumar, Andrew P. Selle, Jatin Chhugani, Matthew Holliman, and Yen-Kuang Chen. Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):220–231, 2007. [cited at p. 30]

[46] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, August 1992. [cited at p. 75, 79, 80]

[47] W. Hung, C. Addo-Quaye, T. Theocharides, Y. Xie, N. Vijaykrishnan, and M. J. Irwin. Thermal-aware ip virtualization and placement for networks-on-chip architecture. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 430–437, Washington, DC, USA, 2004. IEEE Computer Society. [cited at p. 89]

[48] Intel c++ compiler 10.0 for linux. [cited at p. 41, 43]

[49] Ilya Issenin, Erik Brockmeyer, Bart Durinck, and Nikil Dutt. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 49–52, New York, NY, USA, 2006. ACM. [cited at p. 59]

[50] I. Kadayif, M. Kandemir, and M. Karakoy. An energy saving strategy based on adaptive loop parallelization. In *Proc. of the 39th Design Automation Conference*, pages 195–200, 2002. [cited at p. 29]

[51] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005. [cited at p. 6]

[52] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 628–633, New York, NY, USA, 2002. ACM. [cited at p. 59]

[53] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 269–276, 1997. [cited at p. 29]

[54] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 690–695, New York, NY, USA, 2001. ACM. [cited at p. 7, 59, 66, 67, 68]

[55] Mahmut Kandemir. Lods: Locality-oriented dynamic scheduling for on-chip multiprocessors. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 125–128, New York, NY, USA, 2004. ACM. [cited at p. 46, 75, 80]

[56] Mahmut Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 219–224, New York, NY, USA, 2002. ACM. [cited at p. 7]

[57] Mahmut T. Kandemir, Ibrahim Kolcu, and Ismail Kadayif. Influence of loop optimizations on energy consumption of multi-bank memory systems. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 276–292, London, UK, April 2002. Springer-Verlag. [cited at p. 76]

[58] Mahmut T. Kandemir, Taylan Yemliha, Seung Woo Son, and Ozcan Ozturk. Memory bank aware dynamic loop scheduling. In *DATE*, pages 1671–1676, 2007. [cited at p. 9]

[59] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The omega library interface guide. Technical report, University of Maryland at College Park, College Park, MD, USA, 1995. [cited at p. 52]

[60] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 346–357, 1997. [cited at p. 29]

[61] F. Kuijlman, H. J. Sips, C. Van Reeuwijk, and W. J. A. Denissen. A unified compiler framework for work and data placement. In *In: Proceedings of the ASCI 2002 Conference*, pages 109–115, 2002. [cited at p. 90]

[62] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, 1991. [cited at p. 29]

[63] Monica S. Lam. Locality optimizations for parallel machines. In *CONPAR 94 - VAPP VI: Proceedings of the Third Joint International Conference on Vector and Parallel Processing*, pages 17–28, London, UK, 1994. Springer-Verlag. [cited at p. 46, 73]

[64] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla S. Ellis. Power-aware page allocation. In *ASPLOS-IX: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–116, New York, NY, USA, November 2000. ACM. [cited at p. 76]

[65] H. Lekatsas and Wayne Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Transactions on CAD*, 18(12):1689–1701, 1999. [cited at p. 12]

[66] S. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR-95-09-01, University of Washington, 1995. [cited at p. 29]

[67] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):358–368, 2007. [cited at p. 30]

[68] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and loop scheduling on numa multiprocessors. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, volume 02, pages 140–147, Washington, DC, USA, 1993. IEEE Computer Society. [cited at p. 46]

[69] Lian Li, Lin Gao, and Jingling Xue. Memory coloring: A compiler approach for scratchpad memory management. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 329–338, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 59, 61]

[70] Wei Li. *Compiling for numa parallel machines*. PhD thesis, Cornell University, Ithaca, NY, USA, 1993. [cited at p. 29]

[71] Wei Li and Keshav Pingali. Access normalization: loop restructuring for numa compilers. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 285–295, New York, NY, USA, 1992. ACM. [cited at p. 90]

[72] Stan Liao, S. Devadas, and K Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Proc. Conf. on Advanced Research in VLSI*, 1995. [cited at p. 12]

[73] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proc. of the 13th International Conference on Supercomputing*, pages 228–237, 1999. [cited at p. 29]

[74] Chang Hong Lin, Yuan Xie, and Wayne Wolf. Lzw-based code compression for vliw embedded systems. In *Proceedings of the conference on Design, automation and test in Europe*, page 30076, 2004. [cited at p. 12]

[75] David K. Lowenthal and Gregory R. Andrews. An adaptive approach to data placement. *International Parallel Processing Symposium*, 0:349, 1996. [cited at p. 90]

[76] lp_solve. [cited at p. 39]

[77] Alberto Macii, Enrico Macii, Fabrizio Crudo, and Roberto Zafalon. A new algorithm for energy-driven data compression in vliw embedded processors. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10024, Washington, DC, USA, 2003. IEEE Computer Society. [cited at p. 13]

[78] Cesar Marcon, Ney Calazans, Fernando Moraes, Altamiro Susin, Igor Reis, and Fabiano Hessel. Exploring noc mapping strategies: An energy and timing aware technique. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 502–507, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 89]

[79] Evangelos P. Markatos and Thomas J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. In *International Conference on Parallel Processing*, pages 258–267, August 1992. [cited at p. 46, 54, 56]

[80] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, 1994. [cited at p. 75, 81]

[81] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 10296–10301, 2003. [cited at p. 29]

[82] Wen mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 754–759, 2007. [cited at p. 30]

[83] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. Mc. Graw Hill, 1994. [cited at p. 33]

[84] Sun Microsystems. *MAJC-5200*. [cited at p. 13]

[85] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 62–73, New York, NY, USA, 1992. ACM. [cited at p. 65]

[86] *MP98: A Mobile Processor*. [cited at p. 13]

[87] Angeles Navarro, Emilio Zapata, and David Padua. Compiler techniques for the distribution of data and computation. *IEEE Transactions on Parallel Distributed Systems*, 14(6):545–562, 2003. [cited at p. 29]

[88] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proceedings of International Symposium on Computer Architecture*, 1994. [cited at p. 13]

[89] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 115–125, New York, NY, USA, 2005. ACM. [cited at p. 61]

[90] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Scratch-pad memory allocation without compiler support for java applications. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 85–94, New York, NY, USA, 2007. ACM. [cited at p. 59, 61]

[91] M. O'Boyle. A hierarchical locality algorithm for numa compilation. In *PDP '95: Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, page 106, Washington, DC, USA, 1995. IEEE Computer Society. [cited at p. 90]

[92] Michael F. P. O'Boyle and Peter M. W. Knijnenburg. Nonsingular data transformations: Definition, validity, and applications. *Int. J. Parallel Program.*, 27-3:131–159, 1999. [cited at p. 29]

[93] K. Olukotun and L. Hammond. The future of microprocessors. *ACM QUEUE Magazine*, September 2005. [cited at p. 13]

[94] The omega project: Frameworks and algorithms for the analysis and transformation of scientific programs. [cited at p. 17]

[95] Openmp: Simple, portable, scalable smp programming. [cited at p. 41]

[96] O. Ozturk, M. Kandemir, I. Demirkiran, G. Chen, and M. J. Irwin. Data compression for improving spm behavior. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 401–406, New York, NY, USA, 2004. ACM Press. [cited at p. 13]

[97] Ozcan Ozturk, Guilin Chen, and Mahmut Kandemir. A constraint network based solution to code parallelization. In *Design Automation Conference (DAC'06)*, July 2006. [cited at p. 29]

[98] Ozcan Ozturk and Mahmut Kandemir. Integer linear programming based energy optimization for banked drams. In *GLSVSLI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pages 92–95, New York, NY, USA, 2005. ACM Press. [cited at p. 13]

[99] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 7, Washington, DC, USA, 1997. IEEE Computer Society. [cited at p. 61]

[100] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987. [cited at p. 46, 75, 79, 80]

[101] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Norwell, MA, USA, 1988. [cited at p. 73, 75]

[102] *POWER4 System Microarchitecture*. [cited at p. 13]

[103] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium. [cited at p. 61]

[104] Laura Ricci. Automatic loop parallelization: an abstract interpretation approach. In *Proc. of the International Conference on Parallel Computing in Electrical Engineering*, pages 112–118, 2002. [cited at p. 29]

[105] S. Richardson. Mpoc: A chip multiprocessor for embedded systems. Technical report, HP Labs, 2002. [cited at p. 13]

[106] Montserrat Ros and Peter Sutton. Code compression based on operand-factorization for vliw processors. In *Proceedings of the Conference on Data Compression*, page 559, 2004. [cited at p. 12]

[107] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, HP, 2001. [cited at p. 84, 102]

[108] Virtutech simics simulator. [cited at p. 41, 54, 80, 81, 99]

[109] Vassos Soteriou and Li-Shiuan Peh. Design-space exploration of power-aware on/off interconnection networks. In *Proceedings of the 22nd International Conference on Computer Design (ICCD)*, October 2004. [cited at p. 102]

[110] Sudarsan Tandri and Tarek S. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. In *ICPP '97: Proceedings of the international Conference on Parallel Processing*, pages 64–73, Washington, DC, USA, August 1997. IEEE Computer Society. [cited at p. 90]

[111] Peiyi Tang and Pen-Chung Yew. Processor self-scheduling for multiple-nested parallel loops. In *International Conference on Parallel Processing*, pages 528–535, August 1986. [cited at p. 46, 55]

[112] B. Tunstall. *Synthesis of Noiseless Compression Codes*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1967. [cited at p. 12]

[113] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993. [cited at p. 75, 80]

[114] Manish Verma, Stefan Steinke, and Peter Marwedel. Data partitioning for maximal scratchpad usage. In *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, pages 77–83, New York, NY, USA, 2003. ACM. [cited at p. 59]

[115] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: A power-performance simulator for interconnection networks. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. [cited at p. 99]

[116] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 30–44, 1991. [cited at p. 29]

[117] W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of Design Automation Conference*, 2004. [cited at p. 13]

[118] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. [cited at p. 73, 80]

[119] Yuan Xie, Wayne Wolf, and Haris Lekatsas. Code compression for vliw processors using variable-to-fixed coding. In *Proceedings of the 15th international symposium on System Synthesis*, pages 138–143, 2002. [cited at p. 12]

[120] Yuan Xie, Wayne Wolf, and Haris Lekatsas. Profile-driven selective code compression. In *Proceedings of the conference on Design, Automation and Test in Europe*, page 10462, 2003. [cited at p. 12]

[121] Rong Xu, Zhiyuan Li, Cheng Wang, and Peifeng Ni. Impact of data compression on energy consumption of wireless-networked handheld devices. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 302, Washington, DC, USA, 2003. IEEE Computer Society. [cited at p. 13]

[122] Liping Xue, Mahmut T. Kandemir, Guangyu Chen, and Taylan Yemliha. Spm conscious loop scheduling for embedded chip multiprocessors. In *ICPADS (1)*, pages 391–400, 2006. [cited at p. 9]

[123] Liping Xue, Ozcan Ozturk, and Mahmut Kandemir. A memory-conscious code parallelization scheme. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 230–233, 2007. [cited at p. 30]

[124] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 258–265, 2000. [cited at p. 13]

[125] Peng Yang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rudy Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor socs. *IEEE Design and Test of Computers*, 18(5):46–58, 2001. [cited at p. 46, 75]

[126] Taylan Yemliha, Guangyu Chen, Ozcan Ozturk, Mahmut T. Kandemir, and Vijay Degalahal. Compiler-directed code restructuring for operating with compressed arrays. In *VLSI Design*, pages 221–226, 2007. [cited at p. 8]

[127] Taylan Yemliha, Mahmut T. Kandemir, Ozcan Ozturk, Emre Kultursay, and Sai Prashanth Muralidhara. Code scheduling for optimizing parallelism and data locality. In *Euro-Par (1)*, pages 204–216, 2010. [cited at p. 8]

[128] Taylan Yemliha, Shekhar Srikantaiah, Mahmut T. Kandemir, Mustafa Karaköy, and Mary Jane Irwin. Integrated code and data placement in two-dimensional mesh based chip multiprocessors. In *ICCAD*, pages 583–588, 2008. [cited at p. 9]

[129] Taylan Yemliha, Shekhar Srikantaiah, Mahmut T. Kandemir, and Ozcan Ozturk. Spm management using markov chain based data access prediction. In *ICCAD*, pages 565–569, 2008. [cited at p. 9]

[130] Yijun Yu and Erik H. D'Hollander. Loop parallelization using the 3d iteration space visualizer. *Journal of Visual Languages and Computing*, 12(2):163–181, 2001. [cited at p. 29]

VITA

NAME OF AUTHOR: Taylan Yemliha

PLACE OF BIRTH: Istanbul, Turkey

DATE OF BIRTH: May 8, 1971

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:
Bogazici University, Istanbul, Turkey
Marmara University, Istanbul, Turkey

DEGREES AWARDED:

Master of Science in Computer Engineering, 1996, Marmara University

Bachelor of Science in Computer Engineering, 1993, Bogazici University

PROFESSIONAL EXPERIENCE:
BYTE magazine (Turkish Edition), Executive editor, 1993-1996
IhlasNet (ISP), IT Manager, 1997-2001
Research Assistant, EECS, Syracuse University, 2001-2003
Teaching Assistant, EECS, Syracuse University, 2003-2009
Graduate Assistant, EECS, Syracuse University, 2009-2011