The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

COMPILER DIRECTED MEMORY HIERARCHY DESIGN AND MANAGEMENT IN CHIP MULTIPROCESSORS

A Thesis in

Computer Science and Engineering

by

Ozcan Ozturk

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2007

The thesis of Ozcan Ozturk was read and approved* by the following:

Mahmut Kandemir
Associate Professor of Computer Science and Engineering
Thesis Adviser
Chair of Committee

Mary Jane Irwin
Professor of Computer Science and Engineering

Yuan Xie
Assistant Professor of Computer Science and Engineering

Aylin Yener
Associate Professor of Electrical Engineering

Vijaykrishnan Narayanan
Associate Professor of Computer Science and Engineering
Director of Graduate Affairs for the
Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Two trends, namely, increasing importance of memory subsystems and increasing use of chip multiprocessing, motivate conducting research on memory hierarchy optimization for chip multiprocessors. One of the critical research topics along this direction is to design an application-specific, customized, software-managed on-chip memory hierarchy for a chip multiprocessor. Another important issue is to optimize the application code and data for such a customized on-chip memory hierarchy. This thesis proposes solutions to the problem of memory hierarchy design and data access management. First, an integer linear programming (ILP) based solution to the combined problem of memory hierarchy design and data allocation in the context of embedded chip multiprocessors is given. The proposed solution uses compiler analysis to extract data access patterns of parallel processors and employs integer linear programming for determining the optimal on-chip memory partitioning across processors and data allocations across memory components. Second, we present and evaluate a compiler-driven approach to data compression for reducing memory space occupancy. Our goal is to study how automated compiler support can help in deciding the set of data elements to compress/decompress and the points during execution at which these compressions/decompressions should be performed. The proposed compiler support achieves this by analyzing the source code of the application to be optimized and identifying the order in which the different data blocks are accessed. Based on this analysis, the compiler then inserts compression/decompression calls in the application code. The compression calls target the data blocks that are not expected to

be used in the near future, whereas the decompression calls target those data blocks with expected reuse but currently in compressed form. Third, we present a constraint network based formulation of the integrated loop-data optimization problem to improve locality of data accesses for a given application code. We present two alternate solutions to the data locality problem with our constraint network based formulation and discuss the pros and cons of each scheme. The first solution is a pure backtracking based one, whereas the second one improves upon the first one by employing three additional optimizations, including backjumping. Fourth, we extend our constraint network based approach to code parallelization for chip multiprocessors. Fifth, we explore how processor cores and storage blocks can be placed in a 3D architecture to minimize data access costs under temperature. This process (topology exploration) has been implemented using integer linear programming. Finally, we present a Scratch Pad Memory space management technique for chip multiprocessors. We focus on the management of a scratch pad space shared by multiple applications executing at the same time that can potentially share data. The proposed approach has three major components; a compiler analysis phase, a runtime space partitioner, and a local partitioning phase. This thesis also presents experimental evidence, demonstrating the success of each of the proposed techniques, and compares our results with those obtained through previously-proposed approaches.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

One of the critical components of an embedded computing system is its *memory organization*. There are several reasons for this. First, many embedded applications are data intensive and make frequent memory references. As a result, a significant fraction of total execution cycles is spent in memory accesses. Second, this large number of accesses also contribute to a large fraction of overall power consumption [29]. Third, since on-chip memory structures constitute a significant portion of overall chip area in embedded designs, they are highly vulnerable to transient errors, making them an important target for reliability optimizations. Finally, many security leaks are exploited through manipulation of memory space. Based on these observations, there have been numerous approaches proposed in the last two decades for optimizing memory behavior of embedded systems and applications, considering different metrics of interest such as performance, power, and reliability. Note that, as embedded applications become more complex and process increasingly larger data sets, the role of the memory system will be even more important in the future.

Continuously-scaling process technology allows millions of transistors to be packed onto a single die. In practice, this means a single chip represents an increasingly powerful computing platform. One promising instantiation of this powerful platform is a *chip multi-processor* where multiple processors are connected via an on-chip communication fabric

to each other and to an on-chip memory space. Past research [5, 82] has discussed the potential benefits of chip multiprocessors over complex single processor based architectures. There are at least three trends that motivate for chip multiprocessing in embedded computing domain. First, computer architects are finding it increasingly difficult to wring more performance out of single-processor based approaches. Second, chip multiprocessor architectures promise to simplify the increasingly complex design, hardware verification, and validation work of processors. This in turn lowers chip costs and reduces time-to-market. Third, many embedded applications from image/video processing domain lend themselves very well to parallel computing. Therefore, they can take advantage of parallel processing capabilities provided by chip multiprocessors.

These two trends, growing importance of memories and increasing use for chip multiprocessing, form a strong motivation for conducting research on memory synthesis and optimization for chip multiprocessors as well as data management related issues. One of the interesting research topics along this direction is to design an application-specific, software-managed, on-chip memory hierarchy. Another important issue is to optimize the application code and data for such a customized on-chip memory hierarchy.

Designing such a customized memory hierarchy is not trivial because of at least three main reasons. First, since the memory architecture to be designed changes from one application to another, a hand-waived approach is not suitable, as it can be extremely time consuming and error-prone to go through the same complex process each time we are to design a memory hierarchy for a new application. Therefore, we need an automated strategy that comes up with the most suitable design for a given application. Second, design of such a memory needs to be guided by a tool that can extract the data sharing

exhibited by the application at hand. After all, in order to decide how the different memory components need to be shared by parallel processors, one needs to capture the data sharing pattern across the processors. Third, data allocation in a memory hierarchy is not a trivial problem, and should be carried out along with data partitioning if we are to obtain the best results.

In this work, we propose a strategy for designing application-specific memory hierarchy for chip multiprocessors that employs both an optimizing compiler and an ILP (integer linear programming) solver. The role of the compiler in this approach is to analyze the application code and detect the data sharing patterns across processors, given the loop parallelization information. The job of the ILP solver, on the other hand, is to determine the sizes of the memory components in the memory hierarchy, how these memory components are shared across multiple processors in the system, and what data each memory component is to hold. Note that, the ILP based solution can be used not only for designing memory hierarchies, but also as an upper bound against which current and future heuristic solutions can be compared.

In addition to memory hierarchy exploration, we also propose data compression to reduce memory space consumption (occupancy) of embedded applications. The goal of data compression is to represent an information source (e.g., a data file, a speech signal, an image, or a video signal) as accurately as possible using the fewest number of bits. Previous research considered efficient hardware and software based data compression techniques and applied compression to different domains. While data compression

can be an effective way of reducing memory space consumption of embedded applications, it needs to be exercised with care since performance and power costs of decompression (when we need to access data stored currently in the compressed form) can be overwhelming. Therefore, compression/decompression decisions must be made based on a careful analysis of data access pattern of the application. Our goal in this part of our research is to study how automated compiler support can help in deciding the set of data elements to compress/decompress and the points during execution at which these compressions/decompressions should be performed. The proposed compiler support achieves this by analyzing the source code of the application to be optimized and identifying the order in which the different data blocks are accessed. Based on this analysis, the compiler then inserts compression/decompression calls in the application code. The compression calls target the data blocks that are not expected to be used in the near future, whereas the decompression calls target those data blocks with expected reuse but currently in compressed form.

The main data locality problem in the memory hierarchy context is to maximize the number of the data accesses satisfied from the higher levels in the memory hierarchy (i.e., those that are close to the CPU). There have been two different trends in optimizing memory behavior. One of these is called code restructuring and usually takes form of loop optimizations. These optimizations process a single loop nest at a time (or sometimes a series of consecutive loop nests as in the case of loop fusioning) and try to change data access pattern of the loop (by re-organizing loop iterations) so that it becomes suitable for the memory layout of the data structures manipulated by the loop. The second trend in optimizing data-intensive applications for locality is to employ data transformations. A

data transformation typically changes the memory layout of data in an attempt to make it suitable for a given loop structure (data access pattern). We try to explore the possibility of employing *constraint network theory* for solving the integrated data-loop optimization problem to improve the data locality of a given application code. Constraint networks have proven to be a useful mechanism for modeling and solving computationally intensive tasks in artificial intelligence. They operate by expressing a problem as a set of variables, variable domains and constraints and define a search procedure that tries to satisfy the constraints (an acceptable subset of them) by assigning values to variables from their specified domains. We present two alternate solutions to the data locality problem with our CN based formulation and discuss the pros and cons of each scheme. The first solution is a pure backtracking based one, whereas the second one improves upon the first one by employing three additional optimizations, including backjumping, which targets at eliminating unnecessary backtracking decisions.

Increasing employment of chip multiprocessors in embedded computing platforms requires a fresh look at conventional code parallelization schemes. In particular, any compiler-based parallelization scheme for chip multiprocessors should account for the fact that interprocessor communication is cheaper than off-chip memory accesses in these architectures. A compiler-based code parallelization targeting these architectures therefore needs to focus on cutting the number of off-chip requests, even if this also entails an increase in on-chip activities. Unfortunately, most of the code parallelization techniques proposed and studied for high-performance parallel machines do not extend directly to chip multiprocessors due to two major factors. First, most of these code parallelizers handle one loop nest at a time and thus fail to capture the data sharing patterns across different

loop nests which are important for minimizing off-chip accesses, as will be discussed later. Second, in parallelizing a loop nest, most of these previous compiler algorithms focus on minimizing interprocessor communication rather than optimizing memory footprint. Based on this observation, we extend our constraint network based approach discussed above to code parallelization for chip multiprocessors. Particularly, we demonstrate that it is possible to use a constraint network based formulation for the problem of code parallelization for chip multiprocessors.

As technology scales, increasing interconnect costs make it necessary to consider alternate ways of building integrated circuits. One promising option along this direction is 3D architectures where a stack of multiple device layers, with direct vertical tunneling through them, are put together on the same chip. In this context, we explore how processor cores and storage blocks can be placed in a 3D architecture to minimize data access costs under temperature constraints. This process is referred to as the topology exploration. Using integer linear programming, we compare the best 2D placement with the best 3D placement, and show through experiments with both single-core and chip multiprocessor systems that the 3D placement generates much better results (in terms of data access costs) under the same temperature bounds. We also discuss the tradeoffs between temperature constraint and data access costs.

Scratch-Pad Memories (SPMs) are important storage components in many embedded applications and used as an alternative or a complimentary storage to on-chip cache memories. One of the most critical issues in the context of SPMs is to select the data elements to place in them since the gap between SPM access latencies and off-chip memory access latencies keep increasing dramatically. Previous research considered this problem

and attacked it using both static and dynamic schemes. Most of the prior efforts on data SPMs have mainly focused on single application scenarios, i.e., the SPM space available is assumed to be managed by a single application at any given time. While this assumption makes sense in certain domains, there also exist many cases where multiple applications need to share the same SPM space. We focus on such a multi-application scenario and propose a nonuniform SPM space partitioning and management across concurrently-executing applications. In our approach, the amount of data to be allocated to each application is decided based on the data reuse each application exhibits. The results from our experiments show that nonuniform partitioning generates much better runtime behavior than uniform partitioning for all the execution scenarios evaluated, and the magnitude of our savings increases with smaller SPM spaces and larger number of applications.

The rest of this thesis is organized as follows. A discussion of the related work is given in Chapter 2. Chapter 3 discusses our custom multi-level memory hierarchy design approach. In Chapter 4, we present our compression techniques. The details of our constraint network based approach to solve the integrated data layout-loop optimization problem are discussed in Chapter 5. Chapter 6 gives our code parallelization approach using constraint networks. Chapter 7 presents the details of our ILP-based optimal topology exploration technique for 3D architectures, and Chapter 8 discusses the shared scratch pad memory (SPM) management scheme in detail. Finally, the conclusions and a brief summary of planned future work are given in Chapter 9.

Chapter 2

Discussion of Related Work

An important advantage of chip multiprocessors is that it reduces the cost of interprocessor data communication from both performance and power perspectives as this communication does not need to go over off-chip buses. The prior work [4, 5, 31, 68, 82, 100, 106, 121, 145, 86, 164, 146, 124, 165, 101, 111, 80] discusses several advantages of these architectures over complex single-processor based designs.

Memory system design and optimization has been a popular area of research for the last two decades. We can roughly divide the work in this category into two classes: memory synthesis/design and software optimizations. Most of the prior work on memory synthesis/design focus on single processor based systems [11, 26, 29, 131, 136, 141, 144, 151]. In [123, 127, 112], we present memory hierarchy management schemes from both energy and performance angles on a software-managed, application specific multi-level memory hierarchy. In comparison, Abraham and Mahlke focus on an embedded system consisting of a VLIW processor, instruction cache, data cache, and second-level unified cache. A hierarchical approach to partitioning the system into its constituent components and evaluating each component individually is utilized. Meftali et al [98] focus on the memory space allocation (partitioning) problem, which they formulate based on an integer linear programming model. Their solution permits one to obtain an optimal distributed shared memory architecture, minimizing the global cost to access the

shared data in the application and the memory cost. The effectiveness of the proposed approach is demonstrated by a packet routing switch example. Gharsalli et al [57] present a new methodology for embedded memory design for application-specific multiprocessor system-on-chips. Their approach facilitates the integration of standard memory components. Further, the concept of memory wrapper they introduce allows automatic adaptation of physical memory interfaces to a communication network that may have a different number of access ports. Li and Wolf [90] introduced a hardware/software co-synthesis algorithm of distributed real-time systems that optimize the memory hierarchy along with the rest of the architecture. Dasygenis et al [49] propose a formalized technique that uses data reuse, lifetime of the arrays of an application and application specific prefetching opportunities. Using these parameters, authors perform a trade-off exploration for different memory layer sizes. Issenin et al [66], on the other hand introduces a multiprocessor data reuse analysis technique to explore a wide range of customized memory hierarchy organizations with different sizes. In [62], authors implement an incremental technique for hierarchical memory size requirement estimation. The main difference between our work and these prior efforts is that our work is very general. Although we focus on application-specific memory design, the proposed solution strategy (based on ILP) is applicable to any array-based embedded application. In addition, we also address the data allocation problem along with the memory space partitioning problem. [74, 116] proposes a dynamic memory management scheme for chip multiprocessors. Our work is also more general than the one in [74, 116], as the latter does not design a hierarchy but only partitions the available memory space across multiple processors. Our previous work also includes different techniques for banked memory structures [77, 110, 115, 79].

On the software side, the prior work considered the optimal use of the available SPM space. Panda et al [132] present an elegant static data partitioning scheme for efficient utilization of scratch-pad memory. Their approach is oriented towards eliminating the potential conflict misses due to limited associativity of on-chip cache. This approach benefits applications with a number of small (and highly reused) arrays that can fit in the SPM. Benini et al [20] discuss an effective memory management scheme that is based on keeping the most frequently used data items in a software-managed memory (instead of a conventional cache). Kandemir et al [76] propose a dynamic SPM management scheme for data accesses. Their framework uses both loop and data transformations to maximize the reuse of data elements stored in the SPM. They enhance their approach in [70]. Cooper and Harvey [45] present two compiler-directed methods for software-managing a small cache for holding spilled register values. Hallnor and Reinhardt [60] propose a new software-managed cache architecture and a new data replacement algorithm. Angiolini et al [11] propose an algorithm to optimally solve the memory allocation problem for scratch-pad memories. They use dynamic programming to solve the problem on a synthesizable hardware architecture. Segments of external memory are mapped to physically partitioned banks of an on-chip SPM. In [12], instead of physically partitioning an SPM and mapping code and data chunks on SPM banks, authors allocate code sections onto a monolithic SPM by using software-only tools. In [15], authors present a compiler strategy that automatically partitions the data among the memory units. They also show that this strategy is optimal among all possible static partitions. Authors, present a dynamic allocation method for global and stack data in [151]. An integrated hardware/software solution

to support scratch-pad memories is presented in [54] at a high abstraction level. A technique for efficiently exploiting on-chip scratch-pad memory with the goal of minimizing the total execution time has been proposed in [133]. They try to partition the application's scalar and array variables between the off-chip DRAM and the on-chip scratch-pad SRAM. Our previous work include [125, 128, 35, 126, 33] different SPM management schemes and energy saving techniques. The main difference between these studies and our work is that, in addition to data allocation, we also focus on customized memory hierarchy design.

Compression techniques have been used for both program code and application data to reduce the memory footprint and the energy consumption. RISC processors have been the main focus for code compression techniques. In our previous research, we have used both integer linear programming (ILP) [113, 114, 117, 119, 122] and heuristic approaches [71, 73, 107, 117, 118, 120, 130, 166] to reduce the memory occupancy and the energy consumption. Among these approaches, except [130], we apply data compression.

To reduce the size of a program's code segment, [46] uses pattern-matching techniques that identify and coalesce together repeated instruction sequences. A RISC system that can directly execute compressed programs is presented in [158]. VLIW (Very Long Instruction Word) processors are now being considered in this area as well. Ros and Sutton [139] present code compression algorithms applied to instruction words. Profile-driven methods that selectively compress code segments in VLIW architectures have also been proposed [50, 162]. Code compression schemes for VLIWs that use Variable-to-fixed (V2F) coding was investigated by [150, 161]. Variable-sized-block method for VLIW code compression has been introduced by Lin et al [95]. Data compression has

been used to reduce storage requirements, bus bandwidth, and energy consumption in the past [7, 19, 21]. In [19], a hardware-assisted data compression for on-the-fly data compression and decompression has been proposed. In their work, compression and decompression takes place on the cache-to-memory path. Lee et al [84] use compression in an effort to explore the potential for an on-chip cache compression to reduce cache miss ratio and miss penalty. Abali et al [7] investigate the performance impact of hardware compression. Compression algorithms that would be suitable to use in a compressed cache have been presented in [8]. Apart from memory subsystems, data compression has also been used to reduce the communication volume. For example, data compression has been proposed to reduce the communication latency and energy consumption in sensor networks [37]. The impact of data compression on a wireless LAN has been investigated in [163]. Our work is different from all these prior efforts since we give the task of management of the compressed data blocks to the compiler. In deciding the data blocks to compress and decompress, our compiler approach makes use of the data reuse information extracted from the array accesses in the application source code.

A potential drawback of chip multiprocessors is the increased cost of off-chip accesses as compared to interprocessor communication. In particular, multiple processors can try to use the same set of buses/pins to go off-chip, and this can put a high pressure on memory subsystem. Consequently, cutting the number of off-chip memory accesses becomes extremely important. Prior research exploit data locality as a potential solution. Most of the prior proposals to the problem are based on loop transformations [83, 88, 89, 159], i.e., modifying the order of loop iterations to make data access pattern more cache friendly. Data reuse and data lifetime information has been exploited in different

directions that include both code space and data space optimizations [76, 133, 155]. Wolf and Lam [157] define reuse vectors and reuse spaces and show how these concepts can be exploited by an iteration space optimization technique. Li [88] also uses reuse vectors to detect the dimensions of loop nest that carry reuse. Carr et al [27] use a simple locality criterion to re-order the computation to enhance data locality. The loop based locality enhancing techniques also include tiling [42, 81, 83]. On the data transformation side, O'Boyle and Knijnenburg [105] explain how to generate code given a data transformation matrix. Leung and Zahorjan [85] focus more on minimizing memory consumption after a layout transformation. Kandemir et al [75] propose a loop optimization technique based on explicit layout representation.

Cierniak and Li [39] were among the first to offer a scheme that unifies loop and data transformations. Anderson et al [10] propose a transformation technique that makes data elements accessed by the same processor contiguous in the shared address space. They use permutations (of array dimensions) and strip-mining for possible data transformations. O'Boyle and Knijnenburg [104] discuss a technique that propagates memory layouts across nested loops. This technique can potentially be used for inter-procedural optimization as well. Our approach is different from all these studies as we employ a constraint network based solution. A constraint network approach is used in [32], which focuses exclusively on data optimizations. In contrast, we address the combined data/loop optimization problem in [36], which is a more general framework that subsumes the previous work. We have also implemented different locality optimization techniques including an integer linear programming (ILP) based approach [72, 129, 109, 56, 34].

Automatic code parallelization has not received a significant attention in the context of chip multiprocessors. Bondalapati propose techniques for parallelizing nested loops that appear in the digital signal processing domain [24]. To parallelize such loops, they exploit the distributed memory available in the reconfigurable architecture by implementing a data context switching technique. Virtex and the Chameleon Systems have been used to implement the proposed methods. In [69], Kadayif et al exploit the use of different number of processor cores for each loop nest to obtain energy savings. In the proposed approach, idle processors are switched to a low-power mode to increase the energy savings. To reduce the performance penalty, a pre-activation strategy is also discussed. Goumas et al [59] propose a framework to automatically generate parallel code for tiled nested loops. They have implemented several loop transformations within the proposed approach using MPI, a message-passing parallel interface. A modulo scheduling algorithm to exploit loop-level parallelism for coarse-grained reconfigurable architectures has been proposed by Mei et al [99]. Hogstedt et al [61] investigate the parallel execution time of tiled loop nests. They use a prediction formula for the execution time of tiled loop nests to aid the compiler. Using this prediction, compiler is able to automatically determine the tiling parameters that minimizes the execution time. Navarro et al [102] target minimizing communication and load imbalance in parallel programs. They 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. Parallelism in a nested loop is exploited in [30] by analyzing data dependencies. A variable renaming technique is used to break anti and output dependencies along with a technique to resolve recurrences in a nested loop. Arenaz et al [13] propose a gated single assignment (GSA)

based approach to exploit coarse-grain parallelism in loop nests with complex computations. They analyze the use-def chains between the statements of a strongly-connected component, as well as between the statements of the different strongly-connected components. Yu and D'Hollander [168] present a 3D iteration space visualizer to analyze the parallelism in nested loops. An iteration space dependency graph is constructed to show the data dependencies and to indicate the maximal parallelism of nested loops. Beletskyy et al [18] propose an approach to parallelize loops with nonuniform dependencies. By adopting a hyperplane based representation, they present how transformation matrices can be found and applied to loops with both uniform and non-uniform dependencies. Ricci [138] proposes an abstract interpretation to parallelize loop nests. Using this abstract interpretation, the author aims to reduce the complexity of the analysis needed for loop parallelization. Lim et al [93] use an affine partitioning framework which unifies different transformations to maximize parallelism while minimizing communication. In the proposed approach, they find the optimal affine partition that minimize communication and synchronization.

There are also recent efforts on feedback directed, iterative and adaptive compilation schemes [23, 40, 41, 44, 55, 78]. While such techniques also search for the best solution over a search space (of possible transformations and their parameters such as tile sizes and unrolling factors), they typically execute the application to be optimized multiple times and, based on the results returned (from these executions), tune the search strategy further to obtain a better version. In comparison, our approach [108] finds the solution in one step. We also believe that the iterative and adaptive solution techniques

proposed in literature can benefit from our approach in reducing the search space, and we plan to investigate this issue in our future research.

Our CN based parallelization strategy [108] is different from these prior efforts in that it is oriented towards minimizing the number of off-chip memory references. Also, as against most of the prior work, it considers multiple loop nests at the same time taking inter-nest effects into consideration.

Design techniques and methodologies for 3D architectures have been investigated to efficiently exploit the benefits of 3D technologies. Recent efforts have focused on developing tools for supporting custom 3D layouts and placement tools [48]. However, the investigation of the benefits of 3D design at the architectural level is still in its infancy. In Deng et al [53], the technology and testing issues are surveyed and a 3D integration framework is presented. Xie et al [160] discuss 3D integration technology, EDA design tools that enable 3D ICs, and implementation of various microprocessor components using 3D technology. Hua et al [63] analyze the trade-off between benefits of 3D integration and thermal issues related to 3D ICs. Lim [94] investigates the problem of physical design for 3D ICs. Beattie et al [17] focus on the spatially distributed 3D circuit models. Das et al [47] study the energy and thermal behavior of 3D designs under a supplied time constraint, and their tool is based on a standard-cell circuit layout. A recent paper provides an overview of the potential benefits of designing an Itanium processor in the 3D technology [22]. However, it does not provide details of the design of the individual components. Several recent efforts also study employment of 3D designs in reconfigurable fabrics [6, 9]. Vaidyanathan et al [152] implement a custom microprocessor design to show the potential performance and power benefits achievable through 3D integration

under thermal constraints. Puttaswamy and Loh [135] evaluate the thermal impact of building high-performance microprocessors in 3D. Li et al [87] study the challenges for L2 cache design and management in 3D chip multiprocessors. 3D Cacti [148] implement a delay and energy model to explore the architectural design of cache memories using 3D circuits. In [134], authors focus on the design of on-chip caches using 3D integration. Zhang et al [169] propose a temperature-aware 3D global routing algorithm by using thermal vias and thermal wires to lower the effective thermal resistance. Goplen and Sapatnekar [58] explore the thermal via placement methods using finite element analysis (FEA). Yu et al [167] develop a thermal-via allocation technique considering temporally and spatially variant thermal-power. Authors define a nonlinear optimization problem to allocate thermal-vias and minimize thermal violation integral and solve this optimization problem by using quadratic programming. Hung et al [64] present a thermal-aware floorplanning approach for 3D architectures. In [92, 91], authors propose a 3D floorplanning and thermal via planning algorithm for thermal optimization in two-stacked die integration. Cong and Zhang [43] propose a multi-level routing approach using an adaptive lumped resistive thermal model and a two-step multi-level TS-via planning. Cheng et al [38] present a simulated annealing based floorplanning algorithm for 3-D ICs.

Chapter 3

Multi-Level On-Chip Memory Hierarchy Design

As embedded applications (in particular, those in multi- media domain) are becoming more and more data intensive, managing data transfers across memory hierarchy, built from components of varying sizes, costs, access latencies and energy consumptions, is becoming increasingly important. While memory hierarchy managements has been well studied in the context of caches from the performance perspectives, software managed memories and optimizing energy behavior have received relatively less attention. This is unfortunate because software-managed hierarchies can be preferable against their hardware-managed counterparts in at least two aspects. First, they are usually the best choice for embedded systems that execute a single application. In such cases, one can design a customized memory hierarchy that suits the needs of the application in question. Second, since locating data in a software-managed hierarchy is carried out by software, it is more energy-efficient as compared to the dynamic lookup based technique employed by hardware-managed cache hierarchies.

Focusing on such a software-managed memory hierarchy, the proposed solution targets array-based embedded applications from the domain of image/video processing. It is based on integer linear programming (ILP) and thus determines the optimal on-chip memory hierarchy for a given application, under the assumed cost model. It needs to be noted that the optimality of a memory system can be defined based on the goal of the

optimization. Our goal is to design a software-managed on-chip memory hierarchy for chip multiprocessors and allocate data across the components of this hierarchy in such a manner that the energy consumption on the memory system (during execution of the application) is minimized. The proposed solution is very general and can come up with any on-chip memory hierarchy. In the most general case, the resulting memory system will have both private components (i.e., memories that are exclusively accessed by a single processor) and shared components (i.e., memories that are shared by multiple processors). Also, in the most general case, each processor can see a different memory hierarchy (in terms of both the number of the components in the hierarchy and their sizes). Figure 3.1 illustrates four different example on-chip memory designs. (a) and (b) correspond to conventional pure shared memory based and pure private memory based designs, respectively. (d) is a typical hierarchy design returned by our ILP-based approach proposed in this section. (c) is a design that would be returned by an approach that partitions the on-chip memory space across a single level (not a hierarchy).

We implemented the proposed approach using a commercial linear solver and performed experiments using ten embedded benchmark codes. The experimental results show that the proposed ILP-based approach (1) comes up with on-chip memory hierarchies that are much better (in terms of energy efficiency) than conventional memory designs based on pure shared memory or pure private memories, and (2) performs better than customized memory architectures that perform optimal memory space partitioning across processors without designing a hierarchy. The results also indicate that, for the

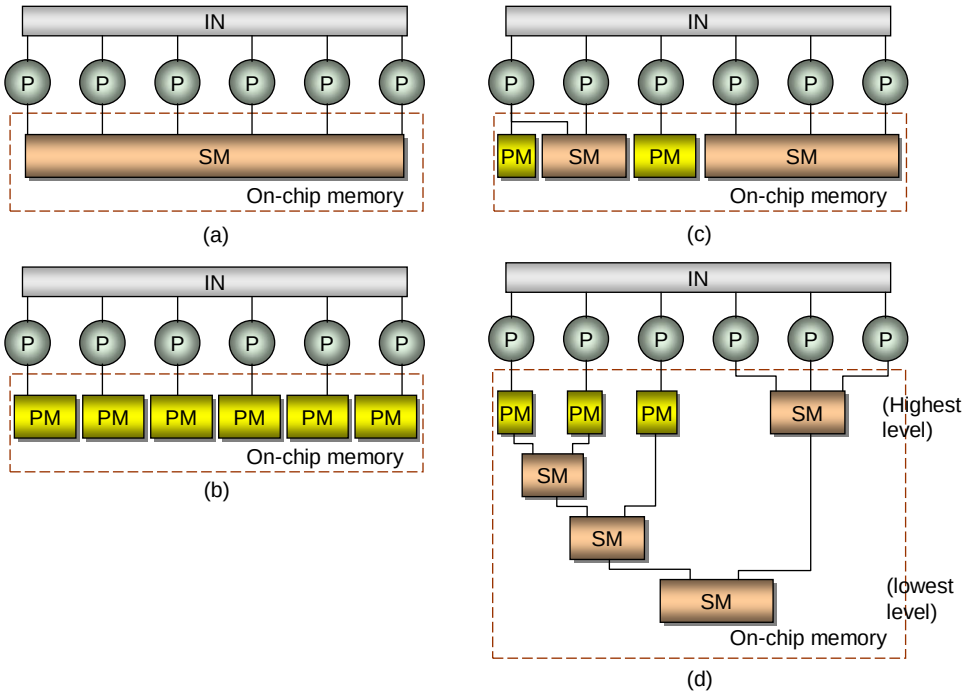


Fig. 3.1 Different on-chip memory designs. (a) Pure shared memory architecture. (b) Pure private memory architecture. (c) Single level based design. (d) Hierarchy based design. SM = shared memory; PM = private memory; IN = interconnection network.

least energy consumption, memory space partitioning and data allocation should be handled at concert. We also found that the solution times taken by our ILP-based approach are not excessive and are within tolerable limits.

3.1 Problem Formulation

3.1.1 Operation of On-Chip Memory Hierarchy and Memory Management Granularity

The on-chip memory hierarchy returned by our approach operates as follows. When a processor executes a memory operation, the memory component that holds the

corresponding data is accessed (and its access latency and energy consumption is incurred). Recall that the on-chip memory space in our framework is software managed. If the required data is not located in a memory component connected to the requester processor and it resides in a remote (but on-chip) memory component, it is accessed from there (this typically incurs a higher energy consumption than a local access). If this also fails, the data is accessed from the off-chip memory.

In our approach, the data transfers across the memory components take place at phase (epoch) boundaries. Specifically, the program is divided into phases and profiled. The profile data gives us the data blocks (considering all array data in the application) accessed by each processor at each phase, and an estimation of the number of accesses to each data block by processors. This information is then passed to the ILP solver (which will be discussed in detail shortly), which in turn determines the locations of data blocks for each phase and optimal on-chip memory hierarchy. After that, this information is passed to the compiler which modifies the application code accordingly to insert explicit data transfer calls. Our main focus is on ILP formulation of the problem and an experimental evaluation of the resulting framework. We do not focus on the details of the code modification phase, which is rather straightforward. Also, we do not address the problem of optimal phase partitioning or data block size selection.

3.1.2 ILP Formulation

ILP provides a set of techniques that solve those optimization problems in which both the objective function and constraints are linear functions and the solution variables

are restricted to be integers. The 0-1 ILP is an ILP problem in which each (solution) variable is restricted to be either 0 or 1 [103]. It is used in this thesis for determining the multi-level on-chip memory hierarchy for a chip multiprocessor architecture and data allocation. Table 3.1 gives the constant terms used in our ILP formulation. We used *Xpress-MP* [1], a commercial solver, though its choice is orthogonal to the focus of this thesis. The goal of our ILP formulation is to minimize the energy consumption in the memory subsystem of a chip multiprocessor. Specifically, our solution decides how on-chip memory components are being shared among processors, what the size of each memory component is, and what level in the hierarchy each memory component is located.

Our objective is to minimize the energy consumption within the memory subsystem, and this can be achieved by dividing the available memory space in such a way that each processor can access the most frequently used data (by itself) from the closest memory location of the right size. By doing so, we can reduce the energy cost of accessing the lower levels of the memory or accessing a remote memory component. There are $N_{MC} = 2^{N_P} - 1$ memory components possible for any level in the hierarchy, each of which can be accessed by different set of processors (N_P is the number of processors). For example, if there are two processors in the architecture, there are possibly three on-chip memory components. These are m_1 (accessed by only processor 1), m_2 (accessed by only processor 2), and m_3 (accessed by both processors 1 and 2). Note that, we are not restricted to have only one copy of each memory component type[1], that is, it is possible to have the same type of memory component at different levels of the memory hierarchy,

[1] A "type" in this context corresponds to a memory component accessed by a specific set of processors.

Table 3.1 The constant terms used in our ILP formulation. These are either architecture specific or program specific.

Constant	Definition
N_P	Number of processors
N_{MC}	Number of possible memory components
N_L	Number of levels in the memory hierarchy
S_M	Total on-chip memory size
N_B	Number of data blocks
S_B	Size of a data block
N_{Ph}	Number of program phases
$L_{size}(i)$	A function that returns the maximum possible size for i^{th} level

except that these same memory component types at different levels should get larger as we move from the higher level to the lower level. This follows from the fact that having a smaller or equal size memory component at a lower level in a hierarchy does not provide any energy benefits.

We assume for now that the entire data manipulated by the application can be stored in the on-chip memory hierarchy (we relax the assumption in Section 3.1.3). We use an access matrix A as an input to hold the number of accesses made by each processor to each data block. Mathematically, we define:

- $A_{p,ph,b}$: the number of accesses to data block b by processor p during program phase ph[2].

Our approach uses 0-1 variables to specify the size of each on-chip memory component (C) possible. Specifically,

- $C_{l,u,m}$: indicates whether memory component u at level l is of size m.

We use M to identify the location of a data block during the course of execution. More specifically,

[2]This matrix is filled after profiling and analyzing the application code.

- $M_{ph,b,l,u,m}$: indicates whether data block b at program phase ph is located in memory component u of size m at level l.

Both $C_{l,u,m}$ and $M_{ph,b,l,u,m}$ are 0-1 variables whose values we want to determine. E is used to capture the energy consumption of a processor at a specific program phase. That is,

- $E_{p,ph}$: captures the energy consumed by processor p during program phase ph.

Since the available on-chip memory space is limited, the following constraint must be satisfied:

$$\sum_{i=1}^{N_L} \sum_{j=1}^{N_{MC}} \sum_{k=0}^{S_M} k \times C_{i,j,k} = S_M. \tag{3.1}$$

In the above expression, memory components are allowed to have sizes of 0, which indicates that these memory components do not actually exist (in the final memory hierarchy designed).

The next constraint indicates that a memory component can have one and only one size:

$$\sum_{i=0}^{S_M} C_{l,u,i} = 1, \quad \forall l, u. \tag{3.2}$$

As we go to the higher levels in the hierarchy, the sizes of the memory components decrease due to space and cost issues. We use the following expression to set a maximum capacity limit for each level in the on-chip memory hierarchy:

$$\sum_{i=1}^{N_{MC}} \sum_{j=0}^{S_M} j \times C_{l,i,j} \leq L_{size}(l), \quad \forall l. \tag{3.3}$$

Although it is possible to come up with different functions to obtain the level size constraint, in our current implementation, we use the following expression for $L_{size}(l)$:

$$L_{size}(l) = \lceil (S_M \times l) / \sum_{i=1}^{N_L} i \rceil. \tag{3.4}$$

A lower level memory component connected to a higher level memory component should be at least accessible by the processors of the higher level memory component. That is, the connections between the higher and lower levels of memories should be in such a way that as the level of the memory component decreases, the number of processors that can access it should increase. For example, if a memory component at level 3 is connected to processors 1 and 2, it should not be connected to a memory component only connected to processor 1 at level 2. We use the term $subset(m_1, m_2)$ to indicate that the processors accessing memory component m_1 is a subset of the processors accessing memory component m_2. This is a matrix given as an input to the ILP solver, which represents all possible memory component pairs in mathematical terms.

$$C_{l+1,u,m} \leq \sum_{i=1}^{N_{MC}} \sum_{j=m}^{S_M} C_{l,i,j}, \forall l, u, m : l \leq N_L - 1 \ \wedge \ subset(u, i). \tag{3.5}$$

As stated earlier, it is possible that a memory component type (for example a memory component accessed by processors 1 and 2) can appear at the different levels in the memory hierarchy. In this case, the size of the memory component at a higher level should be less than the one at the lower level. This can be captured as follows:

$$C_{l_1,u,m_1} \leq 1 - C_{l_2,u,m_2}, \forall l_1, l_2, u, m_1, m_2 : m_1 \geq m_2 \ \wedge \ l_1 > l_2. \tag{3.6}$$

We now give the constraints for locating the data blocks within the memory components. We start by observing that a data block should be mapped to a single memory component. This is expressed as:

$$\sum_{i=1}^{N_L}\sum_{j=1}^{N_{MC}}\sum_{k=0}^{S_M} M_{ph,b,i,j,k} = 1, \quad \forall ph, b. \tag{3.7}$$

If a memory component is used for a data block, that memory component should exist in the final hierarchy. Consequently, we have

$$C_{l,u,m} \geq M_{ph,b,l,u,m}, \quad \forall ph, b, l, u, m. \tag{3.8}$$

The data stored in a memory component should be less than the size of the memory component. In mathematical terms, this can be captured as:

$$\sum_{i=1}^{N_B} M_{ph,b,l,u,m} \times S_B \leq m, \quad \forall ph, b, l, u, m. \tag{3.9}$$

Energy consumption of processor p at program phase ph can be calculated by summing up the energy consumptions due to data accesses performed by p:

$$E_{p,ph} = \sum_{i=1}^{N_B}\sum_{j=1}^{N_L}\sum_{k=1}^{N_{MC}}\sum_{l=0}^{S_M} M_{ph,i,j,k,l} \times A_{p,ph,i} \times AE(local(p,k), j, l, ports(k)), \quad \forall p, ph. \tag{3.10}$$

In the above expression, $A_{p,ph,i}$ returns the number of data accesses to data block i by processor p during program phase ph. There are four parameters that affect the energy consumption. These are data locality, the level of the memory component, the size of the

memory component, and the number of ports the memory component has. In addition to accessing the non-local data (i.e., a data in a remote on-chip memory component), accessing a lower level memory component will require a higher interconnect energy consumption. Similarly the size of the memory component being accessed is another key parameter to energy consumption. As the number of processors sharing the memory component increase, the required number of ports will increase. This in turn will increase the energy consumption. We use function $AE(location, level, size, ports)$ to capture the energy consumption of a specific access. In the above expression, the locality is captured by $local(p, k)$, which returns 1 if data block k is local to processor p; that is, it is stored in one of the memory components connected to processor p.

We next give our objective function:

$$\min \quad \sum_{i=1}^{N_P} \sum_{j=1}^{N_{Ph}} E_{i,j}. \tag{3.11}$$

Since $E_{i,j}$ denotes the energy consumption of processor i at program phase j, the total energy consumed by all processors throughout the entire program execution (i.e., over all program phases) can be calculated by summing them up.

3.1.3 Incorporating Off-Chip Memory

Recall that so far we assumed the entire data manipulated by the application can be stored in on-chip memory hierarchy. If we are to relax this assumption by allowing a smaller on-chip memory space than the total data size, then off-chip memory accesses have to be included in the total energy consumption expression. In order to do that, first,

Expression (3.7) has to be replaced with:

$$\sum_{i=1}^{N_L}\sum_{j=1}^{N_{MC}}\sum_{k=0}^{S_M} M_{ph,b,i,j,k} \leq 1, \quad \forall ph, b \tag{3.12}$$

In the above expression, a data block is ensured to reside in at most one on-chip memory component. That is, it is possible that a data block is not in one of the on-chip memory components. In addition to this modification, the energy consumed during off-chip memory accesses needs to be calculated. $O_{ph,b}$ in the expression below captures whether a data block is in the off-chip memory or not:

$$O_{ph,b} \geq 1 - \sum_{i=1}^{N_L}\sum_{j=1}^{N_{MC}}\sum_{k=0}^{S_M} M_{ph,b,i,j,k} \quad \forall ph, b. \tag{3.13}$$

The energy consumption due to off-chip accesses can then be calculated as follows:

$$OE_{p,ph} = \sum_{i=1}^{N_B} O_{ph,b} \times A_{p,ph,i} \times AE_{off} \quad \forall p, ph. \tag{3.14}$$

In this expression, AE_{off} denotes the energy consumption of a single off-chip data block access. Based on these modifications, the objective function given in Expression (3.11) is replaced with:

$$\min \quad \sum_{i=1}^{N_P}\sum_{j=1}^{N_{Ph}} (E_{i,j} + OE_{i,j}). \tag{3.15}$$

3.1.4 Example

We now give an example showing how our approach works. We assume that there are 3 processors and 2 program phases with 7 data blocks, each with a size of 1K.

Table 3.2 The number of data block accesses made by different processors.

Processor	D_1	D_2	D_3	D_4	D_5	D_6	D_7
Phase 1							
1	0	0	1	16	2	2	3
2	9	6	3	0	2	1	2
3	8	6	1	0	3	2	2
Phase 2							
1	12	0	2	0	1	3	2
2	0	7	1	8	2	2	3
3	0	7	3	9	2	1	2

The total on-chip memory space is assumed to be 7K with 2 levels. Table 3.2 shows the number of data block accesses made by the different processors at each of the two phases. In Figure 3.2 the memory hierarchy generated by our approach is shown. The data allocation for phase 1 is shown on the left-hand-side and that for phase 2 is shown on the right-hand-side. As can be seen from this figure, data block D_4 is privately used by processor P_1 during phase 1, whereas it is shared by processors P_2 and P_3 during phase 2. Since data blocks D_3, D_5, D_6 and D_7 are used by all the processors during both the phases, they are placed into a shared memory component residing at level 2. Note that these four data blocks are used by all the processors but not as frequently as the other data blocks.

3.1.5 Further Extensions

Recall from our earlier discussion that, in our final design, as we go to the higher levels in the hierarchy, the sizes of the memory components decrease. If we are to relax this assumption by allowing smaller memory components to reside at lower levels, then the relative size constraint captured by Expression (3.6) in our formulation can be dropped. This expression ensures that the size of the memory component at a higher level

is less than the one at the lower level. Similarly, Expression (3.5), which is used to limit the connections between different layers of the hierarchy, needs to be modified. In this expression, index variable j iterates over the available size spectrum from m to S_m. To accommodate our modification, we need to set the lower bound of j to 0. Consequently, Expression (3.5) should be replaced with:

$$C_{l+1,u,m} \leq \sum_{i=1}^{N_{MC}} \sum_{j=0}^{S_M} C_{l,i,j}, \forall l, u, m : l \leq N_L - 1 \wedge subset(u, i). \tag{3.16}$$

Another relaxation to our baseline approach could be to allow data block duplication in the different memory components and at the different levels. In order to accommodate this modification, Expression (3.7) needs to be modified to allow a data block to reside in more than one memory component. More specifically, we have:

$$\sum_{i=1}^{N_L} \sum_{j=1}^{N_{MC}} \sum_{k=0}^{S_M} M_{ph,b,i,j,k} \geq 1, \quad \forall ph, b. \tag{3.17}$$

In addition to the above modification, the energy consumption of processor p at program phase ph (that is, $E_{p,ph}$), has to be modified as well:

$$E_{p,ph} = \sum_{i=1}^{N_B} \min(\sum_{j=1}^{N_L} \sum_{k=1}^{N_{MC}} \sum_{l=0}^{S_M} M_{ph,i,j,k,l} \times A_{p,ph,i} \times AE(local(p,k), j, l, ports(k))), \quad \forall p, ph. \tag{3.18}$$

In the above expression, there could be more than one instance of a data block, each with a different energy consumption. In this case, the requesting processor will access the closest

data block, which will consume the least energy. To obtain this minimum value, extra 0-1 variables need to be defined; however, we do not elaborate on this any further.

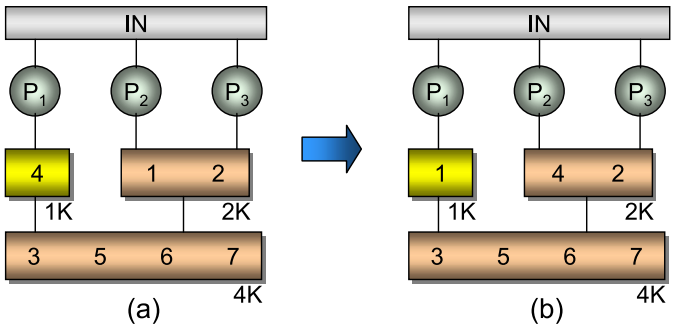


Fig. 3.2 An example on-chip memory hierarchy design and data allocation. (a) After phase 1. (b) After phase 2. The numbers within the memory components are data block ids.

3.1.6 Experimental Evaluation

Our goal in this section is two-fold. First, we want to show that it is possible to apply our ILP-based approach to realistic embedded application codes. For this purpose, we test our approach using ten real-world application codes extracted from the domain of embedded image/video processing. Our second goal is to compare our approach to alternate on-chip memory design schemes and check whether it really makes sense to employ this approach. Such an evaluation is important given that the ILP solvers usually take more time than fast heuristic solutions.

3.1.6.1 Setup

Table 3.3 gives the important properties of the ten application codes used in this study. These benchmark codes are collected from several research groups who work on embedded image/video application development. The second column of this table gives a description of each benchmark and the next one shows the number of source code lines (all these codes are array-based and are written in the C language). The fourth column gives the number of memory references made by each application. Note that this number is an inherent characteristic of an application and is independent of the particular memory hierarchy used during execution. The next two columns give the memory energy consumption (i.e., the energy consumption due to memory references including both on-chip components and off-chip memory) for the pure private memory based and pure shared memory based on-chip memory architectures (see Figure 3.1) under the default (base) simulation parameters given in Table 3.4.

Table 3.3 Application codes and their important properties.

Benchmark	**Brief Description**	**Number of C Lines**	**Number of Memory References (M)**	**Memory Energy Consumption (mJ)**	
				Pure Shared	**Pure Private**
Compress	Digital Image Compression	127	625.2	196.2	183.1
Conv-Spa	Convolution in Spatial Domain	231	811.4	113.4	109.6
Filter	Triangular Filter	270	118.6	106.7	115.7
Laplace	Morphological Laplacian	438	109.6	310.3	301.0
LU-Decomp	LU Decomposition	86	227.3	256.7	280.3
Minkowski	Image Dilation and Erosion	455	839.5	576.8	602.2
Seg-02	Image Segmentation	622	107.1	605.2	584.7
Text	Texture Analyzer	1018	144.9	899.3	813.9
Img-Trans	Image Transformation	972	222.9	1142.8	1228.5
Verifier	Image Verifier	1353	196.3	986.7	976.9

Table 3.4 Important base simulation parameters used in our experiments.

Parameter	Default Value
Number of Processors	8
Total On-Chip Memory Space	2MB
Data Block Size	128 bytes
Off-Chip Memory Capacity	32MB
Off-Chip Memory Access Latency	100 cycles
Bus Arbitration Delay	5 cycles
Replacement Policy	LRU
Maximum Number of Levels in Hierarchy	5

To conduct our experiments, we modified the Simics [96] tool-set (running on Solaris 9 operating system) and simulated the different schemes. The default configuration parameters are given in Table 3.4, and these are the values that are used unless explicitly stated/varied in the sensitivity experiments. The energy consumption and latency values for the different memory components (i.e., the AE values used in our formulation) are obtained using the CACTI tool [137] under the 0.07μ process technology. The two most important factors that affect the energy consumption and access latency of a memory components are the size (capacity) and the number of ports. We modeled using CACTI memory components of different capacities and ports. A few necessary changes were needed to be made to the basic CACTI model to obtain the energy model for software-managed memory (as against conventional caches).

For each of the ten applications shown in Table 3.3, we performed experiments with five different schemes. Pure Shared (PS) and Pure Private (PP) are the two conventional schemes. In PS, there is a single monolithic on-chip memory space shared by all the processors. In PP, each processor has its private on-chip memory, i.e., the total on-chip memory space is divided equally among the parallel processors. ILP-ML represents the ILP-based multi-level on-chip memory hierarchy design and data allocation

strategy discussed in this chapter. ILP-SL is similar to ILP-ML; the difference is that it uses only a single level (i.e., the available on-chip memory space is partitioned across the processors optimally without having a hierarchy, as exemplified in Figure 3.1(c)). It is important to emphasize that all these versions use optimized data allocation across memory components, which is determined using integer linear programming. Our goal in making experiments with the ILP-SL version is to isolate the energy benefits that are coming explicitly from memory hierarchy design. Our next scheme, ILP-ML* is similar to ILP-ML, except that it does not perform optimal data allocation. Instead, it allocates data based on the following heuristic. The program is profiled and the frequently accessed data by each processor is placed into the memory components which is directly accessible by that processor. The idea is to minimize the energy consumption spent in accessing frequently used data. The reason why we make experiments with this version is to isolate the benefits that are coming from optimal data allocation; that is, to demonstrate that optimal memory hierarchy design alone is not sufficient for achieving maximum energy reductions, and a unified approach that combines data allocation and memory hierarchy design is necessary.

Before presenting our energy savings, we want to remark on our ILP solution times. The largest solution time we experienced during our experiments was 5.5 minutes (for application Verifier). Therefore, since memory hierarchy design is an offline process, we can say that our solution times are not excessive.

3.1.6.2 Base Results

We start by presenting the energy and performance results with the schemes explained above. Figure 3.3 gives the energy consumption values with different schemes.

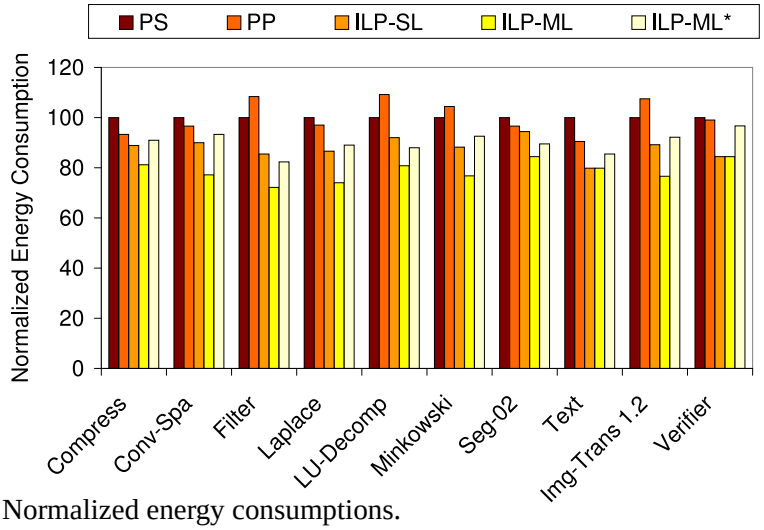


Fig. 3.3 Normalized energy consumptions.

All energy results are normalized with respect to the fifth column of Table 3.3. Our first observation from these results is that there is no clear winner between the PS and PP schemes. The relative performance of one scheme to another depends mostly on the amount of data shared by parallel processors and how these data are shared. When we look at the remaining three versions, we see that ILP-ML generates the best results for all the applications tested. The average energy reduction it brings is 21.3%. In comparison, the average energy reductions with the ILP-SL and ILP-ML* schemes are 12.1% and 9.9%, respectively. This result clearly emphasizes the importance of exploiting multiple levels (in designing memory hierarchy) and of optimal data allocation. In particular, when we compare the ILP-SL and ILP-ML schemes, we see that they generate the same results in only two benchmarks (Text and Verifier). On the other hand, the rest of the applications in our experimental suite take advantage of multiple levels. The reason why the ILP-ML* scheme performs poorly as compared to ILP-ML is lack of optimal data allocation. This means that optimal memory space partitioning and optimal data allocation should be carried out together if we are to minimize energy consumption.

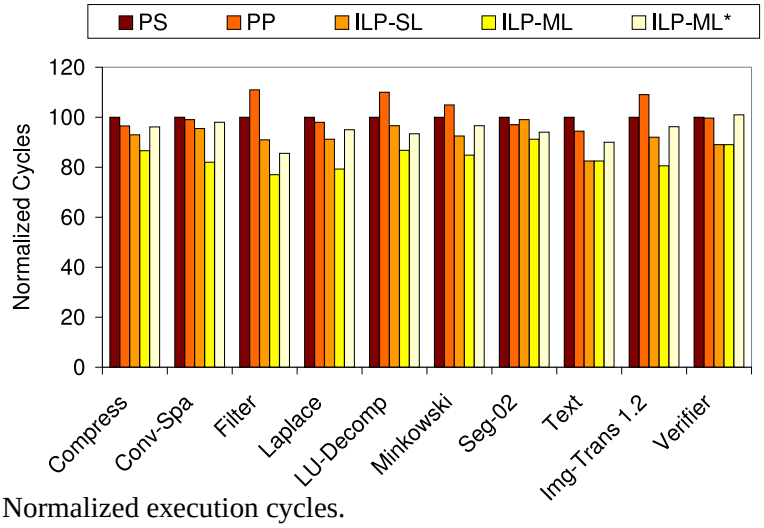


Fig. 3.4 Normalized execution cycles.

While our memory partitioning and data allocation approach is designed for minimizing energy consumption rather than optimizing performance, it is also important to consider its performance. This is because the resulting memory design may not be acceptable if it causes a significant increase in original execution cycles (as compared to conventional memory designs such as PP or PS), even though it could reduce memory energy consumption significantly. The normalized execution cycles achieved by the different schemes are shown in Figure 3.4. While these results are not as good as those shown in Figure 3.3 from the perspective of our ILP-based approach, we still observe important benefits. In fact, the ILP-ML scheme brings nearly 15% improvement in original execution cycles over the pure shared memory based scheme. We also need to mention that, to reduce execution cycles even further, the ILP-based approach can be targeted, with some modifications, at minimizing execution cycles as well (instead of minimizing energy consumption), though this option is not evaluated in this work. Still, when we consider the results shown in Figures 3.3 and 3.4 together, we see that the ILP-ML scheme is very useful from both performance and energy perspectives.

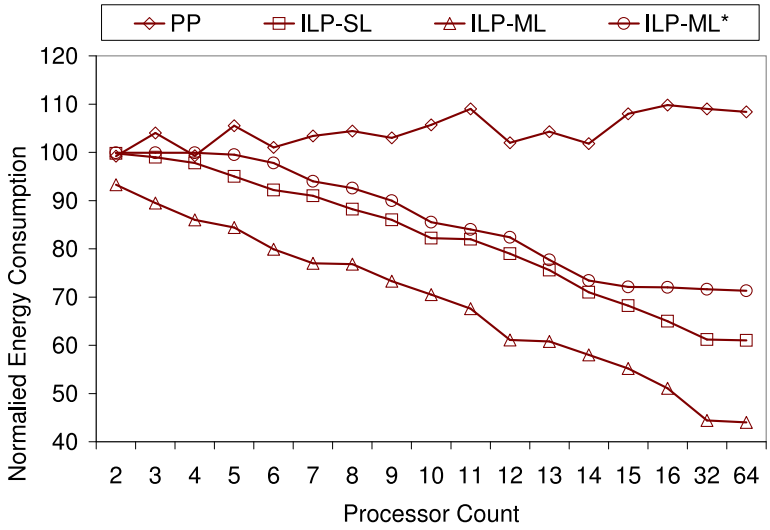


Fig. 3.5 Impact of processor count.

3.1.6.3 Sensitivity Analysis

In this section, our goal is to vary some of the default (base) values of the simulation parameters (shown in Table 3.4), and check how doing so affects our energy savings. We present our results for a single application only (Minkowski), due to space concerns. The first parameter we change is the number of processors (processor count), and the results are given in Figure 3.5. We see that all three ILP-based schemes evaluated take advantage of increased number of processors. The main reason for this behavior is the fact that, when the number of processor is increased, careful partitioning of on-chip memory space becomes more important, and all three ILP-based schemes allow us to achieve that to different extents.

The second parameter whose value we vary is the total on-chip memory size. The results are given in Figure 3.6. M/4 on the x-axis of this graph corresponds to the default on-chip memory capacity used so far in our experiments (M is the total amount of data processed by this application). We see that all the three ILP-based schemes start to

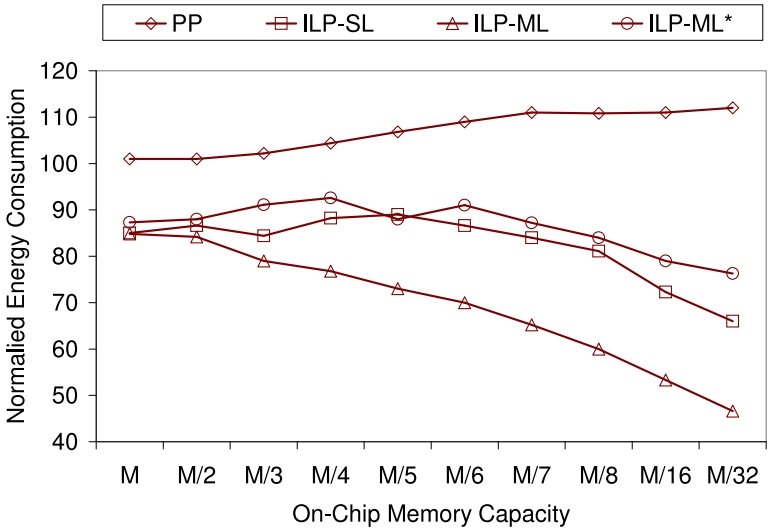


Fig. 3.6 Impact of on-chip memory capacity.

converge as the on-chip memory capacity is increased. This can be explained as follows. When the memory space is small, partitioning is important for minimizing energy consumption. However, when memory space is large, it becomes more of a problem of data allocation. Therefore, the benefits brought by the ILP-ML scheme are more emphasized with small memory sizes. This is in a sense an encouraging result though. This is because embedded applications are growing in terms of both complexity and data sizes, and the fact that our approach performs better with smaller memory spaces means that we can expect it to be even more successful in the future.

The third parameter we study is the number of levels in the on-chip memory hierarchy. Recall from Table 3.4 that in our experiments so far we allowed at most five memory levels. In the results shown in Figure 3.7 we give energy consumptions with different number of levels (from 1 to 5). Obviously, when we have only one level, ILP-ML reduces to ILP-SL. We see from these results the importance of working with large number of levels if it is possible to do so. This is because the energy savings reduce as

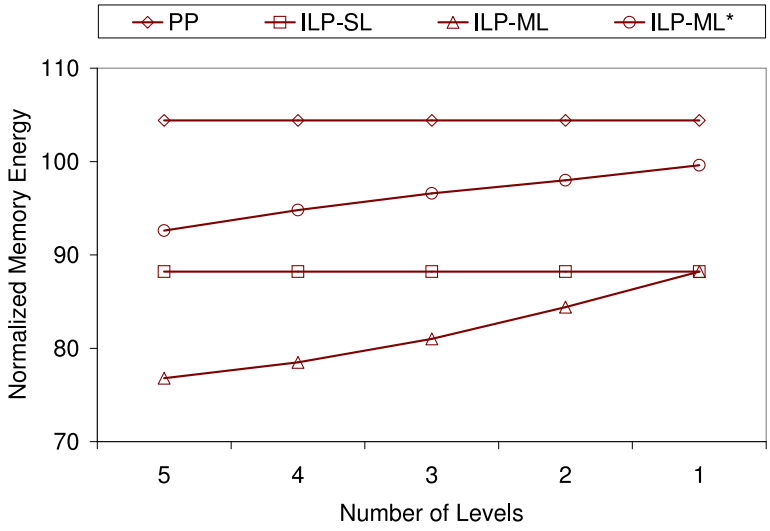


Fig. 3.7 Impact of the number of levels in the memory hierarchy.

we reduce the maximum number of allowable levels. However, not all the applications really use all the available levels. In fact, we observed during our experiments that, for any given application, there is a number of maximum levels beyond which we do not see any further energy reductions.

The last parameter we investigate is the data block size. Figure 3.8 gives the energy results with different block sizes (B represents the default block size, which is 128 bytes as given in Table 3.4). We see that both ILP-ML and ILP-SL take better advantage of smaller block sizes as compared to the ILP-ML* scheme. This is mainly because ILP-ML* does not employ optimal data allocation.

To summarize, our experimental analysis shows that the ILP-ML scheme performs really well from both energy and execution time perspectives. Also, our experiments with different parameters emphasize the importance of employing both optimal memory space

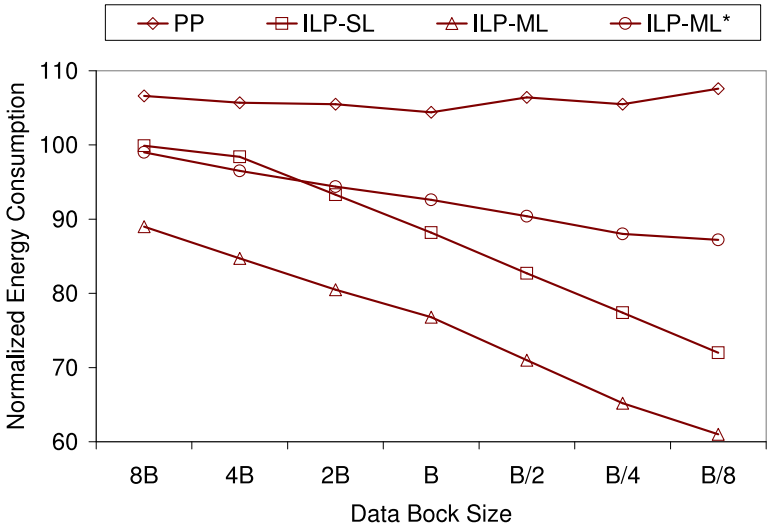


Fig. 3.8 Impact of data block size.

partitioning and optimal data allocation (together in a unified setting) for the best energy savings.

Chapter 4

Increasing On-Chip Memory Space through Data Compression

One of the important consequences of putting multiple processors on the same die is the reduction in the cost of inter-processor communication (as compared to the case where each processor resides on a separate chip). This is true from both performance and power perspectives. However, this architecture also magnifies the cost of off-chip memory accesses. This is because the main memory accesses in this architecture need to go through the costly off-chip interconnect to access large off-chip memory components. Therefore, it is critical to reduce the number of off-chip memory accesses as much as possible, even if this causes an increase in the volume of on-chip communication activities among the chip multiprocessor. In practice, this requires reusing on-chip data as much as possible. In other words, in the ideal case, when a data block is brought from the off-chip memory to the on-chip memory space, we want all accesses to that data block (potentially coming from different processors) to be made before it is returned to the off-chip memory.

Unfortunately, optimizing data locality for each processor independently of the other processors in the chip multiprocessor may not achieve this desired high reuse of shared data blocks. This is because many previously-proposed approaches to data locality optimization do not really care how shared data is accessed. As a result, the *inter-processor reuse distance* for a given data block, i.e., the difference in cycles (or loop iterations) between two successive accesses to the same data block by different processors

can be very large in practice. Let us consider a simple scenario, for illustrative purposes, where a data block is requested by two different processors. After the access to the data block by the first processor, we can keep the block in the local on-chip memory of that processor until the time it is required by the second processor. This will work fine if the inter-processor reuse distance is short, i.e., the second processor requests the data shortly after the first access completes. On the other hand, if the inter-processor reuse distance is large, we have two options. Either we can send the block to the off-chip memory, which means it will need to be brought back into the on-chip memory space when the second processor requests it. Or, we can continue to keep it in the on-chip memory space until the second request takes place. The drawback of the first option is clear: an extra off-chip access, which we desperately want to avoid. The drawback of the second scheme is that this block occupies space in the on-chip memory without any use until the execution point when the second processor requests it.

We propose a *data compression* based approach to reduce the negative impact of the second option mentioned above, i.e., keeping the data block in the on-chip memory even if the reuse distance is large. Specifically, we propose to keep such data blocks (with large inter-processor reuse distances) in the on-chip memory space in a compressed form. When the second request comes, we decompress the data block and forward it to the requester (an on-chip transfer). The advantage of this scheme is that since the data block remains compressed between the two accesses it occupies less memory in the on-chip memory space, which makes more space available for new blocks. The drawback is that the block needs to be decompressed before the second request. However, depending on the relative cost of decompression with respect to off-chip memory transfers, this could

still be less costly than sending the block to the off-chip memory after the first access and reading it back later for the second access. However, we should not compress the data block if its inter-processor reuse distance is short. Since a typical data-intensive parallel application running on an embedded chip multiprocessor can manipulate multiple data blocks at the same time (each potentially competing for the on-chip memory space), what we need is a global on-chip memory space optimization scheme.

Focusing on this problem under a software-managed memory hierarchy, we propose an automated compiler support that can help in deciding the set of data elements to compress/decompress and the points during execution at which these compressions/decompressions should be performed. Optimizing compiler first analyzes the source code of the application to be optimized and identifies the order in which different data blocks are accessed. Based on this analysis and data reuse information, the compiler then automatically inserts compression and decompression calls in the application code. The compression calls target the data blocks that are not expected to be used in the near future, whereas the decompression calls target those data blocks with expected reuse but currently in compressed form. In providing compiler support for data compression, we want to achieve two objectives. First, we want to enable memory space savings without the involvement of the application programmer; that is, the programmer gets the benefits of reducing the memory space consumption without any effort on her part. Second, we want to minimize the potential impact of compression on performance. To do this, our automated approach makes use of data reuse analysis.

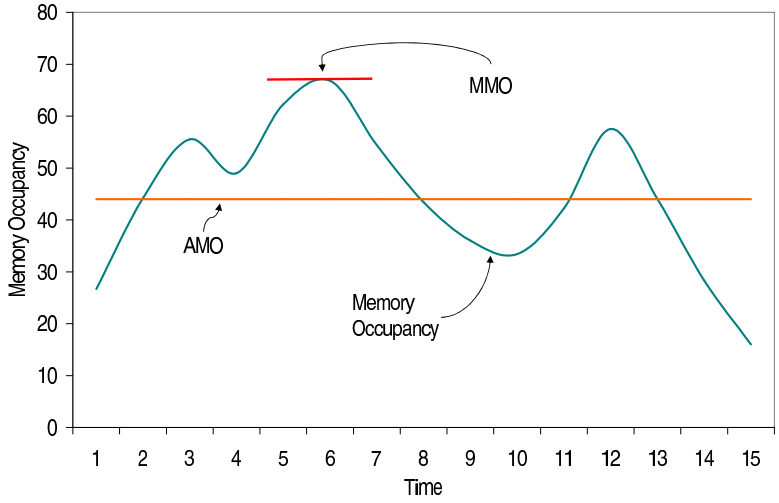


Fig. 4.1 An example memory space consumption curve and MMO and AMO metrics.

We consider two different execution scenarios, one with single processor and one with multiple processors. In the multiprocessor case, the compressions and decompressions can be carried out by a management thread, which is different from the main computation thread. The performance overhead of our approach depends on the scenario under consideration. Our experimental results show that the proposed approach reduces both maximum and average memory occupancies significantly. In addition, we show that it can save more memory space than an alternate compiler technique that exploits lifetime information to reclaim the memory space occupied by dead data blocks, that is, the data blocks that completed their last uses. Our results also indicate that best memory space savings are achieved by combining our data compression-based approach with dead block recycling. It needs to be emphasized that we do not propose a new data compression technique. Instead, we demonstrate how an optimizing compiler for embedded systems can schedule data compressions and decompressions intelligently to minimize memory space occupancy at runtime while also minimizing the associated performance overheads.

4.1 Memory Space Occupancy

Memory space occupancy indicates the memory space occupied by an application data at each point during the course of execution. There are two important metrics associated with memory space occupancy. The first one is the *maximum memory occupancy* (MMO), which gives the maximum memory space occupied by data during the course of execution when considering all execution cycles. The second metric is the *average memory occupancy* (AMO), which gives the memory occupancy when averaged over all execution cycles. Figure 4.1 illustrates these two metrics for an example case. Note that, the drops in the memory occupancy curve indicate either some application-level dead memory block recycling or system-level garbage collection. Both these metrics, MMO and AMO, can be important targets for optimizations. MMO is critical since it captures the amount of memory that needs to be allocated for the application if the application is to run successfully without an out-of-memory exception. The AMO metric on the other hand can be important in a multi-programming based embedded environment where multiple applications compete for the same memory space. The goal behind our compiler-directed approach is to reduce both MMO and AMO for array/loop-intensive embedded applications. Note that, array/loop-intensive applications are frequently used in embedded image/video processing.

4.2 Compiler Algorithm

Employing data compression in managing the memory space of the system requires a careful analysis of the data access pattern of the application being considered.

This is because using data compression in an untimely manner can cause significant performance and power penalties. For example, compressing data blocks with short reuse distances can increase the number of decompressions dramatically. Also, decompressing data blocks with long reuse distances prematurely can increase memory space consumption unnecessarily. Therefore, one needs to be very careful in deciding both the set of data blocks to compress/decompress and the points in execution to compress/decompress them. Clearly, this is an area that can benefit a lot from automation. Our goal is to reduce MMO and AMO as much as possible, with as little performance penalty as possible.

4.2.1 Data Tiling and Memory Compression

Our scheme compresses only the arrays that can benefit from data compression (this can be determined either through profiling or via programmer annotations). These arrays are referred to as the "compressible" arrays. We do not compress scalar variables or incompressible arrays (i.e., the arrays that cannot benefit from data compression). Figure 4.3 shows the organization of the memory space for supporting our compiler-directed data compression approach. We divide the memory space into three parts: *compressed area*, *decompression buffer*, and *static data area*. The static data area contains scalar variables, incompressible arrays, and the directories for compressible arrays. The data entities in the static area are statically allocated at compilation time. The compressed area and the decompression buffer, however, are dynamically managed at runtime based on the compiler-determined schedule for compressions and decompressions.

We divide each compressible array into equal-sized *tiles* (*blocks*). An element of a tiled array X can be indexed using the following expression: $X[\![\vec{I}]\!][\vec{J}]$, where $\vec{I}$ is the tile

subscript vector, which indexes a tile of array X; and $\vec{J}$ is the intra-tile subscript vector, which indexes an element within a given tile. For example, Figure 4.2 shows an array X that has been divided into nine (3×3) tiles, and each tile contains sixteen (4×4) elements. $X[\![2,3]\!]$ refers to the tile at the second row, third column; and $X[\![2,3]\!][3,2]$ refers to the data element at the third row, second column of tile $X[\![2,3]\!]$.

Figure 4.3 shows how we store tiled arrays in the memory. For each compressible array X, our compiler creates a directory, each entry of which corresponds to a tile of array X, and can be indexed using a tile subscript vector. Each entry in the directory of array X, denoted as $X[\![\vec{I}]\!]$ ($\vec{I}$ is a tile subscript vector), contains a pointer to the memory location where the corresponding tile is stored. As mentioned above, the directory of each array is stored in the static data area of the memory space.

An array tile can either be compressed or uncompressed. Uncompressed tiles are stored in the decompression buffer, and compressed tiles are stored in the compressed area. The decompression buffer is divided into equal-sized blocks, and the size of a block is equal to that of a tile. We use a *free table* to keep track of the free blocks in the decompression buffer. When we need to decompress a tile, we first need to allocate a free block in the decompression buffer.

Compressed tiles are stored in the compressed area. The memory in this area is divided into equal-sized *slices*. The size of a slice is smaller than that of a block in the decompression buffer. In our implementation, a slice is equal to a quarter of a block. Although the size of tiles is a constant, the compression ratio depends on the specific tile. Therefore, the number of slices required to store a compressed tile may vary from one tile to another. In Figure 4.3, we can observe that the slices of the same tile form a link table.

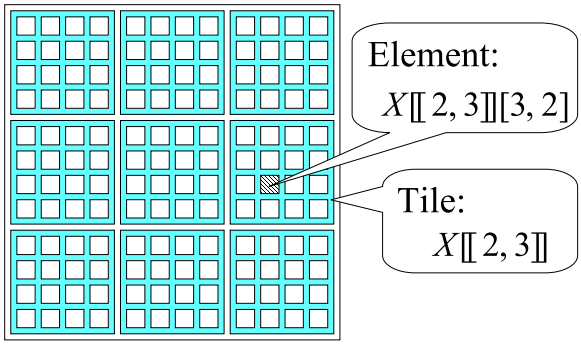


Fig. 4.2 Data tiling for array X.

Like in the case of the decompression buffer, the compressed area also has a free table keeping all free slices.

Figure 4.4 shows the architecture of our system. When the program starts its execution, all tiles are in the compressed format and are stored in the compressed area. A compressed tile is decompressed and stored in the the decompression buffer by a *decompressor* before it is accessed by the program. If this tile belongs to an array that is not written (updated) by the current loop nest, the compressed version of this tile remains in the compressed area. On the other hand, if this tile belongs to an array that might be written by the current loop nest, we discard the compressed version of this tile, and return the slices occupied by this tile to the free table of the compressed area. When we need to decompress a new tile but there is no free space in the decompression buffer, we select a set of old tiles in the decompression buffer and discard them to make space for the new tile. If a victim tile (the tile to be evicted) belongs to an array that might be written by the current loop nest, we must decompress and store its compressed version in the compressed area before we evict its uncompressed version. On the other hand, if this tile belongs to an array that is not written by the current loop nest, we can discard the uncompressed

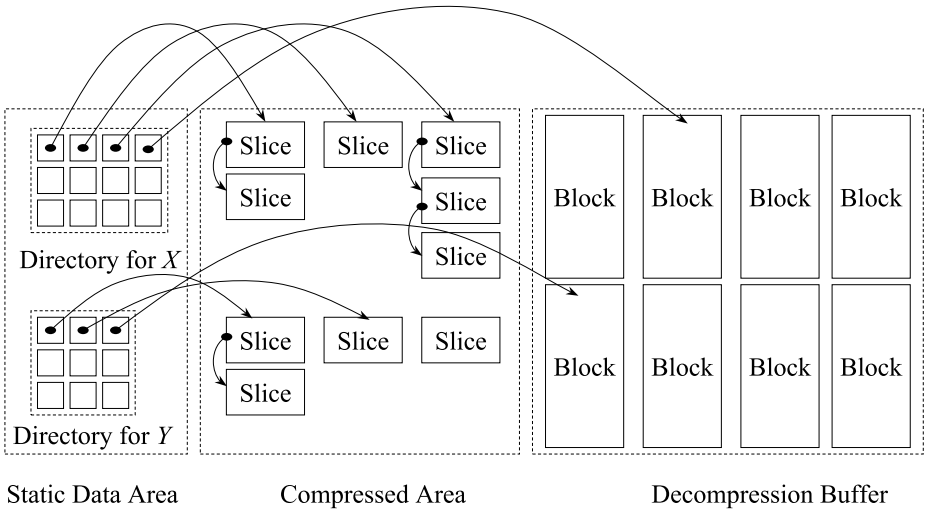


Fig. 4.3 Memory organization.

version of this tile without recompressing it. The important point to emphasize here is that our approach is not tied to any specific compression/decompression algorithm, and the compressor and decompressor can be implemented either in software or hardware. In our current implementation, however, we use only software compression/decompression.

It should be emphasized that data tiling is required by our implementation of memory compression, not by the logic of the application. Therefore, we do not require the programmer to be aware of data tiling. Our compiler automatically (in a user-transparent manner) tiles every array that needs to be compressed. Data tiling requires two mapping functions p and q that map an original array subscript vector to a tile subscript vector and an intra-tile subscript vector, respectively. That is, we map $X[\vec{I}]$ into $X[\![p(\vec{I})]\!][q(\vec{I})]$. Given a tile size T, we use the following mapping functions:

$$p((i_1, i_2, ..., i_n)) = (\lfloor i_1/N_1 \rfloor, \lfloor i_2/N_2 \rfloor, ..., \lfloor i_n/N_n \rfloor);$$

$$q((i_1, i_2, ..., i_n)) = (i_1 \mod N_1, i_2 \mod N_2, ..., i_n \mod N_n);$$

where $N_1, N_2, ..., N_n$ are magnitudes (extents) of each dimension subscript vector such that $N_1 N_2 ... N_n = T$. When an array is tiled, our compiler also *rewrites* the program statements that access this array accordingly.

4.2.2 Loop Tiling

While data tiling transforms memory layout of each compressible array, loop tiling (iteration space blocking) transforms the order in which the array elements are accessed within a loop nest. If used appropriately, loop tiling can significantly reduce the number of decompressions invoked during the execution of a loop nest. Figure 4.5 gives such an example. Figure 4.5(a) is the original code of a loop nest, which accesses a 600×600 array X. We apply data tiling to array X such that the size of each tile is 100×100. Figure 4.5(b) shows the code after data tiling. For illustration purposes, let us assume that the decompression buffer can contain up to three tiles, and that we use an LRU based policy to select the victim tiles in the decompression buffer. We can compute that, during the execution of this loop nest, we need to invoke the decompressor 100 times for each tile. Hence, the decompressor is invoked $100 \times 36 = 3600$ times. By applying loop tiling to the loop nest shown in Figure 4.5(b), we obtain the tiled loop nest in Figure 8.5(c). In this tiled code, loop iterators i and j are the *inter-tile iterators* and the loop nest formed by them is referred to as the *inter-tile loop nest.* Similarly, the iterators ii and jj are the *intra-tile iterators* and the loop nest formed by them is referred to as the *intra-tile loop nest.* During the execution of this loop nest, the decompressor is invoked only 36 times, once for each i, j combination. Loop tiling has been widely studied in the literature (e.g., [159]). Due to the space limitation we have, we do not discuss the details of loop tiling.

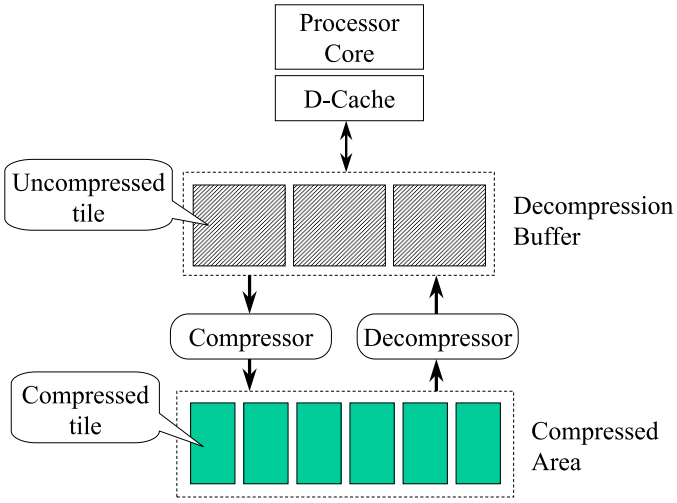
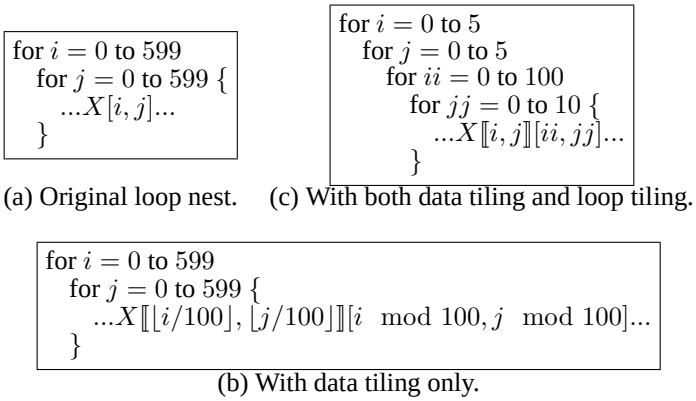


Fig. 4.4 Architecture supporting memory compression.

In the rest of this section, we assume that the loop nests in the application program have been appropriately tiled according to the layout of the arrays imposed by data tiling. It is to be mentioned however that our compiler uses loop tiling for a different purpose than most of the commercial and academic compilers.

4.2.3 Compression-Based Space Management

Our compiler inserts buffer management code at each loop nest that uses the decompression buffer. For ease of discussion, we use the following abstract form to represent a tiled loop nest:



(a) Original loop nest. (c) With both data tiling and loop tiling.

(b) With data tiling only.

Fig. 4.5 Code transformation for data tiling and loop tiling.

$$
\begin{aligned}
&\text{for } \vec{I} = \vec{L} \text{ to } \vec{U} \ \{ \\
&\quad T_1(R_1(\vec{I}), W_1(\vec{I})); \\
&\quad T_2(R_2(\vec{I}), W_2(\vec{I})); \\
&\quad \ldots \ldots \\
&\quad T_n(R_n(\vec{I}), W_n(\vec{I})); \\
&\}
\end{aligned}
$$

where $\vec{I}$ is the iteration vector, $\vec{L}$ and $\vec{U}$ are the lower and upper bound vectors for the loop nest. T_i ($i = 1..n$) represents an intra-tile loop nest. Since we focus on the access pattern of each array at a tile level, we treat each intra-tile loop nests as an atomic operation. $R_i(\vec{I})$ is the set of tiles that might be read, and $W_i(\vec{I})$ is the set of tiles that might be written in the intra-tile loop nest T_i at the inter-tile iteration $\vec{I}$. $R_i(\vec{I})$ and $W_i(\vec{I})$ can be

computed as follows:

$$R_i(\vec{I}) = \{X_k[\![f_j^{(i)}(\vec{I})]\!] \mid \text{“... = ...}X_k[\![f_j^{(i)}(\vec{I})]\!][...]\text{...” appears in } T_i\},$$
$$W_i(\vec{I}) = \{X_k[\![f_j^{(i)}(\vec{I})]\!] \mid \text{“}X_k[\![f_j^{(i)}(\vec{I})]\!][...] = ...\text{” appears in } T_i\},$$

where X_k is an array accessed by T_i, and $f_j^{(i)}(\vec{I})$ is a mapping function that maps inter-tile iteration vector $\vec{I}$ into a tile of array X_i. Note that, we must be conservative in computing $R_i(\vec{I})$ and $W_i(\vec{I})$.

Figure 4.6 shows a transformed loop nest augmented with the decompression buffer management code. In the transformed code, we use counter c to count the number of intra-tile loop nests that have been executed up to a specific point. B is the set of tiles that are currently in the decompression buffer. Before entering intra-tile loop nest T_i, we need to decompress all the tiles in the set $R_i(\vec{I}) \cup W_i(\vec{I}) - B$. That is, all the the tiles that will be used by T_i must be available in the uncompressed format before we start executing T_i. When decompressing a tile t, we may need to evict a tile from the decompression buffer if there is no free block in the decompression buffer (indicated by $|B| = D$). Each tile t in the decompression buffer is associated with an integer $t.r$, indicating when this tile will be reused in the future. Specifically, $t_1.r < t_2.r$ indicates that tile t_1 will be reused earlier than t_2. When evicting a tile, we select the one that will be reused in the furthest future. Each tile t in the decompression buffer also has a flag $t.w$ indicating whether this block has been written since its last decompression. If this victim tile has been written, we need to recompress this tile. Before entering T_i, we also update the *next reuse time* ($t.r$) for each tile (t) used by T_i. The next reuse time of tile t is computed using $t.r = c + d_i(t)$,

where c is number of intra-tile loop nests that have been executed, and $d_i(t)$ is the *reuse distance* of tile t at intra-tile loop nest T_i. The reuse distance of tile is the number of intra-tile loop nests executed between the current and the next accesses to this tile.

We use a compiler-based approach to compute $d_i(t)$–the reuse distance of tile t at intra-tile loop nest T_i. In the following discussion, we explain how $d_i(t)$ can be computed at compilation time. The tiles in $R_i(\vec{I}) \cup W_i(\vec{I})$, the set of tiles used by the i^{th} intra-tile loop nest at inter-tile loop iteration $\vec{I}$, can be divided into three types: (1) the tiles that will be reused by another intra-tile loop nest at the same inter-tile iteration $\vec{I}$, (2) the tiles that will be reused by some intra-tile loop nest at another inter-tile iteration $\vec{I'}$ ($\vec{I} \prec \vec{I'}$), and (3) the tiles that will never be reused. The set of tiles belonging to the first type, i.e., the tiles that will be reused at the same inter-tile loop iteration, can be computed as follows:

$$U_i(\vec{I}) = (R_i(\vec{I}) \cup W_i(\vec{I})) \cap \bigcup_{j=i+1}^{n} (R_j(\vec{I}) \cup W_j(\vec{I})),$$

where n is the number of intra-tile loop nests in the body of the inter-tile loop nest. For each tile $t \in U_i(\vec{I})$, the reuse distance can be computed as:

$$d_i(t) = j - i,$$

where j is the minimum integer greater than i such that $t \in R_j(\vec{I}) \cup W_j(\vec{I})$.

```
for I⃗ = L⃗ to U⃗ {
   ... ...
   T_i(R_i(I⃗), W_i(I⃗));
   ... ...
}
```

(a) Original loop nest.

```
B — the set of tiles in the buffer;
D — the size of buffer;
t — the tile to be loaded;
⇒  procedure load(t) {
⇒    if(t ∉ B) {
⇒      if(|B| = D) {
⇒        for each v ∈ B such that v.r < c
⇒          v.r = ∞; // the next use time of v has been mispredicted
⇒        select v ∈ B such that
⇒          directory for v is not marked and v.r is max;
⇒        if(v.w = 1) compress(v);
⇒        evict(v); B = B − {v};
⇒      }
⇒      decompress(t); B = B + {t};
⇒    }
⇒    t.r = c + d_i(t);
⇒  }

for I⃗ = L⃗ to U⃗ {
     ... ...
⇒    c = c + 1;
⇒    // mark the tiles in W_i(I⃗) ∪ R_i(I⃗),
⇒    // preventing these tiles from being evicted.
⇒    mark_directory_entry(X_{k_1}[[f_1^(i)(I⃗)]]);
⇒    mark_directory_entry(X_{k_2}[[f_2^(i)(I⃗)]]);
⇒    mark_directory_entry(X_{k_m}[[f_m^(i)(I⃗)]]);

⇒    // load tiles in W_i(I⃗)
⇒    load(X_{k_1}[[f_1^(i)(I⃗)]]); X_{k_1}[[f_1^(i)(I⃗)]].w = 1;
⇒    load(X_{k_2}[[f_2^(i)(I⃗)]]); X_{k_2}[[f_2^(i)(I⃗)]].w = 1;
⇒    load(X_{k_j}[[f_j^(i)(I⃗)]]); X_{k_j}[[f_j^(i)(I⃗)]].w = 1;

⇒    // load tiles in R_i(I⃗)
⇒    load(X_{k_{j+1}}[[f_{j+1}^(i)(I⃗)]]);
⇒    load(X_{k_{j+2}}[[f_{j+2}^(i)(I⃗)]]);
⇒    load(X_{k_m}[[f_m^(i)(I⃗)]]);

⇒    // unmark the tiles in W_i(I⃗) ∪ R_i(I⃗),
⇒    // allowing these tiles to be evicted.
⇒    unmark_directory_entry(X_{k_1}[[f_1^(i)(I⃗)]]);
⇒    unmark_directory_entry(X_{k_2}[[f_2^(i)(I⃗)]]);
⇒    unmark_directory_entry(X_{k_m}[[f_m^(i)(I⃗)]]);
     T_i(R_i(I⃗), W_i(I⃗));
     ... ...
}
```

(b) The transformed loop nest augmented with the decompression buffer management code. The lines marked with "⇒" are inserted by our compiler.

Fig. 4.6 Code transformation employed by our compiler.

```
for i = 1 to 2
  for j = 1 to 3 {
        c = c + 1;
        load(X⟦i, j⟧, X⟦i, j + 1⟧);
    L1: for ii = 1 to 10
          for jj = 1 to 10 {
            S1: ...X⟦i, j⟧[...]...    // reuse distance d1 = 3
            S2: ...X⟦i, j + 1⟧[...]...// reuse distance d2 = 1
          }
        c = c + 1;
        load(X⟦i, j + 1⟧, X⟦i, j − 1⟧);
    L2: for ii = 1 to 10
          for jj = 1 to 10 {
            S3: ...X⟦i, j + 1⟧[...]...// reuse distance d3 = 1
            S4: ...X⟦i, j − 1⟧[...]...// reuse distance d4 = ∞
          }
  }
```

(a) Example code fragment. The reuse distances for the tiles used at program statements S1, S2, S3, and S4 are 3, 1, 1, and ∞, respectively.

i	j	c	**Nest**	**Tiles of array X in the buffer**		
1	1	1	$\mathcal{L}_1$	$*\llbracket 1,1\rrbracket : 4$	$*\llbracket 1,2\rrbracket : 2$	
1	1	2	$\mathcal{L}_2$	$\llbracket 1,1\rrbracket : 4$	$\llbracket 1,2\rrbracket : 3$	$*\llbracket 1,0\rrbracket : \infty$
1	2	3	$\mathcal{L}_1$	$\llbracket 1,1\rrbracket : 4$	$\llbracket 1,2\rrbracket : 6$	$*\llbracket 1,3\rrbracket : 4$
1	2	4	$\mathcal{L}_2$	$\llbracket 1,1\rrbracket : \infty$	$\llbracket 1,2\rrbracket : 6$	$\llbracket 1,3\rrbracket : 5$
1	3	5	$\mathcal{L}_1$	$*\llbracket 1,4\rrbracket : 6$	$\llbracket 1,2\rrbracket : 6$	$\llbracket 1,3\rrbracket : 8$
1	3	6	$\mathcal{L}_2$	$\llbracket 1,4\rrbracket : 7$	$\llbracket 1,2\rrbracket : \infty$	$\llbracket 1,3\rrbracket : 8$
2	1	7	$\mathcal{L}_1$	$\llbracket 1,4\rrbracket : 7$	$*\llbracket 2,1\rrbracket : 10$	$*\llbracket 2,2\rrbracket : 8$
2	1	8	$\mathcal{L}_2$	$*\llbracket 2,0\rrbracket : \infty$	$\llbracket 2,1\rrbracket : 10$	$\llbracket 2,2\rrbracket : 9$
2	2	9	$\mathcal{L}_1$	$*\llbracket 2,3\rrbracket : 10$	$\llbracket 2,1\rrbracket : 10$	$\llbracket 2,2\rrbracket : 12$
2	2	10	$\mathcal{L}_2$	$\llbracket 2,3\rrbracket : 11$	$\llbracket 2,1\rrbracket : \infty$	$\llbracket 2,2\rrbracket : 12$
2	3	11	$\mathcal{L}_1$	$\llbracket 2,3\rrbracket : 14$	$*\llbracket 2,4\rrbracket : 12$	$\llbracket 2,2\rrbracket : 12$
2	3	12	$\mathcal{L}_2$	$\llbracket 2,3\rrbracket : 14$	$\llbracket 2,4\rrbracket : 13$	$\llbracket 2,2\rrbracket : \infty$

(b) The tiles in the buffer during the execution of the loop nest shown in (a), assuming that the buffer can accommodate up to three tiles at a time. $\llbracket x, y\rrbracket : r$ means that tile $X\llbracket x, y\rrbracket$ will reused at the intra-tile loop nest where $c = r$. The "$*$" indicates that the tile is decompressed.

Fig. 4.7 An example that demonstrates how our approach operates.

For intra-tile loop nest T_i, let us assume:

$$W_i(\vec{I}) = \{X_{k_1}[\![f_1^{(i)}(\vec{I})]\!], X_{k_2}[\![f_2^{(i)}(\vec{I})]\!], X_{k_j}[\![f_j^{(i)}(\vec{I})]\!]\}$$
$$R_i(\vec{I}) - W_i(\vec{I}) = \{X_{k_{j+1}}[\![f_{j+1}^{(i)}(\vec{I})]\!], X_{k_{j+2}}[\![f_{j+2}^{(i)}(\vec{I})]\!], X_{k_m}[\![f_m^{(i)}(\vec{I})]\!]\}.$$

For the tiles in the set $V_i(\vec{I}) = (R_i(\vec{I}) \cup W_i(\vec{I})) - U_i(\vec{I})$, i.e., the tiles of types (2) and (3), our compiler computes their reuse distances by executing the loop nest below (this loop nest is generated by the compiler using Omega library [3],[1] and it is executed only once at the compilation time):

$V = V_i(\vec{I}_0); \quad c = 0;$
for $\vec{I} = \vec{I}_0 + (0, 0, ..., 0, 1)^{\mathrm{T}}$ to $\vec{I}_N$ {
  $c = c + n$;
  for $i = 1$ to n {
    for each $t \in V \cap (R_j(\vec{I}) \cup W_j(\vec{I}))$ { $d_j(t) = c + j - 1$; }
    $V = V - (R_j(\vec{I}) \cup W_j(\vec{I}))$;
  }
}
for each $t \in V$ { $d_i(t) = \infty$; }

In the above compiler-generated code, $\vec{I}_0$ and $\vec{I}_N$ are two vectors such that $|\vec{I}_N - \vec{I}_0| = N$ and $\vec{L} \preceq \vec{I}_0 \prec \vec{I}_n \preceq \vec{U}$, where $|\vec{I}_N - \vec{I}_0|$ denotes the number of loop iterations between

[1]The Omega Library is a tool that provides functions for manipulating sets and relations that are defined using Presburger formulas. Presburger formulas are logical formulas that are built using affine expressions and universal/existential quantifiers.

$\vec{I}_0$ and $\vec{I}_N$, and $\vec{L}$ and $\vec{U}$ are, respectively, the lower and upper bound vectors for the target inter-tile loop nest for which we compute the reuse distances. $\vec{I}_0$ can be any vector between $\vec{L}$ and $\vec{U}$. In our experiments, we use $\vec{I}_0 = \vec{L}$. Note also that integer N is a threshold, and n is the number of intra-tile loop nests in the body of the inter-tile loop nest. The reuse distances larger than nN are treated as infinity (∞). Note that an inaccuracy in computing the reuse distance may lead to performance penalties when the program is executed; however, it does not cause any error in the program execution (i.e., it is not a correctness issue), since we are conservative in computing the set of of tiles that are used in each intra-tile loop nest.

4.2.4 Example

Figure 4.7(a) shows an inter-tile loop nest containing two intra-tile loop nests: $\mathcal{L}_1$ and $\mathcal{L}_2$. Intra-tile loop nest $\mathcal{L}_1$ uses tiles $X[\![i, j]\!]$ and $X[\![i, j+1]\!]$, with reuse distances $d_1 = 3$ and $d_2 = 1$, respectively. Intra-tile loop nest $\mathcal{L}_2$ uses tiles $X[\![i, j + 1]\!]$ and $X[\![i, j - 1]\!]$, with reuse distances $d_3 = 1$ and $d_4 = \infty$, respectively. This small code uses nine tiles throughout its execution. Figure 4.7(b) shows the state of the decompression buffer during the execution of the code fragment shown in Figure 4.7(a). In this figure, we observe that each tile is decompressed only once, which indicates that our compiler-directed buffer management approach is very effective in minimizing the number of decompressions.

4.2.5 Exploiting Extra Resources

While the compiler approach presented above schedules compressions and decompressions such that memory space consumption is reduced without excessively increasing

the original execution time, we can still incur performance penalties. This is because the decompression activities can occur on the critical path of execution and this in a sense symbolizes the tradeoff between memory savings and performance overheads due to data compression. In this section, we show how we can make use of extra resources available in our approach.

In a multiprocessor environment, we can reduce the performance overheads incurred by data compression by overlapping the execution of compression and decompression procedures with that of the computing loop nest. Specifically, for each inter-tile loop nest, our compiler generates two threads: the *computing thread* and the *buffer management thread*. In a multiprocessor based environment, these two threads can be executed in parallel. Figure 4.8 shows the code our compiler generates. Figure 4.8(b) gives the code for the buffer management thread. For each intra-tile iteration T_i at each inter-tile iteration $\vec{I}$, the buffer management thread decompresses each tile t in the set $R_i(\vec{I}) \cup W_i(\vec{I})$ if t is not in the buffer. The management thread increases $t.c$, the *reference counter* associated with tile t, by one for each $t \in R_i(\vec{I}) \cup W_i(\vec{I})$. The reference counter associated with each tile $t \in R_i(\vec{I}) \cup W_i(\vec{I})$ will be decreased by the computing thread after the execution of intra-tile T_i. A non-zero value in the reference counter $t.c$ indicates that tile t is being used or will be used by the computing thread, and consequently, t cannot be evicted from the buffer. On the other hand, when the buffer management thread needs to evict a tile from the buffer to make space for a new tile, it can evict any tile whose reference counter is zero (nevertheless, for better performance, as discussed in Section 4.2.3, we also require $v.r$ be maximized). After increasing the reference counter for each tile in $R_i(\vec{I}) \cup W_i(\vec{I})$, the management thread performs a V operation on a counting semaphore named "Iteration".

The value of this semaphore (Iteration.v) indicates the number of intra-tile loop nests that the computing thread can execute without being blocked. If the value of this semaphore is zero, the computing thread cannot continue with its execution due to the fact that some tiles required by the computing thread are not yet ready in the buffer. After the V operation on semaphore "Iteration", the management thread starts to decompress the tiles that will be used by the next intra-tile loop nest, without further synchronization with the computing thread.

Figure 4.8(c) gives the code (with necessary instructions inserted by our compiler) for the computing thread. Before executing each intra-tile loop nest T_i, the computing thread performs a P operation on the semaphore "Iteration". This P operation blocks the computing thread if some tiles that will be used by T_i are not ready in the buffer. In this case, the computing thread has to wait until the management thread decompresses all the required data tiles. After executing intra-tile loop nest T_i, the computing thread decreases the reference counter for each tile used by T_i. As discussed above, if the value of the reference counter of tile t is reduced to zero, we allow the buffer management thread to reuse the memory space occupied by t.

The management thread is blocked when it needs to decompress a new tile but the value of the reference counter for each tile in the buffer is greater than 0 (that is, none of the tiles have been used yet). In this case, the management thread has to wait for the computing thread to release some tiles by reducing their reference counters. If the computing thread is also blocked at the P operation on semaphore "Iteration", the system is deadlocked. Fortunately, this deadlock cannot happen as long as the following

condition is satisfied:

$$D \geq \max_{\forall i, \vec{I}} |R_i(\vec{I}) \cup W_i(\vec{I})|, \tag{4.1}$$

where D is size of the buffer (in terms of the number of tiles). Note that, if this condition is not satisfied, the single-threaded approach discussed in Section 4.2.3 cannot work properly, either. It is important to note that, in our approach, the condition expressed by 4.1 is always satisfied.

```
// definition of semaphores
Struct Semaphore {
  int v;
  ThreadQueue q;
}

P(s) {
  s.v = s.v − 1;
  if(s.v < 0)
    block in s.q;
}

V(s) {
  s.v = s.v + 1;
  if(s.v ≤ 0)
    unblock one thread in s.q;
}


// semaphores for synchronizing
// the computing and
// management threads
Iteration – counting semaphore,
  Iteration.v is initialized to 0;
Counters – binary semaphore,
  Counters.v is initialized to 1;
```

(a) Semaphores and their operations.

```
for Ī = L̄ to Ū {
  for i = 1 to n {
    c = c + 1;
    for each t ∈ R_i(Ī) ∪ W_i(Ī) {
      if(t ∉ B) {
        if(|B| = D) {
          repeat
            S = {v | v ∈ B ∧ v.c = 0}
          until S ≠ φ;
          select v ∈ S such that
            v.r is maximized
          if(v.w = 1) compress(v);
          evict(v); B = B − {v};
        }
        decompress(t);
        t.c = 0; B = B ∪ {v};
      }
      t.r = c + d_i(t);
      if(t ∈ W_i(Ī)) t.w = 1;
      P(Counters); t.c = t.c + 1;
      V(Counters);
    }
    V(Iteration);
  }
}
```

(b) The buffer management thread.

```
for Ī = L̄ to Ū {
  P(Iteration);
  T_1(R_1(Ī), W_1(Ī));
  P(Counters);
  for each t ∈ R_1(I) ∪ W_1(I)
    t.c = t.c − 1;
  V(Counters);
  ... ...
  ... ...
  P(Iteration);
  T_i(R_i(Ī), W_i(Ī));
  P(Counters);
  for each t ∈ R_i(Ī) ∪ W_i(Ī)
    t.c = t.c − 1;
  V(Counters);
  ... ...
  ... ...
  P(Iteration);
  T_n(R_n(Ī), W_n(Ī));
  P(Counters);
  for each t ∈ R_n(I) ∪ W_n(I)
    t.c = t.c − 1;
  V(Counters);
}
```

(c) The computing thread.

Fig. 4.8 The buffer management thread and the computing thread generated by our compiler.

Table 4.1 Major simulation parameters and their default values.

Configuration Parameter	Value
Functional Units	4 integer ALUs 1 integer multiplier/divider 4 FP ALUs 1 FP multiplier/divider
Issue Width	4
Commit Width	4
Cycle Time	1ns
Memory Space	1 MB

4.3 Experiments

4.3.1 Platform

We implemented our compression-based approach using the SUIF [156] compiler infrastructure. SUIF defines a small kernel and an intermediate representation, around which several independent optimization passes can be added. We implemented our approach as a separate pass. We observed during our experiments that the largest increase in original compilation times was about 65% (over the original case without our optimization). To gather performance statistics, we used the SimpleScalar [14] simulation environment and modeled an embedded architecture. The default values of the major simulation parameters used in our experiments are listed in Table 4.1. To make experiments with the two-processor case, we ran two copies of SimpleScalar in a coordinated fashion. The processors are assumed to share data in this case through a shared memory of 2MB. Note that our main goal is to reduce the memory space consumption of a given application (both MMO and AMO). We are not concerned in this thesis with the question of how to utilize the memory space saved through our compiler-based approach.

As our compression/decompression method, we used the LZO algorithm [2]. LZO is a data compression library which is suitable for data decompression in real-time. It is very fast in compression and extremely fast in decompression. The algorithm is both thread-safe and loseless. In addition, it supports overlapping compression and in-place decompression. We want to mention however that our compiler approach can work with any compression/decompression algorithm and is not tied to a particular one in any way.

For each benchmark code in our experimental suite, we performed experiments with five different versions, which can be summarized as follows:

- *BASE:* The base execution does not employ any data compression or decompression, but, just uses iteration space tiling. The specific tiling strategy used is due to Coleman and McKinley [42]. The memory saving and performance overhead results presented in the rest of this section are all normalized with respect to this base version.

- *LF:* This is similar to the *BASE* scheme in that it does not employ any data compression or decompression. The difference between *LF* and *BASE* is that the former uses a lifetime analysis at a data block level and reclaims the memory space occupied by dead data blocks, i.e., the block that will not be used during the rest of execution. Consequently, as compared to the *BASE* version, one can expect this scheme to reduce both MMO and AMO. This version implements an optimal strategy in recycling the dead blocks, i.e., the memory space allocated to a given data block is recycled into free space as soon as the data block is dead (i.e., completed its last reuse).

Table 4.2 Our benchmarks and their characteristics.

Benchmark	Functionality	MMO	AMO	Cycles (M)
facerec	face recognition	219.6	160.4	434.63
jacobi	Jacobi iteration	293.4	181.0	577.44
lu	LU decomposition	424.8	372.4	886.53
mpeg-2	MPEG-2 encoding	488.8	366.4	961.58
simi	pattern recognition	518.1	411.9	984.30
spec	spectral analysis	267.3	187.3	448.11
wave	wave equation solver	397.2	304.1	798.76
wibi	3D game program	793.6	527.7	1,488.19

- *AGG:* This is a compression/decompression based scheme that uses data compression very aggressively. Specifically, as soon as an access to a data block is completed, it is compressed. While one can expect this approach to reduce memory space consumption significantly, it can also incur significant performance penalties (since data compressions are performed not considering the data reuse patterns exhibited by the application, and consequently, the data blocks with high reuse can also be compressed frequently).

- *CD:* This is the compiler-directed scheme proposed herein. As discussed earlier in detail, it uses compression and decompression based on the data reuse information extracted from the program code by the automatic compiler analysis.

- *CD+LF:* This is a scheme that combines our compression based approach with dead block recycling. In principle, this version should generate the best memory occupancy savings.

The set of benchmark codes used in this study are given in Table 4.2. The second column of this table explains the functionality implemented by each benchmark. The next two columns show MMO and AMO, and the last column gives the execution cycles under

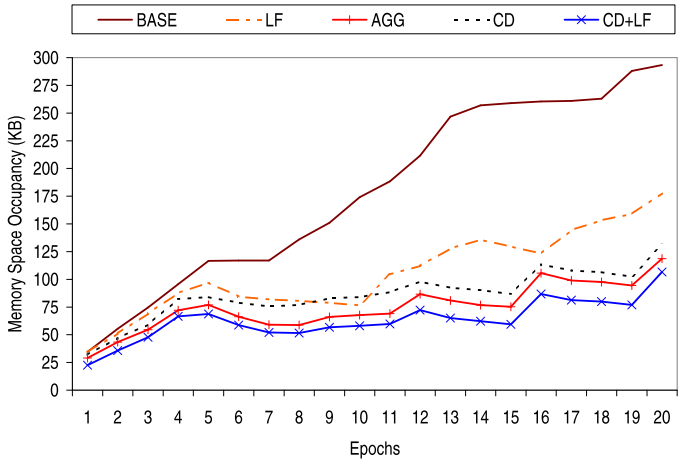


Fig. 4.9 Memory space occupancy for benchmark *jacobi*.

the base version (i.e., the one without any data compression/decompression or lifetime analysis). Recall that our goal is to reduce MMO and AMO as much as possible, while keeping the increase in the original execution cycles under control. As mentioned above, our results are presented as normalized values with respect to those obtained with the *BASE* version. In our experiments, all arrays are marked as compressible.

4.3.2 Results

Our first set of results are presented in Figure 4.9 for a representative benchmark: *jacobi*. This graph gives the memory space occupancy during execution for the five different versions tested. The execution time of each benchmark is divided into 20 epochs (each epoch has the same number of execution cycles). The value corresponding to an epoch is the maximum value seen in that epoch. One can make several important observations from this graph. First, as expected, the memory occupancy trend of the *BASE* version continuously increases. In comparison, the *LF* version has much better memory occupancy behavior. Note that, the sudden drops in its curve correspond to dead block

Table 4.3 MMO and AMO values with different schemes.

Benchmark	*LF*		*AGG*		*CD*		*CD+LF*	
	MMO	**AMO**	**MMO**	**AMO**	**MMO**	**AMO**	**MMO**	**AMO**
facerec	143.3	118.3	114.7	88.2	126.0	97.4	98.8	77.1
jacobi	177.7	105.4	118.7	74.8	131.3	86.0	106.7	63.4
lu	328.0	299.8	253.2	218.8	281.1	236.9	217.8	194.8
mpeg-2	367.4	281.4	307.6	229.9	340.5	254.1	281.5	205.4
simi	427.2	352.0	376.8	298.6	399.0	328.3	352.1	277.1
spec	151.3	114.5	126.3	83.2	143.6	98.1	109.4	71.6
wave	311.5	245.1	258.1	197.7	303.0	257.8	234.0	180.3
wibi	585.4	399.2	498.7	320.3	518.7	338.3	466.7	301.9

recycling. The *AGG* scheme also shows savings over the *BASE* version, due to its aggressive compression. When we look at the behavior of our approach (*CD*), we see that, while it is not as good as the *AGG* scheme, its results are competitive with those obtained using the *LF* scheme. In other words, data compression can be as effective as dead block recycling. Finally, we see that the best space savings are achieved with the *CD+LF* version since it combines the advantages of both data compression and dead block recycling. Table 4.3 gives the MMO and AMO values for all our benchmarks. We see that both *CD* and *CD+LF* schemes are able to reduce both MMO and AMO significantly.

While these results clearly show that our approach is able to reduce memory space occupancy significantly, the memory consumption is only one part of the whole picture. The other part is the impact on execution cycles. Recall from our earlier discussion in Section 4.2 that the performance overheads incurred by our approach depends largely on the underlying execution model. In our experiments, we consider the two execution models discussed in Section 4.2.[2] Figure 4.10 presents the percentage increase in execution

[2] A potential third option, which uses specialized hardware circuit devoted to compression/decompression is not evaluated in this section. However, we believe that most of our compiler analysis used for the multiprocessor case (Section 4.2.5) can be reused for handling the specialized hardware case as well.

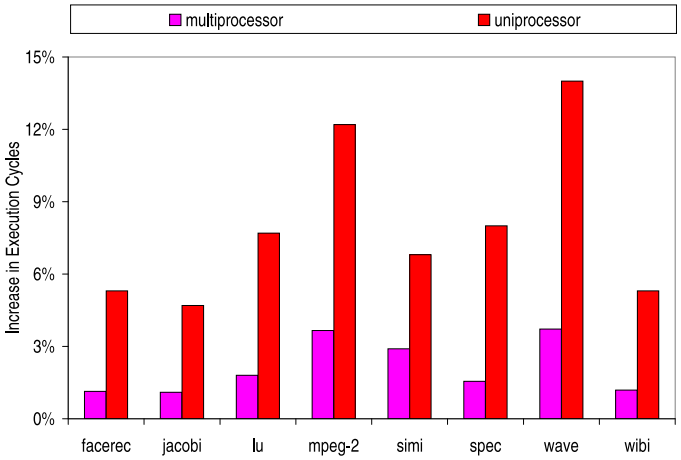


Fig. 4.10 Increase in execution cycles.

cycles for our approach (*CD*) with respect to the *BASE* scheme. The results are given for both single processor and multiple processor cases. As can be seen from these results, the average degradation in performance in the cases of uniprocessor and multiprocessor is 15.7% and 3.3%, respectively. The largest increases occur with the *mpeg-2* and *wave* benchmarks since they exhibit the lowest data reuse among our benchmarks. Since our approach exploits data reuse information in minimizing the performance overheads due to decompressions, it is less successful with these two codes. The performance degradations with the remaining benchmarks on the other hand are not very large.

Chapter 5

Integrating Loop and Data Optimizations for Locality within a Constraint Network Based Framework

The main data locality problem in the memory hierarchy context is to maximize the number of the data accesses satisfied from the higher levels in the memory hierarchy. On the compiler side, there have been two different trends in optimizing memory behavior. One of these is called code restructuring and usually takes form of loop optimizations. These optimizations process a single loop nest at a time (or sometimes a series of consecutive loop nests as in the case of loop fusioning) and try to change data access pattern of the loop (by re-organizing loop iterations) so that it becomes suitable for the memory layout of the data structures manipulated by the loop. The main advantage of loop transformations is that they are well understood and backed up by a comprehensive theory, and robust compiler techniques have been developed in both commercial world and academia. In addition, a loop transformation usually operates on a single loop nest at a time and thus it can be customized based on the needs of the current loop being targeted. Its main drawback is that the effectiveness of a loop transformation is limited by intrinsic data dependencies in the loop nest. The second trend in optimizing data-intensive applications for locality is data transformations. A data transformation typically changes the memory layout of data in an attempt to make it suitable for a given loop structure (data access pattern). An example would be converting a row-major memory layout to a column-major one, if doing so improves data locality. Data transformations have been

shown to be effective in cases where loop transformations are not applicable due to data dependencies.

Based on the discussion above, loop and data optimizations are complementary. Consequently, in principle, the best memory behavior can be obtained by a combination of loop and data transformations. Indeed, as will be discussed next, several research projects have already designed and implemented such integrated locality optimizers that apply some sort of combination of loop and data transformations. However, there is an inherent difficulty in optimizing both data layout and loop access pattern simultaneously under a unified setting. This difficulty occurs due to the fact that a given data structure (e.g., a multi-dimensional array) can be accessed by different loop nests of the application, and each such loop nest can demand a different transformation for the said array. This results in a *coupling problem*, where the behaviors of two (or more) different nests are coupled to each other as a result of data sharing between them (e.g., they access the same array). In a linear algebraic formulation, such a coupling presents a nonlinearity, which means that the resulting problem is very difficult to solve.

The main goal of this chapter is to explore the possibility of employing *constraint network theory* for solving the integrated data-loop optimization problem. The reason that we believe a constraint network (CN) based solution can be a promising one for our locality optimization problem is the fact that CN captures problems that involve multiple constraints that operate on the same data structure (domain) and this is exactly what we need for our memory optimization problem. The main contributions of this chapter can be summarized as follows:

- We present a CN based formulation of the integrated loop-data optimization problem. As will be shown, one of the nice properties of our CN formulation is that it can easily be modified to exclude loop or data transformations, if it is desired.

- We present two alternate solutions to the problem with our CN formulation and discuss the pros and cons of each scheme. The first solution is a pure backtracking based one, whereas the second one improves upon the first one by employing three additional optimizations, including backjumping.

- We give experimental evidence showing the success of the proposed approach, and compare our results with those obtained through one of the previously-proposed integrated approaches to data locality.

```
for j = 1 to N
  for i = 1 to N {
    X(i,j) = X(i-1,j+2) + Y(i,j)
  }
for i = 1 to N
  for j = 1 to N {
    Z(j,i) = X(i,j) + 1.0
  }
```

Fig. 5.1 An example code fragment.

5.1 Data Layout and Loop Transformation Abstractions

In this section, we present the loop (computation) and data space abstractions employed by our CN based approach. Our focus is on affine programs, which are built from a series of nested loops manipulating multi-dimensional arrays. Notice that many programs

from the domain of embedded image/video processing fall into this category [29]. In these programs, array subscript functions and loop bounds are affine functions of enclosing loop indices.

Consider such an access to an m-dimensional array in an n-deep loop nest. Let $\bar{I}$ denote the iteration vector (consisting of loop indices starting from the outermost loop). Each array reference can be represented as $\mathcal{S}\bar{I}+\bar{o}$, where the $m \times n$ matrix $\mathcal{S}$ is called the access (or reference) matrix [157] and the m-element vector $\bar{o}$ is called an offset vector. In order to illustrate the concept, let us consider the first loop nest shown in Figure 5.1 and the array reference $X(i-1, j+2)$ in it. For this reference, we have

$$\bar{I} = \begin{bmatrix} j \\ i \end{bmatrix}, \ \mathcal{S} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \text{ and } \bar{o} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}.$$

An element is said to be reused if it is accessed (read or written) more than once in a loop nest. There are two types of reuses: temporal and spatial. Temporal reuse occurs when two references (not necessarily distinct) access the same memory location; spatial reuse arises between two references that access nearby memory locations. The notion of nearby locations is a function of the memory layout of elements. It is important to note that the most important reuses (whether temporal or spatial) are the ones exhibited by the innermost loop. If the innermost loop exhibits temporal reuse for a reference, then the data element accessed by that reference can be kept in a register throughout the execution of the innermost loop (assuming that there is no aliasing). Similarly, spatial reuse is most beneficial when it occurs in the innermost loop because, in that case, it may

enable unit-stride accesses to consecutive locations in memory. The purpose of loop and data transformations, in the context of data locality optimization, is to transform the code structure and/or data layout such that most of the data reuses (after the transformation) take place in the innermost loop position.

We use linear matrices $\mathcal{V}$ and $\mathcal{W}$ to represent loop and data transformations, respectively. For an m-dimensional array accessed within an n-deep loop nest, $\mathcal{V}$ is n-by-n and $\mathcal{W}$ is m-by-m. A reference to such an array, represented by $\mathcal{S}\bar{I} + \bar{o}$, is rewritten as $\mathcal{W}\mathcal{S}\mathcal{V}^{-1}\bar{I'} + \mathcal{W}\bar{o}$ after the loop and data transformations are applied [39]. In this new expression, $\bar{I'}$ represents the iteration vector of the transformed loop nest, i.e., we have $\bar{I'} = \mathcal{V}\bar{I}$. Our optimization goal can then be defined as one of determining a suitable $\mathcal{W}$ for each array and a suitable $\mathcal{V}$ for each loop nest so that $\mathcal{W}\mathcal{S}\mathcal{V}^{-1}\bar{I'} + \mathcal{W}\bar{o}$ is in a *desired form* for as many references as possible. There are two types of desired forms. In the first one, the last column of $\mathcal{W}\mathcal{S}\mathcal{V}^{-1}$ is a zero vector (i.e., all its entries are zero), and this corresponds to having temporal reuse in the innermost loop (after the transformation). The second form corresponds to having spatial reuse in the innermost loop and requires that the last column of $\mathcal{W}\mathcal{S}\mathcal{V}^{-1}$ is $[0 \quad 0 \quad 0 \cdots 0 \quad 1]^T$. Since each array reference can be optimized either for temporal and spatial reuse in the innermost loop position, we have 2^p potential optimization alternatives for p references to the array. Clearly, some of these alternatives are not legal, i.e., the loop transformations they suggest lead to the violation of one or more data dependencies; thus, they cannot be applied if we are to preserve the original code semantics. As an example, we consider again the first loop nest of the code fragment in Figure 5.1. In this example, the dependence vector in the first loop nest is $[2 \quad -1]^T$. As a result, interchanging the order of the two loops in this nest by applying

a loop transformation such as

$$\mathcal{V} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

would result in the new dependence vector $[-1 \quad 2]^T$, which is not acceptable since it is not lexicographically positive. This means that loop interchange transformation is not legal for this loop nest.

One of the problems that make it tough to determine suitable $\mathcal{V}$ and $\mathcal{W}$ matrices is the *coupling between loop nests due to accessing common arrays.* Since dynamic array layout transformation at runtime is a costly operation, we need to determine a single memory layout (which means a single $\mathcal{W}$ matrix) for each array in the application code. Since a given array can be accessed by multiple nests in the application, we need to capture the constraints arising from all these accesses. This coupling issue makes the data locality optimization problem very hard [39] and forces designers to develop heuristics. For example, let us consider the entire code fragment in Figure 5.1. Considering the accesses to array X, we see that, the first loop nest prefers a column-major memory layout for this array, corresponding to the data transformation matrix[1]

$$\mathcal{W} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

[1]We assume that the default memory layout of multi-dimensional arrays is row-major.

In contrast, if the second loop nest prefers a row-major memory layout for the same array, which means no data transformation, i.e., we have:

$$\mathcal{W} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Even this small example clearly illustrates that choosing the right layout transformation for the arrays and the accompanying loop transformations for the nest (which means satisfying multiple constraints) is not an easy task when one considers multiple loop nests simultaneously. As will be discussed in detail in the next section, a constraint network precisely captures these multiple constraints, and consequently, one can make use of known solution strategies (on constraint networks) to search for acceptable solutions.

5.2 CN Based Formulation and Solution

5.2.1 CN Background

A constraint network (CN) can be described as a triple $CN =< \mathcal{P}, \mathcal{Q}, \mathcal{R} >$, where $\mathcal{P}$ is a finite set of variables, $\mathcal{Q}$ is a list of possible values for each variable, and each member of $\mathcal{R}$ is a constraint, which is defined on a subset of variables $\mathcal{T}_i \subseteq \mathcal{P}$, and it denotes the variables' simultaneous legal value assignments [52]. The arity of the constraint refers to the number of variables it contains. The most common constraints are binary constrains, i.e., those defined over two variables.

A *solution* of a constraint network $CN = < \mathcal{P}, \mathcal{Q}, \mathcal{R} >$, where $\mathcal{P} = \{p_1, p_2, \cdots, p_s\}$, is a *valid* (*consistent*) *instantiation* of all its variables from domain $\mathcal{Q} = \{q_1, q_2, \cdots, q_b\}$

that satisfies all the constraints indicated by $\mathcal{R}$. An instantiation in this definition corresponds to an assignment of values to variables. It can be complete or partial depending on whether all the variables have been assigned values or not. A valid instantiation is an instantiation that satisfies all constraints that are defined using the variables involved. Therefore, we can define a solution as the process of finding a complete assignment of values (from set $\mathcal{Q}$) to variables (from set $\mathcal{P}$) such that all constraints in $\mathcal{R}$ are satisfied. Note that such a solution may or may not exist. If it does not exist, the usual approach is to relax the network by dropping one or more constraints from $\mathcal{R}$ and retrying a solution.

5.2.2 Problem Formulation for Data Locality

We now define our constraint network

$$CN = \{< \mathcal{P}_v, \mathcal{P}_w >, < \mathcal{Q}_v, \mathcal{Q}_w >, \mathcal{R}\}$$

that captures the data locality constraints derived by the compiler. In our customized CN, set $\mathcal{P}_v$ contains the loop transformations for all loop nests and set $\mathcal{P}_w$ contains data (layout) transformations for all the arrays that appear in the code fragment being optimized. That is, $\mathcal{P}_v$ = $\{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3, \cdots, \mathcal{V}_v\}$, and $\mathcal{P}_w$ = $\{\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3, \cdots, \mathcal{W}_w\}$, where v and w correspond to the number of nests and the number of arrays, respectively. Note that these are unknowns we want to determine; i.e., at the end of our optimization process, we want to determine a loop transformation for each loop nest in the program and a data transformation for each array in the program.

Sets $\mathcal{Q}_v$ and $\mathcal{Q}_w$ capture the domains for all variables in sets $\mathcal{P}_v$ and $\mathcal{P}_w$, respectively. Specifically, for a loop variable $\mathcal{V}_i$, it is the possible loop transformations that can be assumed by this variable so that data locality can be optimized. And, similarly, for a layout variable $\mathcal{W}_j$, it is the set of possible data transformations that can be used for enhancing data locality. For efficiency reasons, we may want to limit the number of entries in sets $\mathcal{Q}_v$ and $\mathcal{Q}_w$.[2] In mathematical terms, for a $\mathcal{V}_i$, we have $\mathcal{D}_{V_i} = \{T_1, T_2, \cdots, T_{f_i}\}$, where f_i is the number of possible values for $\mathcal{V}_i$; i.e., $\mathcal{D}_{V_i}$ is the domain for $\mathcal{V}_i$. Similarly, for a $\mathcal{W}_j$, we have $\mathcal{D}_{W_j} = \{M_1, M_2, \cdots, M_{g_j}\}$, where g_j is the number of possible values (data layouts) for $\mathcal{W}_j$. Note that, we can now express the $\mathcal{Q}_v$ and $\mathcal{Q}_w$ sets as follows:

$$\begin{aligned} \mathcal{Q}_v &= \{\mathcal{D}_{V_1}, \mathcal{D}_{V_2}, \mathcal{D}_{V_3}, \cdots, \mathcal{D}_{V_v}\} \quad \text{and} \\ \mathcal{Q}_w &= \{\mathcal{D}_{W_1}, \mathcal{D}_{W_2}, \mathcal{D}_{W_3}, \cdots, \mathcal{D}_{W_w}\}, \end{aligned}$$

each $\mathcal{D}_{V_i}$ and $\mathcal{D}_{W_j}$ itself is a set, as explained above. The set $\mathcal{R}$ captures the constraints extracted by the compiler from each and every nest. Each member $\mathcal{R}_i$, which captures the possible locality constraints associated with the ith nest in the code fragment being optimized, is composed of $\mathcal{R}_{i,k}$ sets of the following form:

$$\mathcal{R}_{i,k} = \{T_k, M_{k_1}, M_{k_2}, M_{k_3}, \cdots, M_{k_w}\},$$

[2] For example, one can restrict the matrix entries to 0, 1 and -1.

where $T_k \in \mathcal{D}_{V_i}$ and $M_{k_j} \in \mathcal{D}_{W_j}$. If an array is not accessed by a particular loop nest i, its corresponding entry is set to ϵ in all $\mathcal{R}_{i,k}$ sets of $\mathcal{R}_i$. Informally, each such constraint captures a condition for data locality, and indicates that in order to have good data locality (i.e., for each array reference to have temporal or spatial reuse in the innermost loop position), the loop nest can be transformed using transformation matrix T_k and the arrays accessed by it should have the layouts captured by the data transformation matrices $M_{k_1}, M_{k_2}, M_{k_3}, \cdots, M_{k_w}$, each corresponding to one array accessed by the loop nest in question. Note that, for each nest i, we can have multiple constraints (listed in $\mathcal{R}_i$), and a solution has to satisfy only a single $\mathcal{R}_{i,k}$ for that nest. That is, we can write $\mathcal{R}$ as:

$$
\begin{aligned}
\mathcal{R} \quad = \{ & \\
& \{\mathcal{R}_{1,1}, \mathcal{R}_{1,2}, \mathcal{R}_{1,3}, \cdots\}, \\
& \{\mathcal{R}_{2,1}, \mathcal{R}_{2,2}, \mathcal{R}_{2,3}, \cdots\}, \\
& \cdots \\
& \{\mathcal{R}_{v,1}, \mathcal{R}_{v,2}, \mathcal{R}_{v,3}, \cdots\} \\
\}, &
\end{aligned}
$$

where $\mathcal{R}_i = \{\mathcal{R}_{i,1}, \mathcal{R}_{i,2}, \mathcal{R}_{i,3}, \cdots\}$. Our task then is to set the values of the contents of the sets $\mathcal{P}_v$ and $\mathcal{P}_w$ such that we satisfy a $\mathcal{R}_{i,k}$ constraint from each $\mathcal{R}_i$; that is, the selected layouts and loop transformations go very well in all nests. The search procedures on constraint networks help us achieve this (if it is achievable).

We now give an example to illustrate how this formulation is carried out in practice. We consider the code fragment in Figure 5.2. In this code fragment, we have three

```
for i = 1 to N
  for j = 1 to N {
    X(i,j) = Z(j,i) + 2.0
  }
for i = 1 to N
  for j = 1 to N {
    Y(i,j) = Z(i,j) * Z(i,j)
  }
for i = 1 to N
  for j = 1 to N {
    a = a + X(j,i)/3.0
  }
```

Fig. 5.2 An example code fragment.

different loop nests accessing three different arrays. Let us restrict ourselves to only loop interchange as the only possible loop transformation and dimension re-indexing (i.e., transforming a memory layout from row-major to column-major) as the only possible data transformation. We have:

$$CN = \{< \mathcal{P}_v, \mathcal{P}_w >, < \mathcal{Q}_v, \mathcal{Q}_w >, \mathcal{R}\},$$

where $\mathcal{P}_v$ = $\{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3\}$, $\mathcal{P}_w$ = $\{\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3\}$, $\mathcal{Q}_v$ = $\{\mathcal{D}_{V_1}, \mathcal{D}_{V_2}, \mathcal{D}_{V_3}\}$, and $\mathcal{Q}_w$ = $\{\mathcal{D}_{W_1}, \mathcal{D}_{W_2}, \mathcal{D}_{W_3}\}$. Here, $\mathcal{V}_1$, $\mathcal{V}_2$, and $\mathcal{V}_3$ are the loop transformations we want to determine for the first, second, and third loop nest, respectively, and $\mathcal{W}_1$, $\mathcal{W}_2$, and $\mathcal{W}_3$ are the data transformations we want to determine for arrays X, Y, and Z, respectively. Also, $\mathcal{Q}_v$ and $\mathcal{Q}_w$ capture our loop transformation and data transformation domains, respectively, and can be expressed as follows:

$$\mathcal{Q}_v = \left\{ \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} ; \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} ; \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} \right\}$$

and

$$\mathcal{Q}_w = \left\{ \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} ; \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} ; \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} \right\}.$$

Note that a $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ matrix in $\mathcal{Q}_v$ corresponds to no loop transformation case, whereas a $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ matrix represents loop interchange. Similarly, a $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ matrix in $\mathcal{Q}_w$ corresponds to no memory layout optimization case, while a $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ matrix in the same set indicates dimension re-indexing (converting the memory layout from row-major to column-major). Note that these domains restrict the potential loop/data transformations to be used in optimizing data locality. It is easy to see that, in this example, we have $\mathcal{D}_{V_1} = \mathcal{D}_{V_2} = \mathcal{D}_{V_3}$ and $\mathcal{D}_{W_1} = \mathcal{D}_{W_2} = \mathcal{D}_{W_3}$.

Let us now focus on the set $\mathcal{R}$, which captures the necessary constraints for data locality. We have $\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3\}$, where $\mathcal{R}_i$ captures the possible locality constraints for the ith loop nest. More specifically, we have:

Our task is to select a constraint from each $\mathcal{R}_i$ – to satisfy – such that there is no conflict (as far as the layouts of the arrays are concerned) among the constraints selected. In the next section, we discuss the search algorithms we implemented within our CN based framework.

$$\mathcal{R}_1 = \{\{\begin{bmatrix}1&0\\0&1\end{bmatrix}, \begin{bmatrix}1&0\\0&1\end{bmatrix}, \epsilon, \begin{bmatrix}0&1\\1&0\end{bmatrix}\}, \{\begin{bmatrix}0&1\\1&0\end{bmatrix}, \begin{bmatrix}0&1\\1&0\end{bmatrix}, \epsilon, \begin{bmatrix}1&0\\0&1\end{bmatrix}\}\}$$

$$\mathcal{R}_2 = \{\{\begin{bmatrix}1&0\\0&1\end{bmatrix}, \epsilon, \begin{bmatrix}1&0\\0&1\end{bmatrix}, \begin{bmatrix}1&0\\0&1\end{bmatrix}\}, \{\begin{bmatrix}0&1\\1&0\end{bmatrix}, \epsilon, \begin{bmatrix}0&1\\1&0\end{bmatrix}, \begin{bmatrix}0&1\\1&0\end{bmatrix}\}\}$$

$$\mathcal{R}_3 = \{\{\begin{bmatrix}1&0\\0&1\end{bmatrix}, \begin{bmatrix}0&1\\1&0\end{bmatrix}, \epsilon, \epsilon\}, \{\begin{bmatrix}0&1\\1&0\end{bmatrix}, \begin{bmatrix}1&0\\0&1\end{bmatrix}, \epsilon, \epsilon\}\}$$

5.2.3 Solutions

In this section, we discuss how the constraint network described in the previous section can be solved. For many constraint network problems, search is epitomized by the backtracking algorithm [52]. A backtracking based search algorithm extends current partial solution by assigning values (from their domains) to new (unassigned) variables. When the algorithm reaches a point where there exist no further assignment for the variable under consideration that is consistent with all the values assigned to other variables so far, it backtracks. In backtracking, if possible, the value assigned to the variable immediately preceding the dead-end variable is changed and the search continues. Therefore, there are two possibilities for termination. The first one occurs when we assign a value to every variable and all the constraints in the network are satisfied, i.e., we reach a complete valid assignment. The second one occurs, on the other hand, when it can safely be concluded that no solution exists. While a backtracking algorithm is exponential as far as its asymptotic complexity is concerned, for the medium size constraint networks (such as the ones we are considering in our locality problem), they generate results quickly (as will be discussed later when we present our experimental results). In the rest of this discussion, we refer to this algorithm as the *simple backtracking algorithm.*

Before summarizing its operation, let us make an important definition: *state space.* A state space can be described as a quadruple (S, s_0, O, S_g) [52]. Here, S is a set of states, s_0 is the initial state, O is a set of operators that map states to states, and $S_g \subseteq S$ is the set of final states (also called the solution states). Note that a state space can be represented as a *search graph* (also called *state graph*). Now, any search algorithm on constraint networks can be expressed as traversing its corresponding state graph in some order. In this search graph, the nodes represent valid (consistent) partial instantiations and the edges capture operators that take a partial instantiation as an input and generates an augmented partial instantiation by instantiating a new variable (with a value from its domain). Clearly, the initial state is an empty instantiation, i.e., no variable is instantiated, whereas a solution state represents a complete valid instantiation (obviously, we can have multiple solution states).

A backtracking algorithm simply traverses the state space of partial instantiations in a depth-first manner (on the corresponding search graph). In its first phase, it selects a variable at a time and instantiates it using a value from its domain. This phase is termed as the *forward phase*. In the second phase, when no valid solution exists for the current variable under consideration, the algorithm backtracks. This phase is referred to as the *backward phase.*[3]

It needs to be noticed that both forward and backward phases of the backtracking algorithm can be improved. To improve the forward phase, we need to be careful in (i)

[3]Under certain cases, it is possible to restrict the search space and generate a backtrack-free search space. Such transformation techniques usually make the underlying constraint network arc-consistent and/or path-consistent; but, they are beyond the scope of this work. Note that if the network is transformed in this fashion, we do not have the backward phase.

selecting the new variable to instantiate and (ii) selecting the value to instantiate the variable with. In our constraint network, (i) corresponds to carefully selecting the next loop nest or array variable to consider (actually, their transformations), and (ii) corresponds to carefully selecting the value (transformation) for the variable chosen. As for (i), we select a variable such that its instantiation maximally constrains the rest of the search space. In practice, this means selecting the viable with the smallest number of viable (loop or data) transformations. On the other hand, for (ii), we select a (loop or data) transformation that maximizes the number of options available for future instantiations. It is also possible to improve the performance of the backward phase of the algorithm. Specifically, in some cases, it can be evident that backtracking to the immediately previous variable (and re-instantiating it with a different value) will not lead to a solution. As an example, let us consider the following simple scenario. Suppose that, in the previous step during the optimization process, we selected the layout M_s for variable $\mathcal{W}_i$, and in the current step we selected the layout $M_{s'}$ for variable $\mathcal{W}_j$. If, at this point, we see that there cannot be any solution based on these partial assignments, the simple backtracking algorithm returns to variable $\mathcal{W}_i$ and assigns a new layout (say $M_{s''}$) to it (assuming that we tried all alternatives for $\mathcal{W}_j$). However, it must be noted that, if there is no constraint in the network in which both $\mathcal{W}_i$ and $\mathcal{W}_j$ appear together, assigning a new value to $\mathcal{W}_i$ would not generate a solution, since $\mathcal{W}_i$ is not the culprit for reaching the dead-end. Instead, backjumping skips $\mathcal{W}_i$ and determines an array (say $\mathcal{W}_k$) among the arrays that have already been instantiated that co-appears with $\mathcal{W}_j$ in a constraint, and assigns a new value (layout) to it (i.e., different from its current value). In this way, backjumping can prevent

useless assignments and, as a result, expedite our search. Figure 5.3 gives an illustration that compares backtracking and backjumping.

In our experiments (results of which will be presented later), we explicitly quantify the benefits coming from these improvements. Note that these improvements rarely affect the quality of the solution returned by the backtracking algorithm. If a solution exists to the problem under consideration, both the base and improved schemes return the same solution. However, if multiple solutions exist, they can potentially find different solutions, though it was not the case in our experiments. In most cases, we can expect to improve (reduce) solution times, i.e., the time it takes to find a complete consistent instantiation.

In our running example discussed earlier in Section 5.2.2, our approach returns the following loop and data transformations as one possible solution to the underlying constraint network (whether we use backtracking or the improved search): The loop transformations for the first, second, and third loop nests are $\mathcal{V}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\mathcal{V}_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, and $\mathcal{V}_3 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, respectively; and the data transformations for arrays X, Y, and Z are $\mathcal{W}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\mathcal{W}_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, and $\mathcal{W}_3 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, respectively. What these transformations indicate that, while the first loop nest is not transformed, the second and third loop nests are transformed using loop interchange; and the layouts of arrays Y and Z are transformed, whereas the layout of array X is not modified. One can easily verify that these layouts and transformations go very well as far as optimizing data locality is concerned.

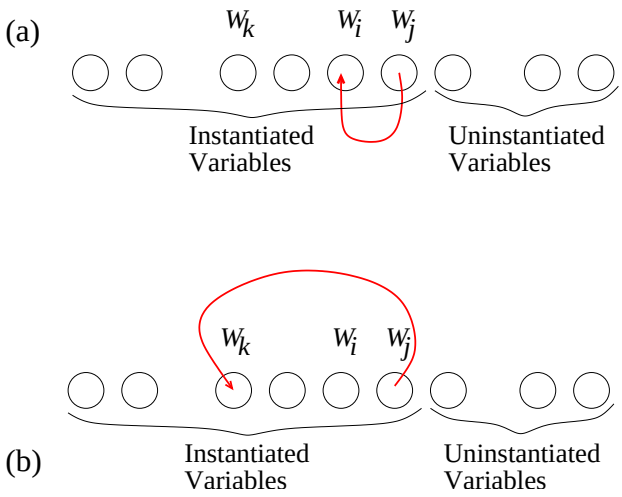


Fig. 5.3 (a) Backtracking. (b) Backjumping.

5.2.4 Discussion

We now discuss how our CN based formulation can be modified if we are to restrict our approach to only determining data layouts or to only determining loop transformations. These simplified problem formulations may be necessary in some cases. For example, when we know that the loop structures cannot be modified in the application code being handled (e.g., as a result of explicit loop parallelization which could be distorted badly by a subsequent loop transformation), we might want to determine only memory layouts. Similarly, in some cases it may not be possible to work with any layouts other than the conventional ones. Then, loop transformations are the only option to improve data locality. Actually, both these cases are easy to handle within our CN based data locality optimization framework. Basically, what we do is to drop the corresponding locality constrains from consideration. For example, if we consider applying layout optimizations only, we can redefine our network as $CN = \{< \mathcal{P}_w >, < \mathcal{Q}_w >, \mathcal{R}\}$, instead

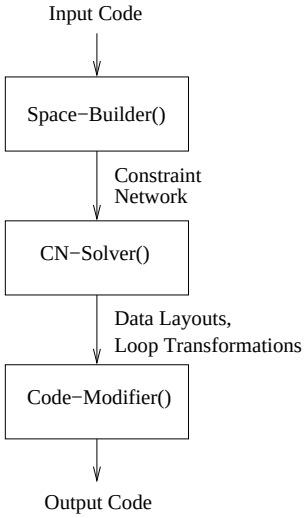


Fig. 5.4 Building blocks of our implementation.

of $CN = \{< \mathcal{P}_v, \mathcal{P}_w >, < \mathcal{Q}_v, \mathcal{Q}_w >, \mathcal{R}\}$, as we did earlier. Also, in this case, $\mathcal{R}$ holds only the memory layout constraints.

5.3 Experiments

There are two goals of our experimental analysis. First, we would like to see how much improvement our approach brings over the unoptimized (original) codes. Second, we want to compare our approach to previously-proposed locality optimization schemes.

5.3.1 Setup

We made experiments with seven different versions of each code in our benchmark suite, which can be summarized as follows:

- Base: This is the base version against which we compare all the optimized schemes. This version does not apply any data locality optimization to the program code.

Table 5.1 Benchmark codes.

Benchmark Name	Benchmark Description	Data Size (KB)	Domain Size	Sol Space Size	Compilation Time (ms)	Execution Time (ms)
encr	digital signature for security	1,137.66	324	7212	34.41	434.63
wood	color-based surface inspection	2,793.81	851	17638	104.33	961.58
SP	all nodes shortest path	2,407.38	688	14096	96.76	984.30
srec	speech recognition	974.19	752	16223	26.05	448.11
jpegview	JPEG image display	1,628.54	990	20873	82.34	798.76

- Loop-Heur: This represents a previously-proposed loop optimization scheme, and is adapted from the thesis of Li [88]. The approach uses both linear loop transformations (interchange, reversal, skewing, and scaling) and iteration space tiling for maximizing temporal and spatial reuse in the innermost loop positions.

- Data-Heur: This version is adapted from Leung and Zahorjan [85]. It transforms memory layout of each multi-dimensional array to maximize locality. It might be applicable in cases where loop transformations are not, due to data dependencies. However, since it is a pure memory layout transformation based scheme, it cannot optimize for temporal locality.

- Integ-Heur: This represents an approach that combines loop and data transformations under a unified optimizer. It is based on the algorithm described in [39]. It formulates the locality problem as one of finding a solution to a non-linear algebraic system and solves it by considering each loop nest one-by-one, starting with the most expensive loop nest (in terms of execution cycles).

- Loop-CN: This is the loop transform-only version of our CN based approach, as explained in Section 5.2.4.

- Data-CN: This is the data transform-only version of our CN based approach, as explained in Section 5.2.4.

- Integ-CN: This is the integrated optimization approach proposed in this section. It combines loop and data transformations under the CN based optimization framework.

Unless stated otherwise, Integ-CN does not employ the improvements discussed in Section 5.2.3. That is, the Integ-CN version corresponds to the simple backtracking algorithm discussed earlier. These optimizations, which target at both forward and backward phases, will be evaluated separately later.

Figure 5.4 depicts the implementation of our CN based approach. Note that this implementation includes all three versions, namely, Loop-CN, Data-CN, and Integ-CN. The input code is first analyzed by the compiler and possible data layouts and loop transformations that go well with them are identified on a loop nest basis (note that these form the entries of our constraint set $\mathcal{R}$). The module that implements this functionality is called Space-Builder() since it builds the solution space that is explored by our search algorithms. This solution space information is subsequently fed to the CN-Solver(), which determines the desired memory layouts and loop transformations (if it is possible). This layout and loop transformation information is then fed to the compiler which implements the necessary code modifications and data re-mappings. The compiler parts of this picture (i.e., Space-Builder() and Code-Modifier()) are implemented using the SUIF compiler [156]. The CN solver we wrote consists of approximately 1700 lines C++ code. In our

Table 5.2 Normalized solution times with different schemes. An "x" value indicates "x times of the base version".

Benchmark Name	Solution Times						
	Loop-Heur	**Data-Heur**	**Integ-Heur**	**Loop-CN**	**Data-CN**	**Integ-CN**	**Integ-CN***
encr	1.21	1.34	2.28	3.83	4.17	5.12	3.09
wood	1.36	1.37	2.54	3.46	3.77	4.98	3.21
SP	1.10	1.28	2.07	4.08	4.90	6.22	2.83
srec	1.47	1.89	2.86	3.61	4.15	5.46	4.11
jpegview	1.28	1.42	2.15	2.38	2.92	3.87	2.94

experiments, we restricted the entries of all loop and data transformation matrices to 1, -1 and 0.

Our experiments have been performed using the SimpleScalar infrastructure [14]. Specifically, we modeled an embedded processor that can issue and execute four instructions in parallel. The machine configuration we use includes separate L1 instruction and data caches; each is 16KB, 2-way set-associative with a line size of 32 bytes. The L1 cache and main memory latencies are 2 and 100 cycles, respectively. Unless stated otherwise, all the experiments discussed in this section use this machine configuration.

Table 5.1 lists the benchmark codes used in our experimental evaluation. The second column of this table gives a description of the benchmark, and the next column gives the total data size (in KBs) manipulated by each benchmark. The fourth column gives the size of the domain for each benchmark and the fifth column shows the size of the solution space (i.e., the total number of constraints in the $\mathcal{R}$ set). The sixth column gives the compilation times for the base version described above, and the seventh column gives the execution times, again under the base version.

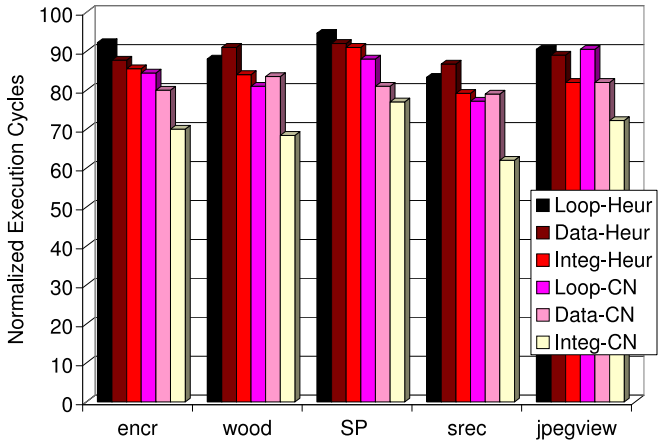


Fig. 5.5 Normalized execution cycles with different schemes.

5.3.2 Results

The solution times of the different optimized schemes, normalized with respect to those taken by the base version, are given in Table 5.2. These times are collected on a 500MHz Sun Sparc architecture. Note that the solution times given for our CN based approaches include the total time spent in the three components shown in Figure 5.4, and are normalized with respect to the sixth column of Table 5.1. Let us focus on the columns between two and seven (the last column will be discussed later). At least two observations can be made from these columns. First, as expected, our CN based schemes take longer times to reach a solution compared to the previously-proposed heuristics. Second, among our schemes, Integ-CN takes the largest time since it determines both loop and data transformations. It should be noted however that since compilation is essentially an offline process and code quality is a strong requirement in embedded computing, our belief is that these solution times are within tolerable limits.

The bar-chart in Figure 5.5 gives the execution times. Each bar is normalized with respect to the corresponding value of the base scheme (see the last column of Table 5.1).

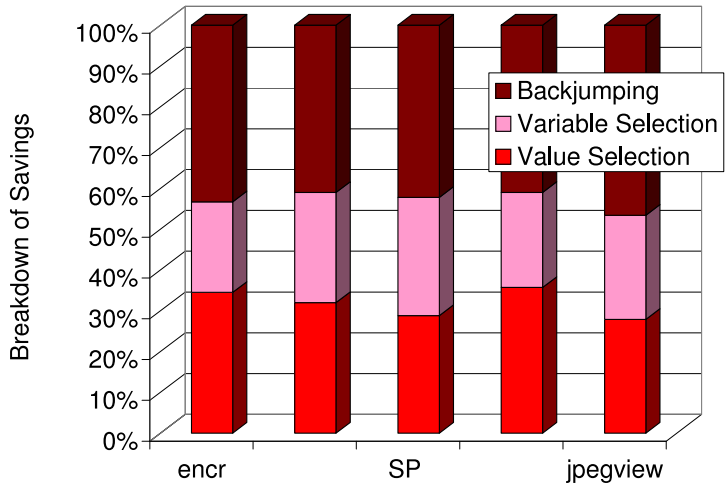


Fig. 5.6 Breakdown of benefits.

The first observation one can make from this graph is that the Integ-CN scheme generates the best savings across all five benchmark codes tested. The average savings it achieves is around 30.1%. In comparison, the average savings with the Loop-CN and Data-CN are 15.8% and 18.9%, respectively. These results clearly emphasize the importance of considering both loop and data transformations. When we look at the other schemes tested, the average savings are 10.3%, 10.7% and 15.6% for Loop-Heur, Data-Heur and Integ-Heur, respectively. Considering the results for Integ-Heur with those of Integ-CN, one can conclude that the CN based approach is very successful in practice, and outperforms a sophisticated heuristic approach that makes use of both loop and data transformations.

In our next set of experiments, we change the data cache size (keeping the remaining cache parameters fixed) and see how this effects the behavior of the schemes tested. The results are presented in Figure 5.7. Recall that the default cache size used in our experiments was 16KB. It is easy to see from this graph that, as we increase the size of the data cache, all the optimized versions tend to converge to each other. This is understandable when one remembers that the y-axis of this figure is normalized with respect to

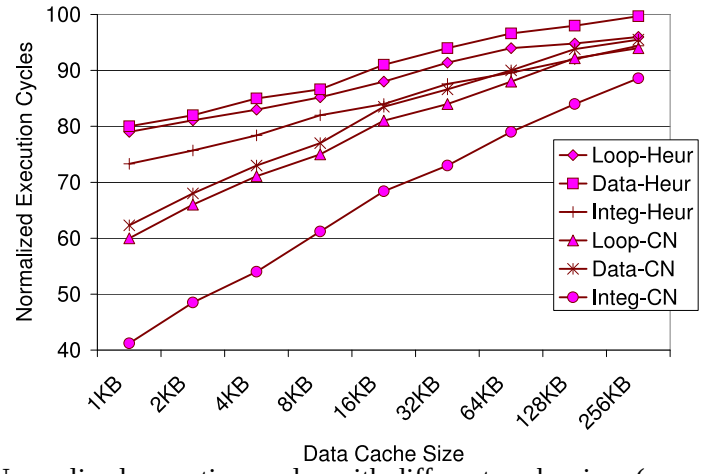


Fig. 5.7 Normalized execution cycles with different cache sizes (wood).

the execution cycles taken by the base version. The base version takes a great advantage of increasing cache size and since all the versions are normalized with respect to it, we observe relative reductions in savings with increasing size. The second observation from Figure 5.7 is that, when the data cache size is reduced, we witness the highest savings with the Integ-CN version. These results are very encouraging when one considers the fact that the increase in data set sizes of embedded applications is far exceeding the increase in on-chip storage capacities. Therefore, one can expect the Integ-CN scheme to be even more successful in the future.

We now evaluate the benefits that would be brought by the enhancements mentioned in Section 5.2.3. Recall that these enhancements are of three types, two for the forward phase and one for the backward phase. The forward phase enhancements are the careful selection of the next variable to instantiate and the careful selection of the value to be used in instantiating it. The backward phase enhancements is to backjump (instead of backtrack; see Figure 5.3) when it can be certained that backtracking would be useless at that particular point. It is important to note that these enhancements are not expected

to improve execution times. In fact, in our experiments, although in some cases they led to different selections for variables (due to the change in traversal of the state space), the overall change in execution times were negligible. Rather, one can expect them to reduce the solution time. The last column of Table 5.2, titled Integ-CN*, gives the solution times for our benchmarks under this enhanced scheme. We notice that the solution times of this enhanced scheme are much better than those obtained via the basic backtracking approach (compare the last two columns of Table 5.2). In other words, these three enhancements are very effective in shortening the solution times. We also quantified the individual contribution of each of these three enhancements to the reductions in solution times. The results are given as a stacked bar-chart in Figure 5.6, and indicate that all three enhancements are useful in practice.

Chapter 6

Constraint Network Based Code Parallelization

One of the issues that need to be tackled by compilers in the context of chip multiprocessors is automatic code parallelization. While code parallelization – in its different forms – have been around for more than two decades now, the problem requires a fresh look when considered in the context of chip multiprocessors. The main reason for this is the fact that interprocessor communication is not very expensive in chip multiprocessors and, one can tolerate some increase in interprocessor communication activity if it leads to a reduction in off-chip memory accesses, which are very costly (compared to interprocessor communication) in these architectures.

Our main goal on this subject is to explore a new code parallelization scheme for chip multiprocessors used in embedded computing. This compiler approach is based upon the observation that, in order to minimize the number of off-chip memory accesses, the data reuse across different (parallelized) loop nests should be exploited as much as possible. This can be achieved by ensuring that a given processor accesses the same set of elements in different nests as much as possible. Unfortunately, this practically makes a solution that handles one loop nest at a time unsuitable. Instead, what is needed is an approach that considers the parallelization constraints across all the loop nests in the program code being optimized. The proposed approach achieves this by employing an optimization strategy based on *constraint networks* which has been explained in the previous

chapter in detail. Constraint networks have proven to be a useful mechanism for modeling and solving computationally intensive tasks in artificial intelligence [51, 149]. They operate by expressing a problem as a set of variables, variable domains and constraints (an acceptable subset of them) and define a search procedure that tries to satisfy all the constraints by assigning values to variables from their specified domains. This chapter shows that it is possible to use a constraint network based formulation for the problem of code parallelization for chip multiprocessors.

To show that this is indeed a viable and effective alternative to currently existing approaches to code parallelization, we implemented our approach and applied it to a set of five applications that can benefit a lot from chip multiprocessing. Our experimental evaluation shows that not only a constraint network based approach is feasible for our problem but also highly desirable since it outperforms all other parallelization schemes tested in our experiments.

```
for i = 1 to N
  for j = 1 to N {
    A[i][j] = A[i][j] + B[j][i] * C[i][j];
  }
for i = 1 to N
  for j = 1 to N {
    D[i][j] = D[i][j] - B[i][j];
  }
for i = 1 to N
  for j = 1 to N {
    B[j][i] = B[j][i] * C[i][j];
  }
```

Fig. 6.1 Example program fragment.

6.1 Motivating Example

In this section, we discuss using an example code fragment why a constraint network based approach can be a desirable option over the known code parallelization techniques. For this purpose, we consider the code fragment shown in Figure 6.1. In this code fragment, three loop nests manipulate a set of arrays and these nests share arrays B and C, which are the focus of our attention. Let us assume, for ease of illustration, that we are allowed to parallelize a single loop from each loop nest of this program fragment and that we use four processors to parallelize each nest. Figure 6.2 depicts the sections of arrays B and C that are accessed by the processors. The sections accessed by the first processor are in a different color than the others. If we do not care about data sharing among different loop nests, one may choose to parallelize only the i loops from each nest.[1] However, it is not difficult to see that, under such a parallelization scheme, the first processor accesses different segments of arrays B and C in the first and second loop nests (this is true for the other processors as well). Consequently, as the execution moves from the first loop nest to the second one, this processor will make little use of the contents of the on-chip caches. Similarly, in moving from the second nest to the third one, again, the different sections of the arrays are accessed. While it is true that the first and third nests access the arrays in the same fashion (i.e., access the same array sections), the intervening nest (the second one) can easily destroy cache locality if the arrays at issue are large enough. The main problem with such a nest centric parallelization strategy is that it fails to capture the data sharings

[1] Note that none of the loops in this code fragment carries any type of data dependence that prevents the parallel execution of its iterations. Thus, the compiler has a complete flexibility in selecting the loops to run parallel.

among the different loop nests. What we would prefer, in comparison, is a parallelization scheme such that a given processor accesses the same set of array segments during the execution of the different nests of the program. In our example in Figure 6.1, such a parallelization strategy would select one of the following two alternatives: (a) parallelize the i loops from the first and third nests and the j loop from the second nest, or (b) parallelize the j loops from the first and third nests and the i loop from the second nest. It needs to be noted that, under any of these two alternatives, a processor accesses the same segments of arrays B and C in all three loop nests shown. In order to derive such parallelizations, we need an approach that captures the data sharing across the different loop nests and considers all loop nests at the same time (not in isolation). In the rest of this chapter, we present such an approach built upon the constraint network paradigm.

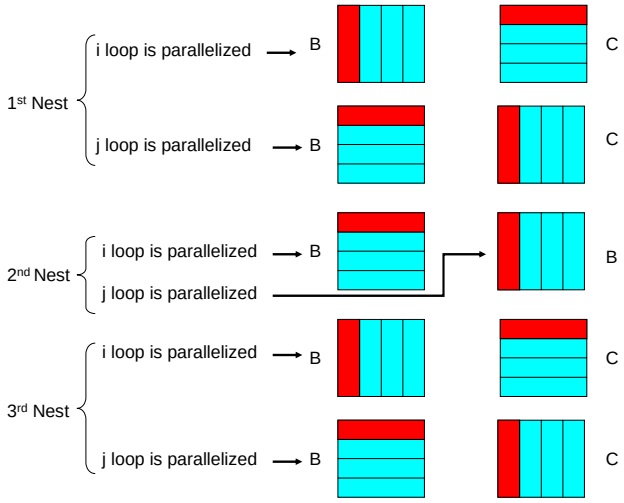


Fig. 6.2 Array partitionings under different parallelization strategies for the example code fragment in Figure 6.1.

6.2 Constraint Network Based Parallelization

6.2.1 Preprocessing

Before building our constraint network that captures all the constraints necessary for determining the loops to parallelize in a given program, we need to identify what type of data (array) access pattern is obtained after each loop parallelization. What we mean by "data access pattern" in this discussion is the style using which the array is partitioned across the parallel processors. As an example, let us consider the access pattern for array B in the first loop nest of the example in the previous section. When the i loop is parallelized, each processor accesses a set of consecutive columns of this array and we can express this access pattern using $[*, block(4)]$, which means that the second dimension of the array is distributed over 4 processors, whereas the first dimension is not distributed. Similarly, the access pattern for the same array when the j loop is parallelized (instead of i) can be captured as $[block(4), *]$. If, on the other hand, for a three-dimensional array in a different example, the first, second, and third dimensions are distributed over 3, 2, and 6 processors, respectively, the resulting data access pattern can be specified as $[block(3), block(2), block(6)]$. Capturing access patterns on different arrays is important because if two different loops, say i and j, of two different nests have the same access patterns on the same set of arrays, these two loops are good candidates for parallelization. The job of the constraint network based approach described in the next subsection is to search for such loops – throughout the program code – that, when parallelized, they result in the same access pattern. This in turn means that a processor accesses the same array

segments when these two loops are parallelized. Consequently, we can expect a good data locality (on-chip cache behavior) from such an access pattern.

6.2.2 Problem Formulation

A *constraint network (CN)* can be described as a triple $CN =< \mathcal{P}, \mathcal{M}, \mathcal{S} >$, where $\mathcal{P}$ is a finite set of variables, $\mathcal{M}$ is a list of possible values for each variable (i.e., its domain), and $\mathcal{S}$ is a set of constraints on $\mathcal{P}$ [52]. In this chapter, $\mathcal{P}$ holds the loop nests in the application code being parallelized, i.e., each element of $\mathcal{P}$ represents a loop nest in the application code. $\mathcal{M}$, on the other hand, holds the possible (legal) parallelization patterns for the loop nests. For a given loop nest $n_i \in \mathcal{P}$ with L loops, each element of the corresponding entry in $\mathcal{M}$, say m_i, is an L-bit sequence where the j^{th} bit indicates whether the j^{th} loop of nest n_i is parallelized ($L(j)$=1) or not ($L(j)$=0). Note that while a nest n_i with L loops can potentially have 2^L possible bit sequences (different combinations of 0 and 1) in m_i, many of these sequences would not be acceptable in practice. For example, due to data and control dependencies, sometimes it is not possible to parallelize a certain loop in a given nest. In this case, we can drop all the sequences with the associated bit set to 1. Also, in some cases, we may not want to exploit a very fine grain parallelism in the loop nest, that is, we may want to limit the number of loops that can run in parallel. This can be achieved by dropping the sequences with certain number of 1s. The last component of our constraint network, $\mathcal{S}$, captures the constraints that need to be satisfied for maximizing data reuse among different loop nests. As stated above, maximizing data reuse among loop nests is important for reducing the number of off-chip data references, which in turn helps reduce both power consumption and execution cycles. An

entry of $\mathcal{S}$, say s_k, is of the form $bs_{k1}, bs_{k2}, \cdots, bs_{kq}$, where each bs_{kp} - where $1 \leq p \leq q$ - is a bit sequence for a nest in the application. What an entry such as s_k indicates to the constraint network that, for exploiting inter-nest data reuse, the corresponding nests should have the bit sequences bs_{k1}, bs_{k2}, $\cdots$, bs_{kq}. We want to emphasize that an $\mathcal{S}$ set may have multiple entries and we need to satisfy only one of them.

Let us now consider an example that illustrates how we can build a constraint network for a specific example. We consider again the program fragment given in Figure 6.1. The corresponding constraint network can be written as follows under the assumption that we are allowed to parallelize only a single loop from each nest:

$$\begin{aligned} CN &= <\mathcal{P}, \mathcal{M}, \mathcal{S}>, \quad \text{where} \\ \mathcal{P} &= \{n_1, n_2, n_3\}; \\ \mathcal{M} &= \{\{(1,0),(0,1)\},\{(1,0),(0,1)\},\{(1,0),(0,1)\}\}; \\ \mathcal{S} &= \{\{(1,0),(0,1),(1,0)\},\{(0,1),(1,0),(0,1)\}\} \end{aligned}$$

We now discuss briefly the components of this constraint network formulation. First, the $\mathcal{P}$ set says that there are three loop nests (n_1, n_2, and n_3) in the program fragment under consideration. The domain set, $\mathcal{M}$, on the other hand, indicates the potential values (parallelization strategies) that each loop nest can assume. For example, the first entry in $\mathcal{M}$, m_1 = $\{(1,0),(0,1)\}$, tells us that, as far as nest n_1 is concerned, there are two possible parallelization strategies. The first strategy, captured by bit sequence $(1,0)$,

corresponds to the case where the first loop is parallelized and the second one is run sequentially (i.e., not parallelized). The second strategy, captured by $(0,1)$, represents an opposite approach where the second loop is parallelized and the first loop is to be executed serially. The reason why we have only two options (two alternate parallelization strategies) for the first loop nest is our assumption above which states that we can parallelize only a single loop from each nest. Relaxing this assumption (i.e., allowing the possibility of parallelizing both the loops) would modify the corresponding entry for n_1 in $\mathcal{M}$ to $\{(1,0),(0,1),(1,1)\}$. We use domain variable m_i to represent the possible (allowable) parallelization strategies for a nest n_i. In our running example, m_1 = m_2 = m_3 = $\{(1,0),(0,1)\}$, which means all three loop nests have the same flexibility as far as parallelization is concerned.

We now explain the constraint set $\mathcal{S}$. This set essentially captures the constraints for maximizing the data reuse across the different nests from a processor's perspective. Let us first focus on the entry $\{(1,0),(0,1),(1,0)\}$. What this entry specifies that in order to maximize the inter-nest data reuse, one option is to parallelize the first loop from the first nest, captured by bit sequence $(1,0)$; the second loop from the second nest, captured by $(0,1)$; and the first loop from the third nest, captured by $(1,0)$. The other entry in $\mathcal{S}$, $\{(0,1),(1,0),(0,1)\}$, can be interpreted in a similar way. The obvious question is how one can come up with these entries, i.e., how we can populate $\mathcal{S}$. The answer to this question lies in the explanation given in Section 6.2.1. When we detect the fact that parallelizing the first loop, second loop and first loop from the first nest, second nest and third nest, respectively, allows a processor to access the same array segments, we build the entry $\{(1,0),(0,1),(1,0)\}$ and put it in the set $\mathcal{S}$.

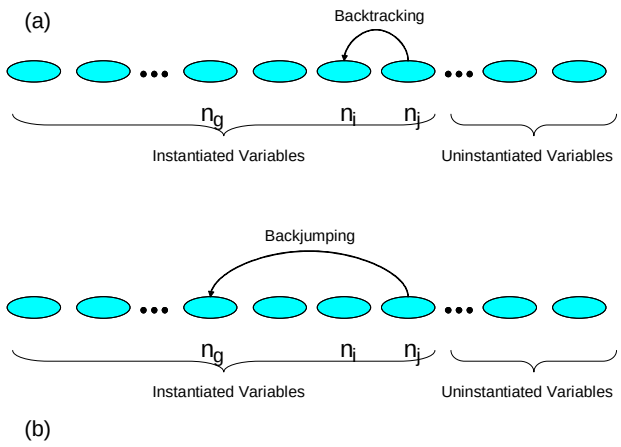


Fig. 6.3 Comparison of backtracking (a) and backjumping (b).

It needs to be emphasized at this point that our job is to select a parallelization strategy for each loop nest, i.e., to choose an entry from m_i for n_i, such that at least one of the entries (s_j) in $\mathcal{S}$ is completely satisfied. For example, if we select $(1, 0)$ from m_1, $(0, 1)$ from m_2, and $(1, 0)$ from m_3, we satisfy the first entry of $\mathcal{S}$ – $\{(1, 0), (0, 1), (1, 0)\}$. While, in this simple example, it is relatively straightforward to identify the required selections, it is not difficult to imagine that, for a large embedded application with tens of loop nests and arrays, this is not a trivial problem to solve. In the next section, we discuss an automated search strategy for this problem.

6.2.3 Proposed Solution

We now discuss a backtracking based solution to the problem of determining a suitable parallelization strategy for a given program. Our solution operates on the constraint network based representation of the problem explained above. We later discuss several enhancements to our backtracking based approach that aim at minimizing the time to find a solution. Recall from our discussion in the previous section that our task is to

assign, for each nest n_i, a value from the corresponding m_i domain such that at least one of the entries, say s_j, of $\mathcal{S}$ is completely satisfied. Here, we have $1 \leq i \leq V$ and $1 \leq j \leq W$, where V is the number of loop nests in the application program and W is the number of possible constraints, any of which can be satisfied.

In a given constraint network, a *partial instantiation* of a subset of variables is an assignment to each variable from its domain. A *consistent partial instantiation*, on the other hand, is a partial instantiation that satisfies all the constraints (of an s_j in our problem) that involve only the instantiated variables [52]. A *backtracking* based algorithm basically traverses the state space of partial instantiations in a depth-first manner. It starts with an assignment of a variable (e.g., randomly selected) and then increases the number of partial instantiations. When it is found that no solution can exist based on the current partial instantiation, it backtracks to the previous variable instantiated, and re-instantiates it with a different value from its domain. Therefore, a backtracking algorithm has both *forward* (where we select the next variable and instantiate it with a value) and *backward* phases (where we return to the previously instantiated variable and assign a new value to it). In the rest of the chapter, this backtracking based scheme is referred to as our *base scheme*. In our code parallelization problem, the base scheme operates as follows. We first select a parallelization strategy from among the strategies specified in the m_1 set for the first nest (say n_1). All s_js that cannot accept this assignment are dropped from consideration immediately. If there exists at least an s_j that can accept this assignment, we continue by selecting a parallelization for n_2, the second nest. If there still exists at least an s_j that is consistent with the assignments made so far (i.e., a consistent partial instantiation), we go ahead with the third nest, and so on. At some point, if we determine that there is no

s_j that is partially consistent with the assignments made so far, we backtrack by changing the parallelization selection made for the last nest considered and check for consistency again, and so forth. At the end of these forward and backward movements (phases), we have either of two outcomes. We either satisfy an s_j completely (i.e., full instantiation) or we fail to satisfy any s_j. In the latter, we can relax the constraints in $\mathcal{S}$ or add some more entries (alternate s_js) to $\mathcal{S}$ and retry.

Let us apply the backtracking based solution to the example given in Figure 6.1. Based on the $\mathcal{P}$, $\mathcal{M}$, and $\mathcal{S}$ sets defined earlier for this code fragment, we first select $(1,0)$ from m_1 for the first nest, n_1. Since this assignment will be consistent with only s_1, s_2 will be dropped and we will proceed with the second loop nest. We continue by selecting $(1,0)$ from m_2 for n_2, which will only be consistent with s_2. Since s_2 was dropped in the previous step, there is no s_j that is partially consistent with this selection. At this point, we backtrack and select a different entry for n_2 from m_2, that is, we continue with $(0,1)$. This will be partially consistent with s_1 = $\{(1,0),(0,1),(1,0)\}$. So far, $(1,0)$ and $(0,1)$ have been selected for the first and the second loop nests, respectively, which is consistent with s_1. Next, we select $(1,0)$ from m_3 for the last loop nest, n_3. Again, this will be consistent with s_1 resulting in a full (consistent) instantiation, which satisfies s_1 completely. Hence, a suitable parallelization strategy is determined for the given code fragment by selecting $(1,0)$ from m_1, $(0,1)$ from m_2, and $(1,0)$ from m_3.

The important point to note is that the base scheme explained above makes random decisions at several points. The first random decision is to select the next variable (array) to instantiate during the forward phase. The second random decision occurs when selecting the value (parallelization strategy) with which the chosen variable is instantiated

(again in the forward phase). In addition, in our base scheme, when we find out that the current instantiation cannot generate a solution, we always backtrack to the previously assigned variable, which may not necessarily be the best option. One can improve all these three aspects of the base scheme as follows. As for the first random decision, we replace it by an improved approach that instantiates, at each step, the variable that maximally constrains the rest of the search space. The rationale behind this is to be able to detect a dead-end as early as possible during the search. Similarly, when selecting the values to be assigned to the instantiated variables, instead of selecting a value randomly, we can select the value that maximizes the number of options available for future assignments. The rationale behind this is to increase the chances for finding a solution quickly (if one exists). Finally, we can expedite our search by *backjumping*, i.e., instead of backtracking to the previously instantiated value, we can backtrack further when it is beneficial to do so.

Backjumping in our approach can be best explained using an example scenario. Suppose that, in the previous step, we selected a parallelization strategy $x \in m_i$ for loop nest n_i, and in the current step, we selected a parallelization strategy $y \in m_j$ for loop nest n_j. If, at this point, we see that there cannot be any solution based on these assignments (i.e., we cannot satisfy any $s_k \in \mathcal{S}$), our base approach explained above backtracks to nest n_i and selects a new alternative for it (i.e., tries a new parallelization strategy, say $z \in m_i$, for it), assuming that we have already tried all alternatives for loop nest n_j. However, it must be noted that, if there is no constraint in the network in which both n_i and n_j appear together, that is, these loop nests do not share any array between them, assigning a new value (parallelization strategy) to n_i would not generate a solution, as n_i cannot be the

culprit for reaching the dead-end. Instead, backjumping skips n_i and determines a loop nest (say n_g) among the loop nests that have already been instantiated that co-appears with n_j in a constraint, and assigns a new parallelization strategy to it (i.e., different from its current value). In this way, backjumping can prevent useless assignments that would normally be performed by backtracking and, as a result, expedite our search. Figure 6.3 gives an illustration that compares backtracking and backjumping. In the remainder of this chapter, our base approach supported by these three improvements is referred to as the *enhanced scheme.* The next section presents experimental data for both the base and enhanced schemes. Before going into our experimental analysis though, we need to make one point clear. If a solution exists to the problem under consideration, both the base and enhanced schemes will find it. However, if multiple solutions exist, they can potentially find different solutions, though it was not the case in our experiments. In most cases, we can expect to improve (reduce) solution times while returning the same results.

6.3 Experimental Evaluation

For each benchmark code in our experimental suite, we generated five different versions. The first version, called the original version, is the unmodified/unparallelized code executed on a single processor. The results with all other versions are given as speedups over this version. These other versions are parallelized codes. The second version we tested parallelizes each loop nest in isolation and is referred to as the nest based parallelization in this section. The third and fourth versions improve over the second one by carrying out some information from one loop nest to another. For example, when a loop nest is being parallelized, we record how the processors access the arrays referenced

in the nest, and use this information in parallelizing the next loop nest. For example, if the first loop nest results in an access pattern represented by $(block(8), *)$ for an array, the parallelization strategy attempted in the second loop nest tries to achieve the same access pattern for the same array. These two versions, named global (first nest) and global (ordered), differ from each other in the order at which the loop nests are processed. Specifically, in the global (first nest) version, the loop nests are processed (parallelized) starting from the first one and ending with the last one, according to their textual order in the program. In comparison, in the global (ordered) version, we first determine an order of processing for the loop nests and then process them according to this order (note that this is just an order for processing the nests and we do not modify the textual positions of the nests with respect to each other). The order (of processing) used in our current implementation is based on the estimated cycle count for the nests (which is obtained through profiling). That is, we parallelize the loop nests one-by-one starting with the most expensive one and ending with the least expensive one. The last version evaluated in our experiments is the constraint network (CN) based approach proposed in this work (our base scheme). As explained earlier, it first formulates the problem on a constraint network and then finds a solution based on a backtracking based search strategy.

We made our experiments using a SimpleScalar based infrastructure [14]. Specifically, we spawned a CPU simulation process for simulating the execution of each processor in our chip multiprocessors and a separate communication simulation process captured the data sharing and coherence activity among the processors. Each processor in the system has been modeled as a simple embedded core (300MHz) that can issue and execute two instructions at each clock cycle. Each processor has separate L1 instruction and data

Table 6.1 Benchmark codes.

Benchmark	Explanation	Domain Size	Data Size	Compilation Time (msec)	Execution Time (sec)
Med-Im04	Medical Image Reconstruction	311	825.55KB	812.52	204.27
Laplace	Laplace Equation Solver	887	1,173.56KB	867.30	69.31
Radar	Radar Imaging	497	905.28KB	690.74	192.44
Shape	Pattern Recognition and Shape Analysis	761	1,284.06KB	1,023.46	233.58
Track	Visual Tracking Control	423	744.80KB	785.62	231.00

caches (each is 16KB, two-way set-associative with a line size of 32 bytes). The assumed latencies for the on-chip caches and the off-chip memory are 2 cycles and 90 cycles, respectively.

The benchmark codes used in our experiments are given in Table 6.1. The third column shows the total search space size (i.e., the sum of the domain sizes of all the loop nests in the application). The fourth column gives the total data size manipulated by each application (sum of all the data arrays). The last two columns give the compilation time and execution time of each benchmark under the original version. We implemented our constraint network using the C++ programming language. Excluding the libraries linked and comment lines, this implementation is about 2,050 C++ lines.

We first present in Figure 6.4 the speedups under four optimized versions of each benchmark when 8 processors are used for execution. We compute the speedup as $T/T'(p,v)$, where T is the execution time of the original version (see the last column of Table 6.1) and $T'(p,v)$ is the parallel execution time of version v under p processors. One of the important results from this bar-chart is that our constraint network based approach generates the best execution times for all the five benchmarks. The average speedups with

the nest based, global (first nest), global (ordered) and our approach are 4.48, 5.72, 5.66 and 6.57, respectively. In addition to this main result, we also observe some other interesting trends from this graph. For example, the loop nest based parallelization strategy results in the worst performance among all the parallel versions we have, meaning that it is very important to capture the interactions between the different loop nests (this is a direct result of array sharing across the loop nests in our applications). In comparison, the versions global (first nest) and global (ordered) perform better than the nest based version. In fact, except for the benchmark Track, the global (first nest) version generates better results than the nest based parallelization. Another interesting result we see is that, except for the benchmark Laplace, global (ordered) perform better than global (first nest), indicating that it is also critical to select a good order to process the loop nests. This is because if we do not start processing with the most expensive nest, it is very likely that we will have a lot of constraints to satisfy when we come to process it. Our constraint network based approach does not have this problem because it determines a solution that satisfies the needs of all the loop nests (if such a solution actually exists). On the other hand, the reason why our approach generates better results than the global (ordered) scheme is that the latter does not optimize all the nests at the same time. Although it favors the optimization of the costly nests over the cheaper ones, in many cases, under the global (ordered) scheme, a number of cheaper nests go unoptimized for the sake of optimizing a single costly nest. Even worse, in some other cases, there are two (almost) equally expensive loop nests and the decisions made by the global (ordered) scheme in optimizing one of them can restrict the potential optimization opportunities for the other.

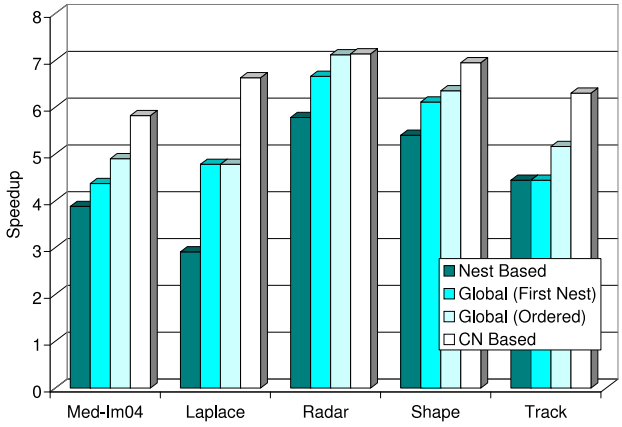


Fig. 6.4 Speedups with different versions over the original version (8 processors).

Table 6.2 Solution times taken by different versions. All times are in seconds.

Benchmark	Nest Based	Global (First Nest)	Global (Ordered)	CN Based
Med-Im04	6.89	9.18	9.33	66.84
Laplace	9.53	12.16	12.83	51.07
Radar	5.97	8.07	8.61	72.59
Shape	9.50	11.74	12.13	82.94
Track	8.54	10.93	11.35	64.30

While the improvements in execution time presented above are certainly important and encouraging, for a fair comparison of the different versions, we need to look at the time they spend in finding a solution as well. Table 6.2 gives (in seconds) the solution times for the different methods. We see from these results that, as expected, our approach takes more time to find a solution than the remaining three optimized (parallel) versions. However, as we will discuss shortly, we can cut this solution time dramatically by adopting a more intelligent search and variable/value selection procedure.

We next look at the behavior of the different versions when the number of processors is changed. While we present results in Figure 6.5 only for the benchmark Shape, the observations we make extend to the remaining four benchmarks as well. We see that the scalability exhibited by the nest based, global (first nest) and global (ordered) versions are

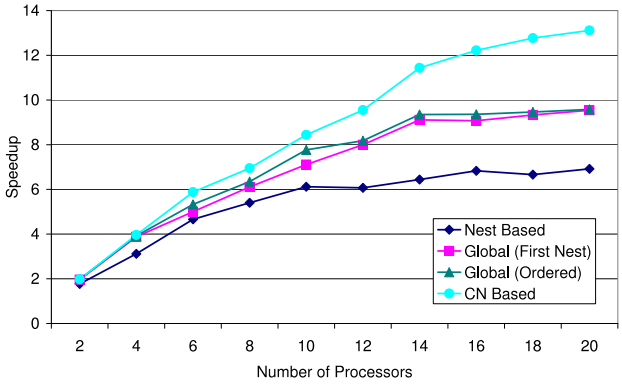


Fig. 6.5 Influence of the number of processors (Shape).

not very good (as we increase the number of processors). This is mainly because of the increased amount of interprocessor data sharing activity which becomes really a dominant factor beyond a certain number of processors. Our approach on the other hand effectively increases the total on-chip cache capacity by maximizing the data reuse across the different loop nests in these applications and this in turn enhances it scalability substantially.

To see what happens when we have more on-chip storage, in our next set of experiments, we increased the per processor L1 capacity to 64KB (recall that the default value used so far was 16KB per processor). The speedup results are presented in Figure 6.6 (for the 8 processor case). When we compare these results with those presented in Figure 6.4, we see that the speedups are reduced for all the parallel versions. The main reason for this behavior is that the original version takes a great advantage of the increased on-chip storage (from 16KB to 64KB for that version) and, as a consequence, the relative speedups achieved by the parallel versions are reduced. We also observe however that, even with a 64KB data cache per processor, the average speedups with the nest based, global (first nest), global (ordered) and our approach are 4.41, 5.07, 5.44 and 6.29, respectively. In other words, our approach still generates the best results among all the versions tested.

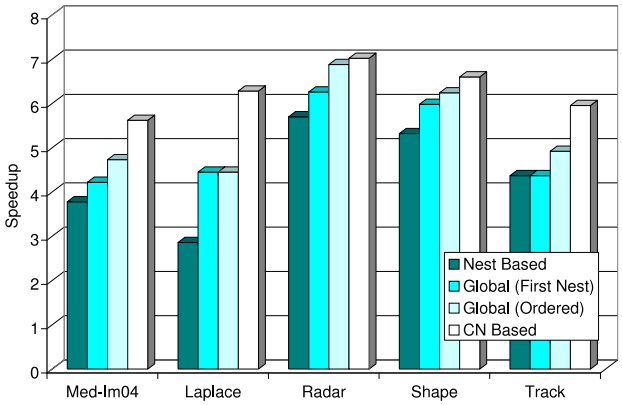


Fig. 6.6 The speedup results with 64KB per processor data caches.

Recall that we discussed in Section 6.2.3 three possible enhancement to our base approach. The first enhancement is to do with the selection of the variable to instantiate next; the second one is related to the selection of the value to be assigned to the selected variable; and the last one is to employ backjumping instead of backtracking. We now quantify the benefits coming from these three enhancements. Our experiments with these enhancements showed that they do not have a significant impact on the solution found (though, in some cases, the solutions found by the base and enhanced versions of our approach differ slightly); therefore, the enhanced version generates almost the same execution time results as our base version used so far in our experimental evaluation. However, these three enhancements impact the solution times significantly. Figure 6.7 presents the reduction in solution times over the last column of Table 6.2. The solution time reduction is in the range of 61% – 70% when considering all five benchmarks. In addition, each bar in Figure 6.7 is divided into three parts, each corresponding to one of the three potential optimizations discussed in Section 6.2.3. While, according to these results, most of the benefits come from backjumping, all three enhancements are very useful in general and

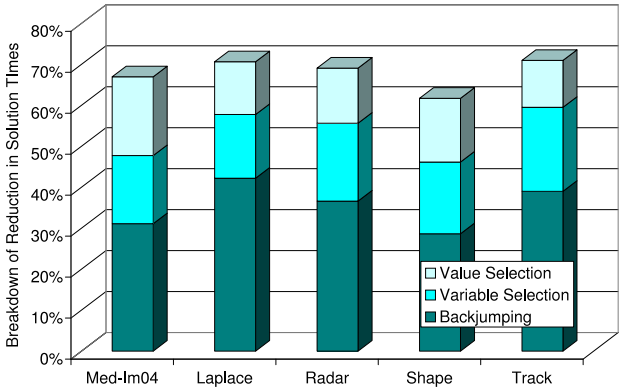


Fig. 6.7 Reductions in solution times brought by the enhanced version over the base version.

contribute significantly to the overall reduction in solution times, without affecting the quality of the generated output code in any way.

Chapter 7

Optimal Topology Exploration for Application-Specific 3D Architectures

As technology scales, the International Technology Roadmap for Semiconductors projects that on-chip communications will require new design approaches to achieve system level performance targets [67]. Three-dimensional integrated circuits (3D ICs) [22, 47, 48, 53] are attractive options for overcoming the barriers in interconnect scaling, offering an opportunity to continue the CMOS performance trend. In a three-dimensional (3D) chip, multiple device layers are stacked together with direct vertical interconnects tunneling through them. Consequently, one of the most important benefits of a 3D chip over a traditional two-dimensional (2D) design is the reduction on global interconnect. Other benefits of 3D ICs include: (i) higher packing density and smaller footprint due to the addition of a third dimension to the conventional two-dimensional layout; (ii) higher performance due to reduced average interconnect length; (iii) lower interconnect power consumption due to the reduction in total wiring length; and (iv) support for realization of mixed-technology chips.

As heat is generated by the power dissipated on the chip, the on-chip junction temperatures also increase, resulting in higher cooling/packaging costs, acceleration of failure mechanisms, and degradation of performance. For 3D ICs, the thermal issues are even more pronounced than the 2D designs due to higher packing density, especially for the inner layer of the die. It is often considered as a major hindrance for 3D integration

[22]. For example, temperature increases force the design to slow down to cool the chip, such that the actual performance benefits from reducing global interconnect could be offset. Therefore, thermal-aware design is very critical to extract the maximum benefits from the 3D integration.

As technology moves towards 3D designs, one of the challenging problems in the context of chip multiprocessor systems is the placement of processor cores and storage blocks across the multiple layers available. This is a critical problem as both power and performance behavior of a design are significantly influenced by the data communication (data access) distances between the processor cores and storage blocks. In particular, if a computation block (e.g., a processor core) frequently accesses data from certain storage blocks, these storage blocks should be placed into positions close (in vertical or horizontal sense) to that processor core. Similarly, in a chip multiprocessor design, if two cores frequently share certain data residing in a given storage block, that storage block should be put close to both these cores to minimize the data communication distances, thereby, improving potentially both performance and power consumption.

The important point to note, however, is that each embedded application can require a different placement of processor cores and storage blocks for achieving the minimum data communication distances. Therefore, in this chapter, our focus is on *application-specific placement* of processor cores and storage blocks in a 3D design space. For this purpose, we propose an integer linear programming (ILP) based processor core/storage block placement for single-core and chip multiprocessor embedded designs. This placement problem is also referred to as topology exploration. While ILP based solutions are known to take large solution times, this issue does not seem to be very pressing in our case

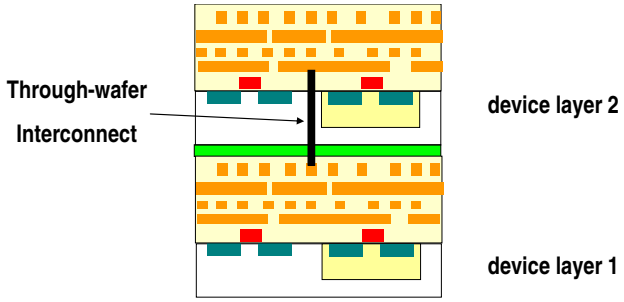


Fig. 7.1 A 3D chip with 2 device layers built from direct vertical interconnect tunneling through them.

because (i) we design a customized placement for a given embedded application, and thus, can afford large solution times as design quality is of utmost importance, and (ii) since a given design typically has only a small number of processor cores and storage blocks, the solution times can be kept under control.

We implemented our ILP based solution within a commercial solver [1], and performed experiments with different types of applications. The first group of applications are a set of six sequential Spec2000 codes and used for single-core designs in this work. The second group of applications, on the other hand, are for chip multiprocessor designs and include four array-based codes parallelized through an optimizing compiler. Our experiments that consider a set of cache lines as a storage block reveal two important results: (i) the best 3D designs significantly outperform the best 2D designs (also obtained using ILP) for a given application under the same temperature constraint for both single-core and chip multiprocessor applications, and (ii) optimized placement of blocks is very important in the 3D domain as the difference between the optimized and random core/storage block placements (in 3D) is very significant in all the cases tested.

7.1 3D Thermal Model

In order to facilitate the thermal-aware processor core/storage block placement process, a compact thermal model is needed to provide the temperature profile. Numerical computing methods (such as finite difference method (FDM) [147]) are very accurate but computationally intensive, while the simplified close-form formula [65] is very fast but inaccurate. Skadron et al proposed a thermal model called Hotspot [142], which is based on lumped thermal resistances and thermal capacitances. It is more efficient than the prior low-level approaches since the variances at temperature are tracked at a granularity of functional block level. In our research, we use a simplified analytical model called the 3D resistor mesh (shown in Fig. 7.2), which is similar to the approach taken by Hotspot, to facilitate the thermal analysis. The model employs the principle of thermal-electrical duality to enable efficient computation of the thermal effects at the block level. The transfer thermal resistance $R_{i,j}$ of block i with respect to block j can be defined as the temperature rise at the block i due to one unit of power dissipated at block j:

$$R_{i,j} = \frac{\Delta T_{i,j}}{\Delta P_{i,j}}.$$

In this simplified model, the device layers are stacked together with the heat sink as the bottom layer, and each device layer consists of the blocks, which are the storage (cache memory) blocks or the processor core(s). These blocks define the thermal nodes of the thermal resistor mesh. (For more details, the reader is referred to [142].) A 3D resistor mesh consists of vertical and lateral thermal resistors, which model the heat flow from

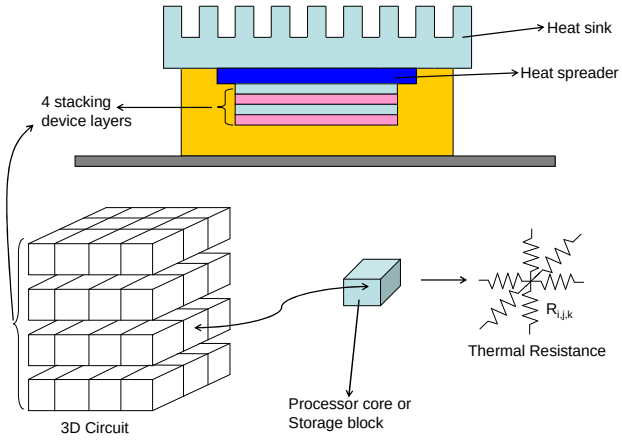


Fig. 7.2 3D resistor mesh model.

wafer to wafer, wafer to heatsink, and heat transferring between the blocks. These vertical and lateral resistances are calculated based on Hotspot [142].

After the 3D thermal resistance network has been determined, the temperature rise at the each block can be estimated by solving the following equation:

$$T = R \times P,$$

where T is the temperature of each block, R is a matrix of thermal resistances, and P is the power consumption of each block. One of the nice properties of this high-level abstraction of temperature behavior is that it can easily be embedded within an ILP-based optimization framework (since it is linear), as will be discussed in the next section.

7.2 ILP Formulation

Our goal in this section is to present an ILP formulation of the problem of minimizing data communication cost of a given application by determining the optimal placement of storage blocks and processor cores under a given temperature bound. A storage block

Table 7.1 The constant terms used in our ILP formulation. These are either architecture specific or program specific. Note that C_Z captures the number of layers in the 3D design. By setting C_Z to 1, we can model a conventional 2D (single layer) design as well. The values of $FREQ_{p,m}$ are obtained by collecting statistics through simulating the code and capturing accesses to each storage block.

Constant	Definition
P	Number of processor cores
M	Number of storage blocks
C_X, C_Y, C_Z	Dimensions of the chip
P_X, P_Y	Dimensions of a processor core
$SIZE_m$	Size of a storage block m
$FREQ_{p,m}$	Number of accesses to storage block m by processor p
$R_{l,v}$	Thermal resistance network
T_B	Temperature bound

in this work corresponds to a set of consecutive cache lines. The data cache is assumed to be divided into *storage blocks* of equal size. In this work, our focus is on the data cache only; however, the proposed approach can be applied to the instruction cache as well.

ILP provides a set of techniques that solve those optimization problems in which both the objective function and constraints are linear functions and the solution variables are restricted to be integers. The 0-1 ILP is an ILP problem in which each (solution) variable is restricted to be either 0 or 1 [103]. Table 7.1 gives the constant terms used in our ILP formulation. We used *Xpress-MP* [1], a commercial tool, to formulate and solve our ILP problem, though its choice is orthogonal to the focus of this chapter. In our ILP formulation, we view the chip area as a 3D grid, and assign storage blocks and cores into this grid.

Assuming that P denotes the total number of processor cores, M the total number of storage blocks, (C_X, C_Y, C_Z) the dimensions of the chip, (P_X, P_Y) the dimensions of the processor core, our approach uses 0-1 variables to specify the coordinates of each storage block and processor core.

Table 7.2 The access percentage of each block by different processors.

Processor	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}
1	0.20	0.18	61.76	0.19	0.17	0.20	3.48	2.86	0.20	0.20
2	0.20	0.19	61.86	0.19	0.22	4.07	2.30	0.18	0.19	0.18
3	0.18	0.18	61.83	0.18	0.18	4.05	2.34	0.19	0.21	0.19
4	0.18	0.22	61.76	0.19	0.18	4.05	2.32	0.18	0.20	0.18
Processor	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}
1	22.83	0.22	0.20	0.18	0.20	0.19	0.18	0.19	1.83	4.55
2	22.64	0.20	0.18	0.18	0.17	0.19	0.18	1.91	4.49	0.28
3	22.65	0.20	0.20	0.22	0.19	0.20	0.18	1.89	4.47	0.29
4	22.59	0.17	0.18	0.18	0.19	0.21	0.18	1.89	4.49	0.31

We use PC to identify the coordinates of a processor core. More specifically,

- $PC_{p,x,y,z}$: indicates whether processor core p is in (x, y, z).

Similarly, MC is used in our formulation to identify the coordinates of a storage block.

- $MC_{m,x,y,z}$: indicates whether storage block m is in (x, y, z).

Although the size of a storage block is given, its dimensions may vary. We use MD to capture the dimensions of a storage block. Specifically, we have:

- $MD_{m,x,y}$: indicates whether storage block m has dimensions of (x, y).

The mapping between the coordinates and the blocks is ensured by variable $MMap$ for the storage blocks, and variable $PMap$ for the processor cores. That is,

- $MMap_{x,y,z,m}$: indicates whether coordinate (x,y,z) is assigned to storage block m.

- $PMap_{x,y,z,p}$: indicates whether coordinate (x,y,z) is assigned to processor core p.

The distances between a processor core and a storage block on each axis (x,y and z) are captured by $Xdist_{p,m,x}$, $Ydist_{p,m,y}$, and $Zdist_{p,m,z}$. Specifically, we have:

- $Xdist_{p,m,x}$: indicates whether the distance between processor core p and storage block m is equal to x on the x-axis.

- $Ydist_{p,m,y}$: indicates whether the distance between processor core p and storage block m is equal to y on the y-axis.

- $Zdist_{p,m,z}$: indicates whether the distance between processor core p and storage block m is equal to z on the z-axis.

In order to facilitate the thermal-aware core/storage block placement, power and temperature values need to be calculated. Temperature of each block (T_i) is obtained using the resistance vector (R) and the corresponding power consumption values (obtained through a Wattch-based simulation [25]).

- $Power_m$: is the power consumption for block m.

- $Temp_m$: is the temperature of block m.

A processor core needs to be assigned to a single coordinate:

$$\sum_{i=0}^{C_X-1}\sum_{j=0}^{C_Y-1}\sum_{k=0}^{C_Z-1} PC_{p,i,j,k} = 1, \quad \forall p. \tag{7.1}$$

In the above equation, i, j and k correspond to the x, y and z coordinates, respectively. A storage block also needs to be assigned to a unique coordinate:

$$\sum_{i=0}^{C_X-1}\sum_{j=0}^{C_Y-1}\sum_{k=0}^{C_Z-1} MC_{m,i,j,k} = 1, \quad \forall m. \tag{7.2}$$

A storage block needs to have unique dimensions:

$$\sum_{i=1}^{C_X}\sum_{j=1}^{C_Y} MD_{m,i,j} = 1, \quad \forall m. \tag{7.3}$$

Each storage block should have dimensions in such a way that its size, $SIZE_m$ (given as input), will fit into the allocated space. That is, $SIZE_m = width \times height$:

$$SIZE_m = \sum_{i=1}^{C_X}\sum_{j=1}^{C_Y} MD_{m,i,j} \times i \times j, \quad \forall m. \tag{7.4}$$

Storage blocks should be mapped to the chip based on the coordinate and dimensions of the corresponding storage block. This requirement can be captured as follows:

$$MMap_{x,y,z,m} \geq MC_{m,x_1,y_1,z-1} + MD_{m,dx,dy} - 1,$$
$$\forall m, x, x_1, dx, y, y_1, dy, z, z_1, dz \text{ such that}$$
$$x_1 + dx \geq x > x_1, \text{ and } y_1 + dy \geq y > y_1. \tag{7.5}$$

In this expression, x_1, y_1 and z_1 denotes the x, y, and z coordinates of a storage block. Similarly, dx and dy denote the dimensions of the storage block. Based on these values, $MMap$ assigns the corresponding coordinates to the storage block m. Similarly, processor cores should be mapped to the chip, which can be expressed as follows:

$$PMap_{x,y,z,p} \geq PC_{p,x_1,y_1,z-1}, \forall p, x, x_1, y, y_1, z, z_1$$
$$\text{such that} \quad x_1 + P_X \geq x > x_1, \quad \text{and} \quad y_1 + P_Y \geq y > y_1. \tag{7.6}$$

In order to prevent multiple mappings of a coordinate in our grid, we force a coordinate to belong a single processor core or a single storage block:

$$\sum_{i=1}^{M} MMap_{i,x,y,z} + \sum_{i=1}^{P} PMap_{i,x,y,z} = 1, \forall x, y, z. \tag{7.7}$$

Manhattan Distance is assumed to be the cost of the *data communication* between a storage block and a processor core. This is also referred to as the *data access cost* in this chapter, and is the metric whose value we want to minimize. Note that in our architecture processor cores communicate by sharing data in storage blocks. To capture the Manhattan Distance, we use two variables, namely, $Xdist_{p,m,x}$ and $Ydist_{p,m,y}$, and employ the following constraints:

$$Xdist_{p,m,x} \geq PC_{p,x_1,y_1,z_1} + MC_{m,x_2,y_2,z_2} - 1,$$
$$\forall p, m, x, x_1, x_2, y_1, y_2, z_1, z_2 \text{ such that } x = |x_1 - x_2|. \tag{7.8}$$

$$Ydist_{p,m,y} \geq PC_{p,x_1,y_1,z_1} + MC_{m,x_2,y_2,z_2} - 1,$$
$$\forall p, m, x_1, x_2, y, y_1, y_2, z_1, z_2 \text{ such that } y = |y_1 - y_2|. \tag{7.9}$$

In addition to the x and y dimensions, there is a communication cost due to z dimension, which captures the vertical dimension. Data communication across the layers is not the same as the communication within a given layer, because it only depends on the wafer thickness, which does not change with the block size, so it needs to be captured separately

as follows:

$$Zdist_{p,m,z} \geq PC_{p,x_1,y_1,z_1} + MC_{m,x_2,y_2,z_2} - 1,$$

$$\forall p, m, x_1, x_2, y_1, y_2, z, z_1, z_2 \text{ such that } z = |z_1 - z_2|. \tag{7.10}$$

In wafer-bonding 3D technology, the dimensions of the vertical through-wafer interconnect are not expected to scale at the same rate as feature size, because wafer-to-wafer alignment tolerances during bonding pose limitations on the scaling of the through-wafer interconnect. Current dimensions of through-wafer via sizes vary from 1μm-by-1μm to 10μm-by-10μm [22, 48, 53]. The relatively large size of via makes the interconnect delay going through wafer to be relatively much smaller.

In order to formulate the temperature constraints, we need to express the power consumption using the following equation:

$$\begin{aligned} Power_m &= \sum_{i=1}^{P}\sum_{j=1}^{M} FREQ_{i,j} \times PC_{i,x,y,z} \times PPOWER_i + \\ & FREQ_{i,j} \times MC_{j,x,y,z} \times MPOWER_j, \ \ \forall m, x, y, z \text{ such that} \\ & m = (z \times C_Y \times C_X) + (y \times C_X) + (x + 1). \end{aligned} \tag{7.11}$$

In this expression, if a processor is assigned to a certain coordinate, power consumption is shaped by that specific processor. Similarly, a memory block forms the power consumption for a specific coordinate if it is assigned to that location. Note that, for each (x, y, z) coordinate in the mesh, only one of these power consumption values will be returned as

non-zero. $PPOWER$ and $MPOWER$ are average power consumption values for each processor or memory block obtained using Wattch [25] and Cacti [137].

We next calculate the temperature of each block using the $Temp = R \times Power$ equation. More specifically,

$$Temp_m = \sum_{j=1}^{P+M+1} R_{m,j} \times Power_j, \ \ \forall m. \tag{7.12}$$

Finally, the temperature constraint is enforced using the following expression:

$$Temp_m \leq T_B, \ \ \forall m. \tag{7.13}$$

Having specified the necessary constraints in our ILP formulation, we next give our objective function. We define our cost function as the sum of the data communication distances in all 3 dimensions. X_{Cost}, Y_{Cost}, and Z_{Cost} denote the total data communication distances traversed along dimensions x,y, and z, respectively. The communication cost due to dimension x is:

$$X_{Cost} = \sum_{i=1}^{P} \sum_{j=1}^{M} \sum_{k=1}^{C_X - 1} FREQ_{i,j} \times Xdist_{i,j,k} \times k. \tag{7.14}$$

Similarly, the cost due to dimension y can be expressed as:

$$Y_{Cost} = \sum_{i=1}^{P} \sum_{j=1}^{M} \sum_{k=1}^{C_Y - 1} FREQ_{i,j} \times Ydist_{i,j,k} \times k. \tag{7.15}$$

Table 7.3 Single-core benchmark codes used in this study.

Benchmark Name	Source	Description	Number of Data Accesses
ammp	Spec	Computational Chemistry	86967895
equake	Spec	Seismic Wave Propagation Simulation	83758249
mcf	Spec	Combinatorial Optimization	114662229
mesa	Spec	3-D Graphics Library	134791940
vortex	Spec	Object-oriented Database	163495955
vpr	Spec	FPGA Circuit Placement and Routing	117239027

Table 7.4 Chip multiprocessor benchmark codes used in this study.

Benchmark Name	Source	Description	Number of Data Accesses
3step-log	DSPstone	Motion Estimation	90646252
adi	Livermore	Alternate Direction Integration	71021085
btrix	Spec	Block Tridiagonal Matrix Solution	50055611
tsf	Perfect Club	Nearest Neighbor Computation	54917732

Finally, the cost due to dimension z can be written as:

$$Z_{Cost} = \sum_{i=1}^{P} \sum_{j=1}^{M} \sum_{k=1}^{C_Z - 1} FREQ_{i,j} \times Zdist_{i,j,k} \times k. \tag{7.16}$$

Consequently, our objective function can be expressed as:

$$\min \quad (\alpha \times (X_{Cost} + Y_{Cost}) + \beta \times Z_{Cost}). \tag{7.17}$$

To summarize, our topology exploration problem can be formulated as "minimize $\alpha \times (X_{Cost}+Y_{Cost}) + \beta \times Z_{Cost}$ under constraints (7.1) through (7.16)." It is important to note that this ILP formulation is very flexible as it can accommodate different number of processor cores, storage blocks, and layers.

Table 7.5 The default simulation parameters.

Parameter	Value
Number of processor cores (in chip multiprocessor designs)	4
Number of blocks	24
Number of layers	2
$\frac{\alpha}{\beta}$	10
Total storage capacity	128KB
Set associativity	2 way
Line size	32 Bytes
Number of lines per block	90
Temperature bound	110°C

7.3 Example

In this section, we give an example demonstrating the effectiveness of our approach. As our example, we use 3step-log, one of our benchmarks, with 4 processor cores and 20 storage blocks. In the 2D case, we assume the dimensions of the chip (C_X, C_Y and C_Z) as $12 \times 8 \times 1$, and in the 3D case, we assume the dimensions to be $8 \times 6 \times 2$. In both the cases, the total number of coordinates (total chip area) is equal to 96. Both the storage blocks and the processor cores are assumed to be 2×2, though they can be set to any value in our formulation. Table 7.2 shows the percentage of accesses to each block by different processor cores.

In Fig. 7.3, the topologies generated by three different schemes are depicted. Fig. 7.3(a) illustrates the best 2D placement (determined using ILP). On the other hand, a randomly-generated storage block/processor core placement is shown in Fig. 7.3(b) for 3D. Finally, Fig. 7.3(c) shows the best placement for 3D (determined using ILP). Note that, each processor core and each storage block is mapped to 4 coordinates since the size of a storage block/processor core is assumed to be 4. To explain these mappings, let us consider Fig. 7.3(c). In this topology, B_{15} is mapped to coordinates (0,0,0), (0,1,0),

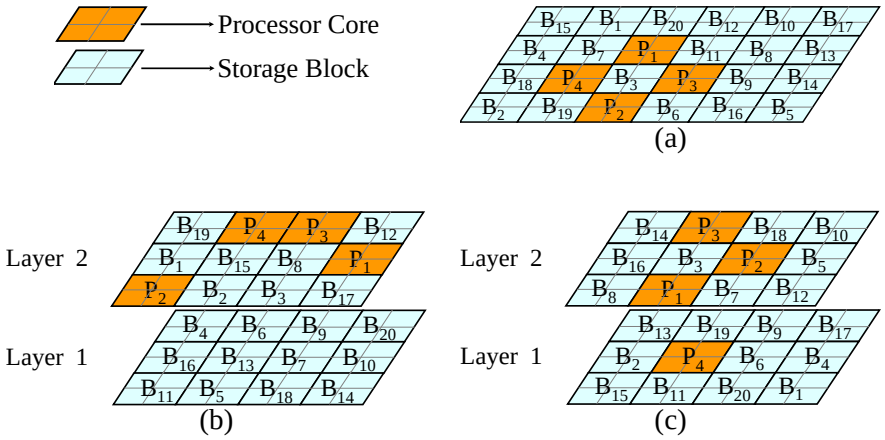


Fig. 7.3 The topologies generated with 4 processor cores (3step-log). (a) Optimal 2D. (b) Random 3D. (c) Optimal 3D. $P_1 \ldots P_4$ are processor cores and $B_1 \ldots B_{20}$ are storage blocks.

(1,0,0), and (1,1,0). As it can be seen from Table 7.2, data block B_3 is the most frequently accessed block by all the processor cores, and that is why it is put into a very close position to all 4 processor cores in optimal 2D and optimal 3D. We also see that, B_{11} is the next most frequently accessed block by all the processor cores in this example.

7.4 Experimental Evaluation

7.4.1 Setup

We performed several experiments with two different set of benchmarks. The first group of applications are sequential Spec2000 codes and used for single-core designs in this work. This benchmark set consists of six applications randomly-selected from the Spec2000 benchmark suite. Table 7.3 lists these codes and their important characteristics. In collecting the statistics on accesses to storage blocks, for each benchmark, we fast-forwarded the first 2 billion instructions, and simulated the next 300 million instructions.

The fourth column of Table 7.3 gives the number of data accesses for each application. The second group of applications, on the other hand, are for chip multiprocessor designs and include several array-based codes parallelized through an optimizing compiler built upon SUIF [156]. The exact mechanism used in parallelizing the code is orthogonal to the focus of this study. What we mean by parallelization in this context is distributing loop iterations across the processors; each processor is typically assigned a subset of loop iterations. Table 7.4 lists the important characteristics of this second group of benchmarks. As before, the third column gives the description of the benchmarks, and the last column gives the number of data accesses. The ILP solution times on an Intel Pentium III Xeon Processor of 549MHz with 1GB of RAM varied between 144 seconds and 3.5 hours, averaging on about 25 minutes. The default simulation parameters used in our experiments are presented in Table 7.5. As our base configuration, we assumed a stack of two device layers connected to each other. We also assumed that the chip is composed of 24 blocks, each of which can be a storage block or a processor core. We conservatively assumed that the block-to-block distance is ten times costlier than that of the layer-to-layer distance. This is denoted as the ratio of $\frac{\alpha}{\beta}$ in this work. Later in this chapter, we modify the values of some of our simulation parameters to conduct a sensitivity analysis.

We performed experiments with four different execution models for each benchmark code in our experimental suite:

• *2D-Random:* This is in a sense a conventional topology which uses a single wafer and the storage blocks and the processor cores are placed randomly.

• *2D-Opt:* This is an integer linear programming based strategy, wherein the storage blocks and processor cores are distributed on the die in a way that minimizes the data

communication cost of the whole system. Note that, this is an optimal core/storage block placement scheme for 2D.

• *3D-Random:* This is same as the 2D-Random case except that there are possibly multiple device layers.

• *3D-Opt:* This is the integer linear programming based placement strategy for 3D proposed in this chapter, wherein the storage blocks and processor cores are placed on several wafers optimally. This scheme represents the optimal placement for 3D.

7.4.2 Results

Fig. 7.4 gives our results based on two layers. Note that, these results are obtained using the values given in Table 7.5. All results are normalized with respect to those of the 2D-Random scheme. Fig. 7.4(a) shows the improvement brought by our approach for the single-core designs, whereas Fig. 7.4(b) gives the similar results for the chip multiprocessor designs. We see that the overall average reduction in data access costs with 2D-Opt is around 63% and 58% for the single-core case and the chip multiprocessor case, respectively. On the other hand, the 3D-Opt scheme reduces the costs by about 82% and 69% on average for the single-core case and the chip multiprocessor case, respectively. In other words, the best 3D design generates much better results than the best 2D design.

To see the impact of the optimal placement of processor cores/storage blocks, we next compare the optimized and random designs in 3D. In Fig. 7.5, we compare our 3D approach against the randomly-generated placements obtained through 3D-Random. As before, the results are normalized with respect to the 2D-Random scheme. The average data communication cost reductions are 42% and 51% for the single-core case and for

the chip multiprocessor case, respectively. Overall, the results presented in Fig. 7.4 and Fig. 7.5 clearly show that employing a 3D design with optimal placement is critical for the best results.

7.4.3 Sensitivity Study

In this subsection, we modify some of the default simulation parameters (see Table 7.5), and conduct a sensitivity study. While we focus only on some of the benchmarks, our observations extend to the remaining benchmarks as well. Recall that the original number of 3D layers we used were two. The bar-chart in Fig. 7.6(a) shows the normalized costs (with respect to those of the 2D-Random scheme) for the benchmark ammp with the different number of layers (the results with the original number of layers are also shown for convenience), ranging from 1 to 4. Note that, the total storage capacity is kept constant for all these experiments and the only difference between two experiments is the number of layers and size of each layer. The number of layers and the corresponding number of blocks per layer for each topology tested are given in Table 7.6. We see from these results that the effectiveness of our ILP-based approach increases with increasing number of layers. The main reason for this behavior is that adding more layers gives more flexibility to our approach in placement. Recall that all the chip multiprocessor results reported so far were obtained using 4 processor cores. Fig. 7.6(b) shows the normalized cost of 3D-Opt with respect to 2D-Opt for the benchmark 3step-log with the different number of processor cores (2 through 4). One can observe from these results that our approach generates significantly better results than the 2D-Opt case with all the number of processors tested. In our last set of experiments, we measure the impact of the temperature constraint on

Table 7.6 Different topologies.

Number of Layers	Number of Blocks per Layer
1	24
2	12
3	8
4	6

our savings. Recall from Table 7.5 that the default temperature bound used in our experiments so far was 110°C. The bar-chart in Fig. 7.6(c) shows the normalized costs for the benchmark ammp with the different temperature bounds, ranging from 80°C to 110°C. Note that, the values given in this graph are normalized with respect to the default 3D-Opt case, where we obtain the best results. As we can see from this graph, having a tighter temperature bound reduces our savings beyond a point. The reason for this behavior is that decreasing the temperature bound also decreases the flexibility in storage block and processor core assignment. For this particular example, reducing the temperature bound below 80°C did not return any feasible solution.

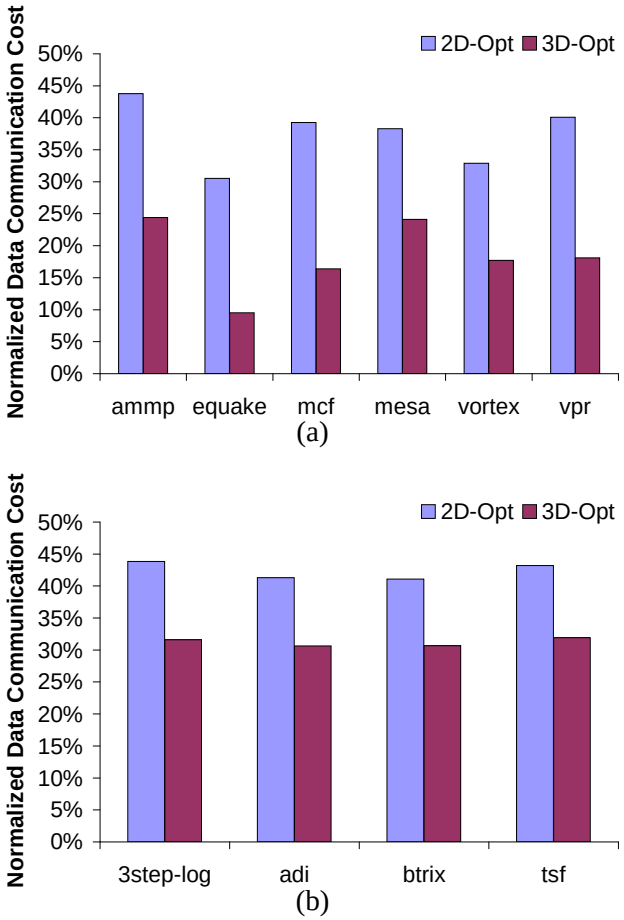


Fig. 7.4 Data communication costs for 2D-Opt and 3D-Opt normalized with respect to the 2D-Random scheme. (a) Single-core design. (b) Chip multiprocessor design (with 4 processor cores).

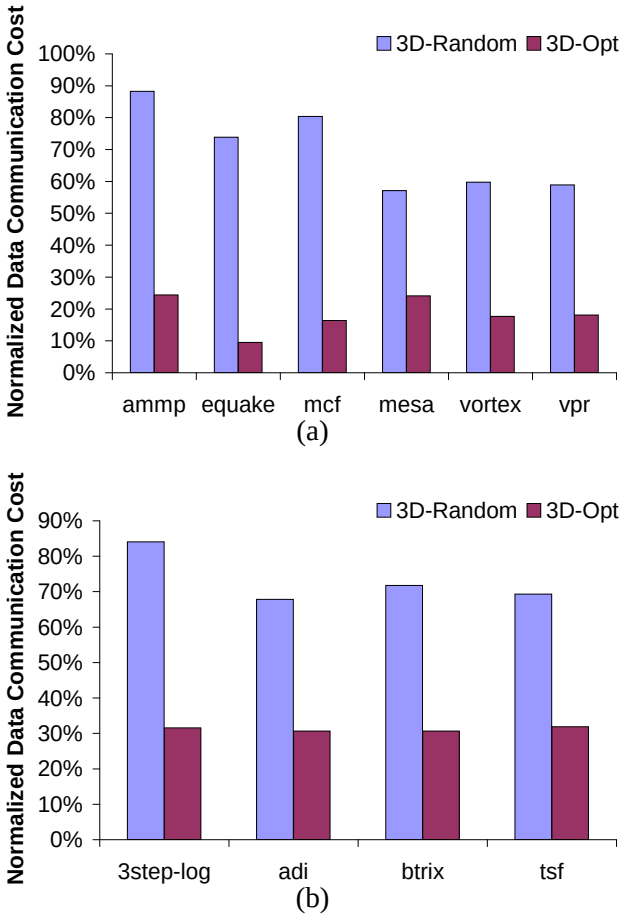


Fig. 7.5 Data communication costs for 3D-Random and 3D-Opt normalized with respect to the 2D-Random scheme. (a) Single-core design. (b) Chip multiprocessor design (with 4 processor cores). The results of 3D-Random are obtained by taking the average over five different experiments with random placement.

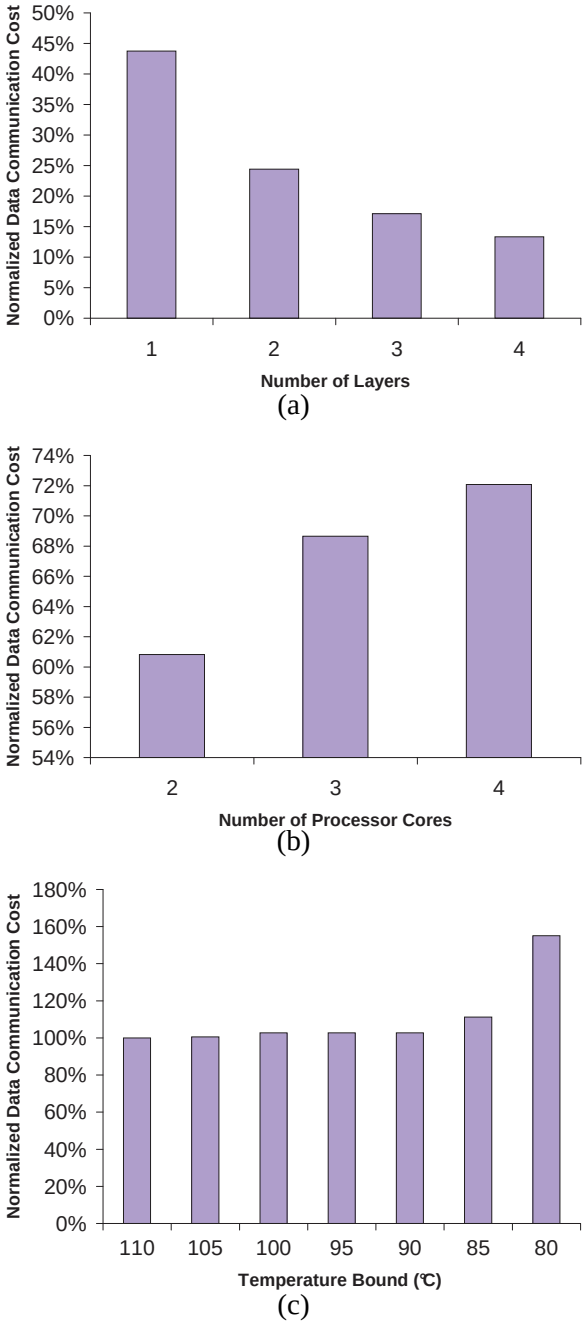


Fig. 7.6 Normalized data communication costs. (a) With the different number of layers (ammp). (b) With the different number of processor cores (3step-log). (c) Under the different temperature bounds (ammp). The normalized results are with respect to 2D-Random, 2D-Opt, and 3D-Opt in (a), (b), and (c), respectively.

Chapter 8

Shared Scratch-Pad Memory Space Management

Increasing latency gap between CPU operations and memory accesses makes it increasingly important to develop strategies for cutting the frequency and volume of data transfers between on-chip storage and off-chip storage. While conventional cache memories are an effective way of reducing the cost of memory accesses, their behavior depends on several runtime factors and thus they may not be the best choice for embedded systems where execution latency guarantees are demanded. In addition, recent research shows that caches result in suboptimal behavior for multi-media applications with regular data access patterns. These inefficiencies of conventional caches in the context of embedded computing motivated designers for adopting software-managed on-chip memories. Scratch-Pad Memories (SPMs) are one such option where software (user application or compiler) explicitly manages memory activity. SPMs are similar to conventional on-chip caches in that they reside on-chip and have low access latency/power consumption; however, unlike conventional caches, their contents are managed by software. Therefore, they are very useful in execution scenarios with hard real-time bounds and in embedded systems that execute multi-media applications.

Most of the prior efforts on data SPMs [11, 15, 16, 54, 76, 97, 133, 143, 153, 154] have mainly focused on single application scenarios, i.e., the SPM space available is assumed to be managed by a single application at any given time. While this assumption

makes sense in certain application domains (e.g., a dedicated embedded core that executes a microcode in an automobile), there also exist many cases where multiple applications need to share the same SPM space. Maybe the most important issue in this context is deciding a suitable partitioning and management of the available on-chip SPM space across the concurrently executing applications. This chapter addresses this important question and proposes a space partitioning and management scheme for shared SPMs. An important characteristic of this scheme is that it captures, using automated compiler analysis, the inherent SPM space requirements of applications at a loop granularity, and uses this information in partitioning the available SPM space across applications in a nonuniform fashion at runtime. What we mean by nonuniformity in this context is that two applications can have different amounts of SPM space if their compiler-determined SPM requirements differ.

This approach is implemented within an optimizing compiler infrastructure and our experiments compare the power/performance of this approach to an alternate approach that divides the available SPM space equally among the concurrent applications (i.e., uniform partitioning). The results from our experiments show that nonuniform partitioning generates much better runtime behavior than uniform partitioning for all the execution scenarios evaluated, and the magnitude of our savings increases with smaller SPM spaces (as compared to total size of the data manipulated by the applications) and larger number of concurrently-executing applications.

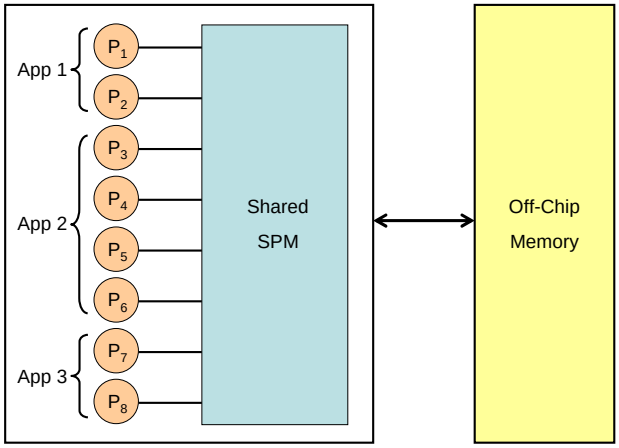


Fig. 8.1 Architectural abstraction.

8.1 Architectural Abstraction

The architectural abstraction we assume in this work contains a number of processor cores and an SPM space to store data. The SPM space is shared by all concurrently-executing applications. We also assume the existence of a large off-chip memory space. Each application in this system is mapped onto a set of processor cores (Fig. 8.1 depicts an example for a scenario with three applications). While the number of cores to use for a given application is certainly important and deserves research, our goal in this chapter is rather to study the SPM space partitioning across the applications. Therefore, in our experiments, we assume that each processor executes a single application at any given time. Processors can potentially have their own caches (or even a shared cache space) as well; but, in this work, we concentrate our attention on the management of the shared on-chip SPM space.

Our focus is on array/loop-intensive applications from the domain of embedded image and video processing. This represents a large segment of data-dominated embedded applications [28]. We also assume that the applications do not share data among

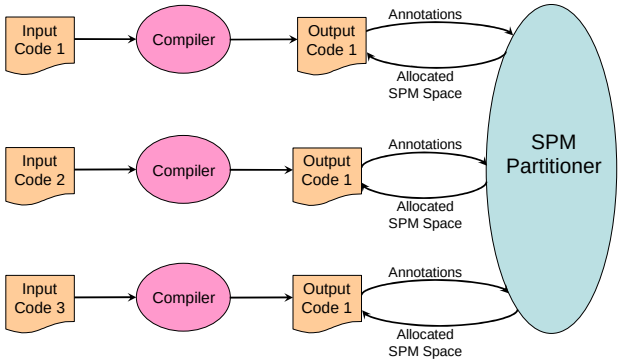


Fig. 8.2 High-level view of our approach.

them. While our approach still works if they share data, the results may not be as good. Extending our approach to capture inter-application data sharing is in our future agenda.

8.2 High Level View Of The Approach

Fig. 8.2 depicts the major components of the proposed approach. The job of the compiler is to analyze each application code and determine the data reuse characteristics of individual loop nests. This data reuse information is then added at the beginning of loops as annotations, which are passed to our SPM partitioner at runtime. SPM partitioner on the other hand implements an algorithm for dynamic allocation of the shared SPM space at runtime. It achieves this by interpreting the annotations inserted by the compiler and taking into account the available SPM space.

Fig. 8.3 illustrates the details of application-SPM partitioner interaction from a single application's point of view. In Fig. 8.3(a), SPM-Demand indicates the SPM space required for the application (obtained from the annotations embedded by the compiler). SPM partitioner receives this request by the Receive-Request directive. The SPM space allocated for this application (based on the requirements of the competing applications) is

sent to the application by the Send-Space command. This space information is received using the Receive-Space directive by the application, which in turn partitions this allocated space among its data elements. Similarly, Fig. 8.3(b) shows how an application releases the allocated SPM space. SPM-Release releases the allocated SPM space and this information is received by the SPM partitioner via the Receive-Release command. In the examples in this chapter, we do not go into the details of how an allocated SPM space is released.

Let us illustrate the working of this approach informally using a simple example. We assume initially three applications are scheduled to be executed in this architecture. The first one starts executing, and while it is in the middle of its second loop nest, the second application starts executing, at which point the SPM partitioner divides the available SPM space between the two. While first two applications are running, a third one starts executing. At this point, our SPM partitioner re-distributes (re-partitions) the available SPM memory space among these three concurrently-executing applications. When the first and second applications finish their executions and terminate, the SPM spaces allocated to these applications are returned to the SPM partitioner. Whole SPM space, now, can be allocated for the only running application in the system (the third one). The rest of this chapter gives details on how our annotations are generated, and how the proposed SPM partitioner operates.

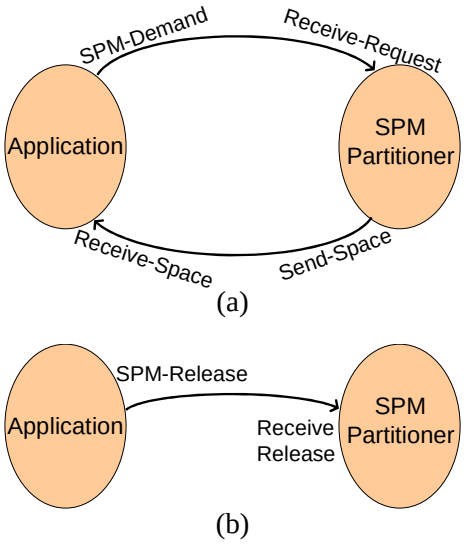


Fig. 8.3 Interaction between an application and our SPM partitioner.

8.3 Technical Details of The Approach

8.3.1 Preliminaries

The iteration space, $\mathcal{I}$, of a loop nest contains a point for each iteration. An iteration vector can be used to label each such point in $\mathcal{I}$. We use $\vec{i} = (i_1\ i_2\ \ldots\ i_n)^T$ to represent the iteration vector for a loop nest of depth n, where each i_k corresponds to a loop index (or its value at a specific point in time), starting with i_1 for the outermost loop index. An iteration space $\mathcal{I}$ can be viewed as a polyhedron bounded by the loop limits.

The subscript function for a reference to an array X_j is a mapping from the iteration space $\mathcal{I}$ to the data space $\mathcal{D}$. The data space can also be viewed as a polyhedron bounded by array bounds. A subscript function defined this way maps an iteration vector to an array element. In this work, we assume that subscript mappings are affine functions of the enclosing loop indices and symbolic constants. Many array references found

in embedded image and video processing codes fall into this category [28]. Under this assumption, a reference to an array X_j can be represented as $Q_j\vec{i} + \vec{q_j}$, where Q_j is a linear transformation matrix called the access (reference) matrix, $\vec{q_j}$ is the offset vector, and $\vec{i}$ is the iteration vector [159]. If the loop nest in question has n loops and array X_j is m-dimensional, Q_j is $m \times n$ and $\vec{q_j}$ is an m-element vector. For example, the reference $X_1[i_1][i_2]$ in Fig. 8.4(a) can be written as $Q_1\vec{i} + \vec{q_1}$, where $\vec{q_1} = (0\ 0\ 0)^T$ and

$$Q_1 = \begin{pmatrix} 1 & 0 & 0 \\ \\ 0 & 1 & 0 \end{pmatrix}.$$

According to Li [88], if $\vec{i}$ and $\vec{j}$ are two iteration vectors that access the same memory location, $\vec{r} = \vec{j} - \vec{i}$ is called a reuse vector (assuming that $\vec{j}$ is lexicographically greater than $\vec{i}$). For a loop nest of depth n, the reuse vector has n entries. The loop corresponding to the first non-zero element from top is the one that *carries* the corresponding reuse. If there are more than one reuse vector in a given direction, they are represented by the lexicographically smallest one. The reuse vector gives us useful information about how array elements are used by different loop iterations in a given nest. In this work, we mainly focus on self reuses (that is, the reuses originating from individual references) as in very rare circumstances group reuse (that is, the reuse between different references to the same array) brings additional benefits that cannot be exploited by self reuse [88]. There is a temporal reuse due to reference $Q_i\vec{i} + \vec{q_i}$ (of array X_i) if and only if there exist two different iteration vectors $\vec{i}$ and $\vec{j}$ such that $Q_i\vec{i} + \vec{q_i} = Q_i\vec{j} + \vec{q_i}$; that is, $\vec{j} - \vec{i} \in Ker\{Q_i\}$. This

last expression indicates that the temporal reuse vector $\vec{r} = \vec{j} - \vec{i}$ belongs to the kernel set (null set) of Q_i [159].

As an example, let us focus on the matrix multiply nest shown in Fig. 8.4(a). The reuse vector due to $X_2[i_1][i_3]$ is $(0\ 1\ 0)^T$. What this means is that, for fixed values of loops i_1 and i_3, the successive iterations of the i_2 loop access the same element from this array. That is, the i_2 loop carries (exhibits) temporal reuse for this reference.

The reuse vectors coming from individual references make up a reuse matrix, R [88]. In considering the matrix multiply code in Fig. 8.4(a) again, we find that:

$$R = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

The first, second, and third columns in this matrix correspond to the reuse vectors due to references $X_1[i_1][i_2]$, $X_2[i_1][i_3]$ and $X_3[i_3][i_2]$, respectively. As will be discussed shortly, each column indicates which array sections (data blocks) should be brought into (and discarded from) the available SPM space at what time.

8.3.2 Compiler-Inserted Annotation and Code Restructuring

Recall that the reuse vector due to $X_2[i_1][i_3]$ in matrix multiply (Fig. 8.4(a)) is $(0\ 1\ 0)^T$. Suppose that the i^{th} row of $X_2[i][1:N]$ resides in the shared SPM, then the iterations (i,1,1), (i,1,2),..., (i,1,N), (i,2,1), (i,2,2),..., (i,2,N),....., (i,N,1), (i,N,2),..., (i,N,N) can access this same array section from the SPM (which is much faster than off-chip memory) during the entire execution of these iterations; and following the execution

of iteration (i,N,N), it can be discarded from the SPM. To effectively use the available shared SPM space, this entire row can be brought into the SPM (if there is available space and our SPM partitioner allows this) just before the iteration (i,1,1) starts execution, that is just before the j loop is entered. For clarity, we use (i,$*$,$*$) to collectively represent these iterations, where $*$ denotes all iterations in the corresponding loop level; we also use $X_2[i_1][*]$ to represent the array section (row), where $*$ indicates all elements in the corresponding dimension. To sum up, by just considering the value of 1 in the reuse vector given as $(0\ 1\ 0)^T$ (the second entry), we can decide the point in the code at which an array section (in our case, this is the i^{th} row of X_2) should be brought into the shared SPM and the point at which it should be removed from the shared SPM.

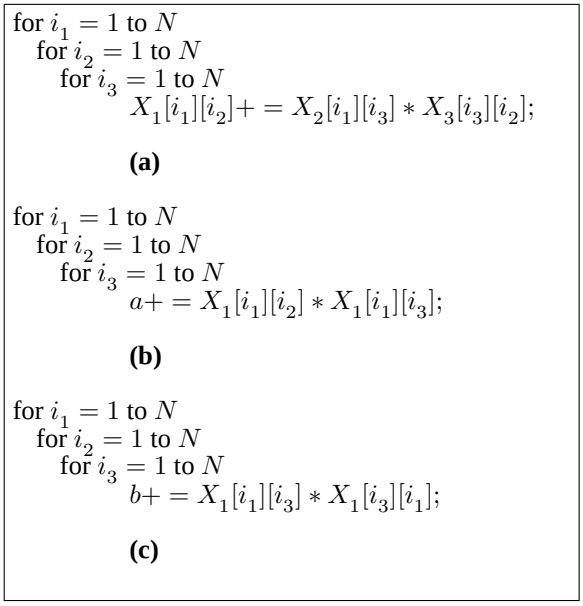


Fig. 8.4 Different code fragments.

This observation suggests that, for the best exploitation of data reuse, an array section should remain in the SPM, if possible, from the start of the loop that carries reuse for the corresponding array reference to the termination of the same loop. For example, if we consider the following generic n-deep loop nest that accesses an m-dimensional array X_i:

for $i_1 = {i_1}^L$ to ${i_1}^U$
  for $i_2 = {i_2}^L$ to ${i_2}^U$
    ...
      for $i_n = {i_n}^L$ to ${i_n}^U$
        $X_i[f_1(i_1, i_2, \ldots, i_n)][f_2(i_1, i_2, \ldots, i_n)] \ldots [f_m(i_1, i_2, \ldots, i_n)];$

In this example nest, each subscript expression is expressed as an affine function given in the form of $f_j(i_1, i_2, \ldots, i_n)$, where $1 \leq j \leq m$. The lower and upper bounds of loop k are given by ${i_k}^L$ and ${i_k}^U$, respectively. Assume that reuse vector for this loop nest is $\vec{r} = (r_1\ r_2\ \ldots\ r_n)^T$ with a single non-zero element (say r_k) for now. That is, the reuse is carried by the i_k loop. Given this information, compiler can make use of the shared SPM space by bringing the array section represented by $A[f_1(i_1, i_2, \ldots, i_{k-1}, *, *, \ldots, *)]$ $[f_2(i_1, i_2, \ldots, i_{k-1}, *, *, \ldots, *)] \ldots [f_m(i_1, i_2, \ldots, i_{k-1}, *, *, \ldots, *)]$ just before executing loop i_k (if the SPM partitioner allows this). In the above array section, $f_l(i_1, i_2, \ldots,$ $i_{k-1}, *, *, \ldots, *)$ denotes all indices enumerated in a dimension l by fixing the first $k-1$ loops at specific values (i_1, i_2, ..., i_{k-1}). Hence, all values (within the loop bound ranges) for all other loop index positions are considered. This array section should normally be kept in the shared SPM (if there is sufficient space), while executing all the iterations specified by $(i_1, i_2, \ldots, i_{k-1}, *, *, \ldots, *)$. When the last iteration in this set has been executed, the section can be removed from the shared SPM. In this way, the mentioned set of

data elements would reside in the SPM as long as they are reused, and as soon as the last iteration that reuses the elements is completed, the set can be removed from the shared SPM. It should be emphasized that if any element in this set is updated while it is in the SPM, it should be written back.

If there are multiple references to the same array, the reuse vectors are considered together and the corresponding data transfers are combined as much as possible. If we consider the example given in Fig. 8.4(b), there are two reuse vectors for array X_1: $(0\ 1\ 0)^T$ (due to reference $X_1[i_1][i_3]$) and $(0\ 0\ 1)^T$ (due to reference $X_1[i_1][i_2]$). It should be noted that, the data transfer indicated by $(0\ 0\ 1)^T$ is subsumed by the transfer indicated by $(0\ 1\ 0)^T$; that is, once the transfer of the i^{th} row to the shared SPM is complete (just before entering the i_2 loop), there is no need for performing an extra transfer for the element $X_1[i_1][i_2]$. Let us now consider the code given in Fig. 8.4(c). The reuse vectors for this example loop nest are $(0\ 1\ 0)^T$ and $(0\ 1\ 0)^T$, both of which require the data transfers to be performed before the j loop is entered. However, the data transfers required for the reuses are not same; one contains the i^{th} row of the array whereas the other contains the i^{th} column. These two transfers overlap only for one element, meaning that they should be performed separately. In order to decide whether the transformations required by different references to the same array can be combined, we use the following rule. Let $Q_1\vec{i} + \vec{q_1}$ and $Q_2\vec{i} + \vec{q_2}$ be two references to the same array. If $Q_1 = Q_2$, there is a good chance that there exists significant amount of data reuse between these two references even if $\vec{q_1} \neq \vec{q_2}$. In this case, these data transfers can be combined and performed together. However, if $Q_1 \neq Q_2$, these two references are treated as if they belong to different arrays.

Once the compiler analyzes an application code and determines the data reuse characteristics of its individual loop nests as explained above, this reuse information (capturing the set of elements that are to be brought to the SPM, and those to be discarded from the SPM) is added at the beginning of loop nests as *annotation*. For example, in Fig. 8.5, three applications (A, B, and C) are shown with the corresponding annotations. Specifically, in the first loop nest of application A, the data requirements (i.e., the elements that are to be brought into the SPM) are specified as $L[i_1]$ and $K[i_1][*]$. This is indicated by the compiler annotation #SPM-Demand($L[i_1]$, $K[i_1][*]$). However, since SPM is a shared resource, we cannot just give an application all the space it demands. Instead, the compiler-inserted annotations are passed to the SPM partitioner (via the SPM-Demand command) and used by it to allocate the suitable amount of space computed by Algorithm 1 shown in the next section.

8.3.3 Runtime Support and Example Execution Scenario

The main task of the runtime system in this work is to partition the available shared SPM space across the competing (concurrently-executing) applications. Note that, while annotations indicate the amount of SPM space required, since SPM space is a shared resource, we need to partition it between the competing applications carefully. Since our SPM partitioner is invoked and executed at runtime, it should be very fast in performing this memory space partitioning. It operates in two steps. In the first step, it determines the applications' memory requirements from annotations, and in the second step, it allocates the available SPM memory space to the applications. The SPM space allocated to each application is then divided amongst the designated array portions of the application based

on their frequency of use. This frequency information can be extracted by the compiler analysis described in Section 8.3.2.

Algorithm 1 gives the sketch of our runtime partitioning algorithm (used by SPM partitioner) for determining the SPM space to be allocated to each application. This algorithm takes the SPM size (S) and the number of applications currently running (A) as input. Based on the data requirements of each application (obtained by the function call ReceiveRequest(i)), it calculates the total SPM space requirement ($Total$) using the while loop in lines 1 through 5. If the total (request size) obtained in the previous step is less than the available SPM size (S), all of the applications are granted with the SPM space they have requested (lines 6 through 10), i.e., in this case, we do not have a space partitioning problem. If not, this means that the available SPM space is not sufficient to accommodate the space requirements of all concurrently-running applications. In this case, the runtime system distributes the available SPM space among these applications proportionally with their request sizes (lines 11 through 16). That is, each application gets the $(Request(i)/Total)$ of the SPM space.

Once an application knows the amount of SPM space it is given by the partitioner (through ReceiveSpace), it somehow needs to distribute this space among its competing data elements (indicated by the annotation). For this purpose, each application executes Algorithm 2.

In other words, Algorithm 2 is used, for each application, to distribute the granted space by the runtime system. Based on the amount of the memory space that could be allocated for this application, the array sections are placed in the SPM starting from the

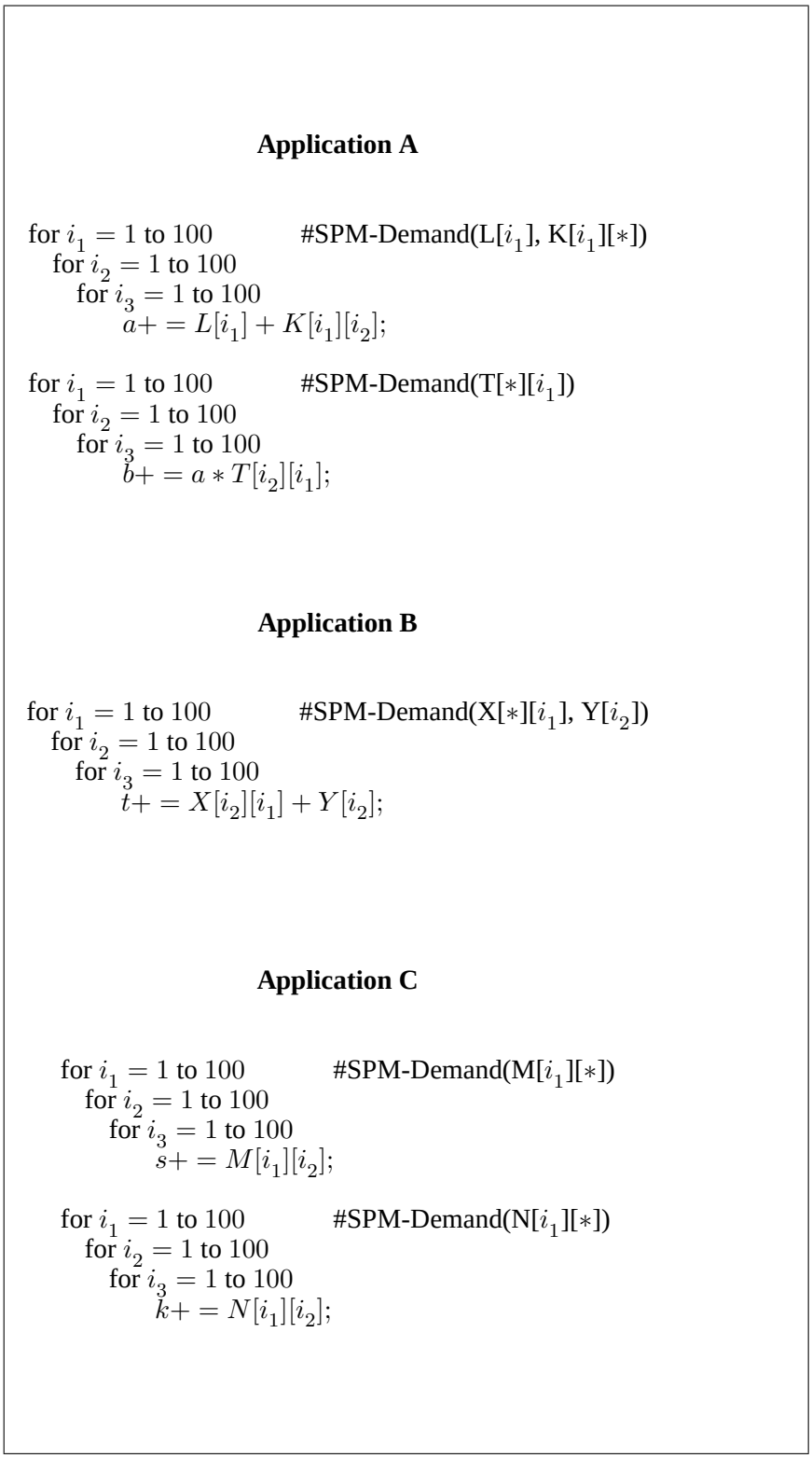


Fig. 8.5 An example scenario.

Algorithm 1 $SPMpartitioner(S, A)$

```
1: while i ≤ A do
2:    Request(i) = ReceiveRequest(i)
3:    Total += ReceiveRequest(i)
4:    i++
5: end while
6: if Total ≤ S then
7:    while i ≤ A do
8:       SendSpace(i,Request(i))
9:       i++
10:   end while
11: else
12:   while i ≤ A do
13:      SendSpace(i,(Request(i)/Total) × S)
14:      i++
15:   end while
16: end if
```

most frequently used ones. In this algorithm, D_i[j] corresponds to the j^{th} array section, with D_i[j].Size and D_i[j].Loc being the size and location (SPM or off-chip) of the corresponding section. Once the SPM space allocated for the application is distributed across the frequently used array elements, the remaining array sections are marked to be accessed from the off-chip memory (lines 8 through 11 in Algorithm 2). Note that the array elements that appear in annotations but cannot be placed into the shared SPM affect performance negatively.

Let us now trace the execution of an example scenario in detail. We assume, for illustrative purposes that, three applications, A, B, and C (see Fig. 8.5), are to be run on a system with 600 bytes of shared SPM space. The execution progress of these applications (in terms of iterations) is shown in Fig. 8.6. First, A starts executing, and when it is done with its first nested loop, B starts executing. That is, the second loop nest of A and the loop nest of B starts executing simultaneously. In terms of the number of iterations, A first executes for 1 million iterations (1M) during the first loop nest shown in Fig. 8.6. Since

Algorithm 2 $Compiler - Assignment$

```
1: D_i = Data requirement of application i
2: M=ReceiveSpace(i,|D_i|)
3: T=0
4: while T + D_i[j].Size < M do
5:    D_i[j].Loc=SPM
6:    T += D_i[j].Size
7:    j++
8: end while
9: while j < D_i.count do
10:   D_i[j].Loc=OffChip
11:   j++
12: end while
```

A is the only application running in the system at this point, the whole SPM space can be used by A. Based on the annotations inserted by the compiler, the data requirements for A are $L[i_1]$ and $K[i_1][*]$ for this loop nest which require 404 bytes (assuming that an integer is 4 bytes), and this requirement fits completely in the available SPM space. When B starts executing, the SPM partitioner divides the available shared SPM space between the two applications. Between iterations 1M and 2M, the data requirements are: A (400 bytes): $T[*][i_1]$ and B (404 bytes): $X[*][i_1]$, $Y[i_2]$. Therefore, the SPM space allocated for applications A and B are 299 and 301 bytes, respectively. If we assume, for simplicity, that all applications execute iterations at the same speed, C starts executing when A is at 1.5M iterations and B is at 0.5M iterations. As before, the available SPM space needs to be divided now among A, B and C by the SPM partitoner proportionally when C starts execution. The data requirement of C for this period (iterations 1.5M-2M) is $M[i_1][*]$. The data requirements for A and B remain the same, and hence, SPM partitioner allocates 200 bytes to each of these applications. At iteration point 2M, A and B terminate, and beyond that point, the whole shared SPM space is allocated for the use of C since it is the only application still running.

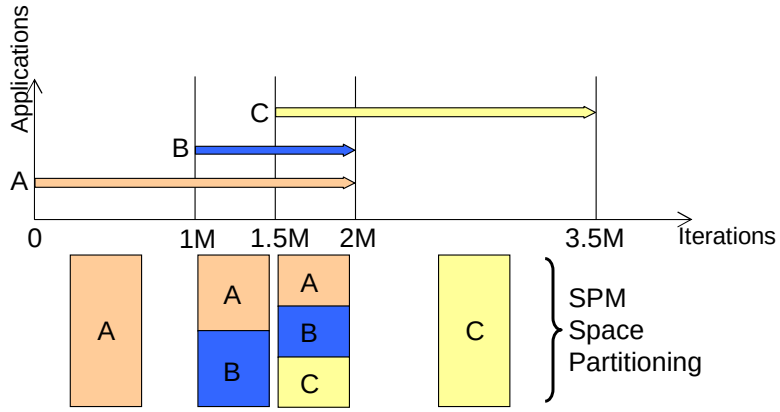


Fig. 8.6 Execution of an example scenario.

8.4 Experimental Analysis

Using a Simics based simulation environment [96], we implemented and simulated four different on-chip memory management schemes. An important feature of Simics is that it allows us simulate the executions of multiple applications simultaneously. The default values of our simulation parameters are presented in Table 8.1. The schemes simulated can be summarized as follows. *Uniform-Cache* divides the on-chip memory space across the concurrently running applications equally, and each portion is assigned to an application and managed as a conventional 4-way set-associative cache memory. Each time a new application is introduced to the system, we perform a new cache partitioning. *Nonuniform-Cache* performs a nonuniform partitioning across the applications, and as in the previous scheme, each portion is managed as a conventional cache. The reason that we make experiments with these two versions is to see how much we lose if we do not software-manage the shared on-chip memory space. The next two schemes are the ones discussed in this work. *Uniform-SPM* treats the on-chip storage as an SPM and divides its capacity equally across the concurrently-running applications. Finally, *Nonuniform-SPM*

Table 8.1 Default simulation parameters.

Parameter	Value
On-Chip Storage Capacity	64KB
Processor Count	8
Number of Ports	3
On-Chip Latency/Off-Chip Latency	1/50

partitions the memory space based on inherent space requirements of the concurrently-running applications. This is the scheme defended in this chapter.

The set of applications used in our experiments are given in Table 8.2. The second column describes each application briefly; the third column lists the number of arrays/loop nests each application has; and the fourth column gives the total number of data accesses made by each application. Since our objective is to study the runtime behavior of multiple applications, we use the *workload completion time* as our main metric. In this discussion, workload captures a set of concurrently-executing applications and completion time is the time the last application in the workload finishes its execution. The data transfers between the off-chip memory and the SPM are captured within this completion time as well. Let us first consider the workloads whose descriptions are given in Table 8.3. The second column gives the applications in each workload, and the third column gives the completion times (defined above) under the *Uniform-Cache* scheme.

Fig. 8.7 gives the workload completion times for the different memory management schemes. For each workload, the completion time of the *Uniform-Cache* scheme is set to 100 (see the last column of Table 8.3 for the absolute values), and the results of the remaining three schemes are *normalized* with respect to that. We can see that the percentage completion time savings brought by the schemes *Nonuniform-Cache, Uniform-SPM*

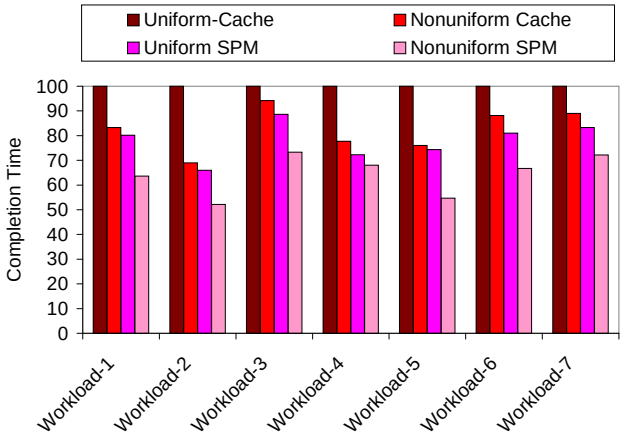


Fig. 8.7 Workload completion times for the different memory management schemes.

Table 8.2 Set of applications used in this study.

Application	Description	Number of Arrays/Nests	Number of Data Accesses
`Vcap`	Video Capture and Processing	8/18	110.4M
`Convolution`	Convolution Filter Implementation	6/13	83.4M
`TM`	Image Conversion	6/11	126.1M
`IA`	Target Detection and Classification	11/27	172.8M
`H.263`	H.263 Decoder Implementation	19/57	336.7M
`ImgMult`	Image Processing	3/8	68.2M
`Face`	Face Detection	10/44	294.2M

and *Nonuniform-SPM* over the *Uniform-Cache* scheme are 17.51%, 22.04% and 35.61%, respectively. That is, the software-managed uniform memory partitioning does not bring significantly better results than the nonuniform conventional cache management. However, the approach proposed in this work, *Nonuniform-SPM*, brings much more benefits than all the other schemes tested. Overall, these results emphasize the importance of partitioning the available shared SPM space across the competing applications nonuniformly, based on their space requirements.

Table 8.3 Example workloads, their contents, and their completion times under the Uniform-Cache scheme.

Workload	Contents	Completion Time
1	Vcap+Convolution+TM	324.6msec
2	Vcap+TM+IA	294.4msec
3	Convolution+H.263+Face	606.1msec
4	TM+IA+H.263	727.2msec
5	IA+H.263+ImgMult	753.9msec
6	ImgMult+Face+Vcap	418.5msec
7	ImgMult+IA+Convolution	360.1msec

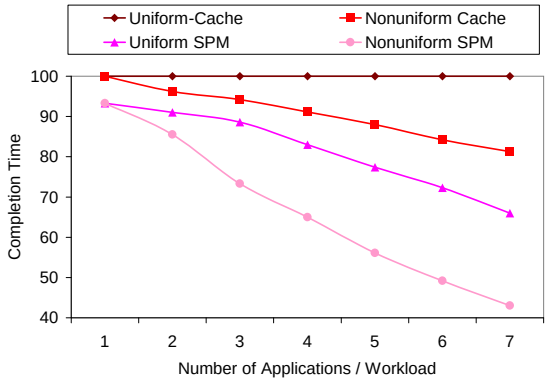


Fig. 8.8 Workload completion times for the different number of applications in a workload.

Fig. 8.8 helps us study the behavior of the schemes tested when the number of applications in a workload is changed. Each point on the x-axis corresponds to a different number of applications per workload. The contents of these workloads are given in Table 8.4. As before, the last column gives the completion time under the *Uniform-Cache* scheme, against which the results to be presented are normalized. We can see that the difference between the schemes *Uniform-SPM* and *Nonuniform-SPM* increases as we increase the number of applications in the workload. This is because the importance of accounting for the individual SPM space requirements of the applications is more critical when we have a larger set of applications (which is expected to be the case in the future).

Table 8.4 Workloads with different number of applications.

Workload	Contents	Completion Time
1	Convolution	286.5msec
2	Convolution+H.263	437.0msec
3	Convolution+H.263+Face	606.1msec
4	Convolution+H.263+Face+TM	722.7msec
5	Convolution+H.263+Face+TM+IA	840.7msec
6	Convolution+H.263+Face+TM+IA+ImgMult	892.6msec
7	Convolution+H.263+Face+TM+IA+ImgMult+Vcap	961.5msec

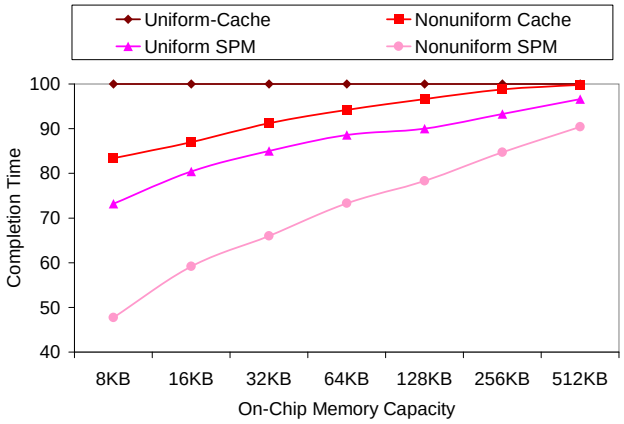


Fig. 8.9 Completion times with the different on-chip memory capacities.

Our next set of experiment study the behavior of these four schemes with different on-chip memory capacities, and the results are presented in Fig. 8.9 for the third workload shown in Table 8.3. While the relative (completion time) savings brought by the *Nonuniform-SPM* scheme get reduced with increasing on-chip memory capacity, we still see substantial savings even with large memory capacities. For example, with the shared SPM capacities of 128KB and 256KB, the reductions achieved by the *Nonuniform-SPM* scheme over the *Uniform-SPM* scheme are about 13% and 9%, respectively.

Our last experiments investigate whether significant variances occur in our savings when the applications are introduced to the system one after another, instead of all-at-once, which was the scenario under consideration so far. To test this, we performed

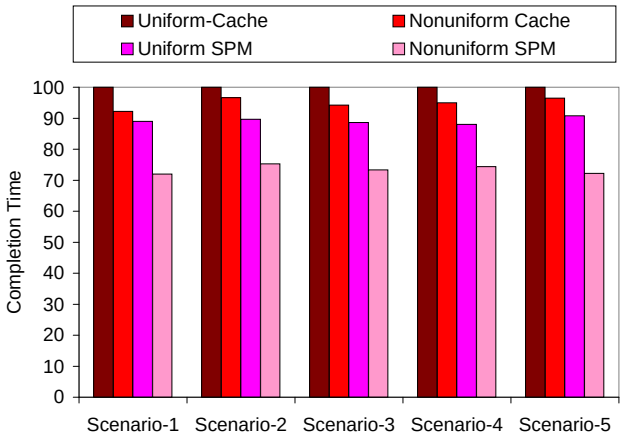


Fig. 8.10 Completion times with the different application introduction patterns.

experiments with five different scenarios with the third workload in Table 8.3. Each scenario introduces the applications to the system in a randomly-selected order. The workload completion results shown in Fig. 8.10 indicates that the relative behaviors of these schemes do not change significantly when the applications are introduced to the system in different patterns. In other words, our data reuse based SPM partitioning scheme is reasonably robust.

Chapter 9

Conclusions and Future Work

One of the most important issues in designing a chip multiprocessor is to decide its on-chip memory organization. A poor on-chip memory design can have serious power and performance implications when running data-intensive embedded applications (e.g., those from the domain of embedded image/video processing). In this thesis, we made several contributions to addressing this problem of memory space management and optimization. First, we proposed an integer linear programming (ILP) based solution to the combined problem of on-chip memory hierarchy design and data allocation across the components of the designed hierarchy. Second, we presented a compiler-directed approach that inserts compression and decompression calls in the application code to reduce maximum and average memory space consumption. Third, we proposed a constraint network (CN) approach to integrate loop and data optimizations to improve the locality. Fourth, we applied our constraint network (CN) based approach to code parallelization. Fifth, we proposed an integer linear programming (ILP) based processor core/storage block placement in a customized 3D design for chip multiprocessor embedded systems. Finally, we presented a SPM space management technique across the concurrently-executing applications. We measured the benefits of the proposed approaches by designing custom memory hierarchies for different applications. Our experimental results revealed that the proposed approaches are effective in optimizing memory behavior of embedded applications. Our

results also showed that the achieved benefits are consistent across a wide range of values of our major simulation parameters.

Our future work includes (1) latency-aware SPM management and (2) applying the proposed optimizations on heterogeneous chip multiprocessors. Below, we discuss details of these projects.

- **Latency-Aware SPM Management:**

  Nanometer design technologies must work under tight operating margins, and are therefore, highly susceptible to any process and environmental variabilities. In particular, with ever-scaling technology and ever-increasing number of transistors, process variability greatly affects power and timing of nanometer scale CMOS circuits, leading to parametric yield loss due to timing violations [140]. The problem can be simply stated as follows. As the components of a chip get smaller and number of transistors keeps increasing, the performance of the design becomes more sensitive to minor changes in fabrication. Consequently, the execution/access latencies of identically-designed components can be different from each other. This variation problem is more severe in memory components since they are usually built using minimum sized transistors for density concerns. In current practice, there are two ways of dealing with this access timing problem in the context of a memory component such as SPM: either increasing the number of clock cycles required to access the SPM or increasing the clock cycle time of the CPU. Note that, while the first option affects only the SPM performance, the second option affects the chip performance in general. We want to emphasize that both these solutions are extremely

conservative and can be termed as the *worst case design option.* This is because they solve the timing failure problem by increasing the time to access the memory component (in terms of the number of cycles or cycle time). While they are easy to implement, it is not difficult to see that they can cost significant performance loss.

Our goal is to study alternate solutions to the access timing problem. Specifically, we would like to explore the possibility of working without the worst case design option. Our first goal in this context is to present an SPM management strategy that works with an SPM whose different words can have different latencies (i.e., an architecture that does not implement the worst case design option). Ultimate goal is to improve SPM performance as much as possible without causing any access timing failure.

- **Heterogeneous Chip Multiprocessors:** Increasing complexity of embedded applications and their large dataset sizes makes it imperative to consider novel embedded architectures that are efficient from both performance and power angles. Heterogeneous Chip Multiprocessors are one such promising example where asymmetric processor cores are placed into the same die. One may choose a large number of small low-power cores in a chip multiprocessor if the primary focus is power. Alternatively, a performance oriented approach may be chosen, where a smaller number of powerful cores with better single-thread performance are used. For a given application, an ideal architecture however will likely to contain both low-power and

powerful cores and memory components of different types (e.g., scratch pads, conventional caches, flash memories, etc). We believe this is where heterogeneous chip multiprocessors can play an important role.

Although heterogeneous chip multiprocessor architectures have been around for a while, in such systems each core performs a distinct task. More recently, researchers have proposed chip multiprocessor architectures where each core has the same ISA (instruction set architecture), enabling a more flexible application-to-core mapping. Clearly, it is not an easy task to map an application onto a heterogeneous chip multiprocessor considering both power and performance parameters. Our main goal on this topic will be to explore new ILP-based application-to-core mapping schemes that balance power and performance requirements of next generation embedded applications. We also plan to extend and apply our previous optimization strategies onto a heterogeneous chip multiprocessor environment. This will be especially important, in our opinion, for future 3D architectures.

Bibliography

[1] Xpress-mp. *http://www.dashoptimization.com/pdf/Mosel1.pdf*, 2002.

[2] Lzo algorithm, http://gnuwin32.sourceforge.net/packages/lzo.htm.

[3] The omega project, http://www.cs.umd.edu/projects/omega/.

[4] Mp98: A mobile processor, http://www.labs.nec.co.jp/MP98/top-e.htm.

[5] Majc-5200, http://www.sun.com/microelectronics/MAJC/5200wp.html.

[6] C. Ababei, H. Mogal, and K. Bazargan. Three-dimensional place and route for fpgas. In *Proceedings of Asia South-Pacific Design Automation Conference (ASP-DAC)*, 2005.

[7] 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.

[8] 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.

[9] A. J. Alexander, James P. Cohoon, Jared L. Colflesh, John Karro, and Gabriel Robins. Three-dimensional field-programmable gate arrays. In *Proceedings of the 8th IEEE International Application Specific Integrated Circuits Conference*, 1995.

[10] 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.

[11] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 318–326, 2003.

[12] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 259–267, 2004.

[13] 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.

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

[15] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[16] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the 10th International Symposium on Hardware/Software Codesign*, pages 73–78, 2002.

[17] Michael Beattie, Hui Zheng, Anirudh Devgan, and Byron Krauter. Spatially distributed 3d circuit models. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 153–158, 2005.

[18] 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.

[19] 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.

[20] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Des. Test*, 17(2):74–85, 2000.

[21] Caroline D. Benveniste, Peter A. Franaszek, and John T. Robinson. Cache-memory interfaces in compressed memory systems. *IEEE Trans. Comput.*, 50(11):1106–1116, 2001.

[22] Bryan Black, Donald Nelson, Clair Webb, and Nick Samra. 3d processing technology and its impact on ia32 microprocessors. In *22nd IEEE International Conference on Computer Design*, pages 316–318, 2004.

[23] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *In Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[24] 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.

[25] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[26] Yun Cao, Hiroyuki Tomiyama, Takanori Okuma, and Hiroto Yasuura. Data memory design considering effective bitwidth for low-energy embedded systems. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 201–206, 2002.

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

[28] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P.G. Kjeldsberg, T. V. Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, Boston, MA, USA, 2002.

[29] Francky Catthoor, Eddy de Greef, and Sven Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[30] 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.

[31] G. Chen, G. Chen, O. Ozturk, and M. Kandemir. Exploiting inter-processor data sharing for improving behavior of multi-processor socs. In *ISVLSI '05: Proceedings of the IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05)*, pages 90–95, 2005.

[32] G. Chen, M. Kandemir, and M. Karakoy. A constraint network based approach to memory layout optimization. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 1156–1161, 2005.

[33] G. Chen, F. Li, O. Ozturk, G. Chen, M. Kandemir, and I. Kolcu. Leakage-aware spm management. In *In Proc. IEEE Computer Society Annual Symposium on VLSI 2006 (ISVLSI 2006), Karlsruhe, Germany*, 2006.

[34] G. Chen, O. Ozturk, and M. Kandemir. An ilp-based approach to locality optimization. In *Languages and Compilers for High Performance Computing, 17th International Workshop*, pages 149–163, 2004.

[35] G. Chen, O. Ozturk, M. Kandemir, and M. Karakoy. Dynamic scratch-pad memory management for irregular array access patterns. In *Design Automation and Test in Europe (DATE'06), Munich, Germany*, 2006.

[36] G. Chen, O. Ozturk, M. Kandemir, and I. Kolcu. Integrating loop and data optimizations for locality within a constraint network based framework. In *Proc. of International Conference on Computer-Aided Design*, 2005.

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

[38] Lei Cheng, Liang Deng, and Martin D. F. Wong. Floorplanning for 3-d vlsi design. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 405–411, 2005.

[39] 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.

[40] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *In Proc. of Euro-Par*, 2004.

[41] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160, 2005.

[42] 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.

[43] Jason Cong and Yan Zhang. Thermal-driven multilevel routing for 3-d ics. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 121–126, 2005.

[44] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Acme: adaptive compilation made efficient. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 69–77, 2005.

[45] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 2–11, 1998.

[46] 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.

[47] Shamik Das, Anantha Chandrakasan, and Rafael Reif. Timing, energy, and thermal performance of three-dimensional integrated circuits. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 338–343, 2004.

[48] Shamik Das, Andy Fan, Kuan-Neng Chen, Chuan Seng Tan, Nisha Checka, and Rafael Reif. Technology, performance, and computer-aided design of three-dimensional integrated circuits. In *ISPD '04: Proceedings of the 2004 international symposium on Physical design*, pages 108–115, 2004.

[49] Minas Dasygenis, Erik Brockmeyer, Bart Durinck, Francky Catthoor, Dimitrios Soudris, and Antonios Thanailakis. A memory hierarchical layer assigning and prefetching technique to overcome the memory performance/energy bottleneck. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 946–947, 2005.

[50] 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.

[51] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[52] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.

[53] Yangdong (Steven) Deng and Wojciech Maly. 2.5d system integration: a design driven system implementation schema. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 450–455, 2004.

[54] 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 *Proceedings of the 41st Annual Conference on Design Automation*, pages 238–243, 2004.

[55] G. G. Fursin, M. F. P. O'Boyle, and P. M. W. Knijnenburg. Evaluating iterative compilation. In *In Proc. Workshop on Languages and Compilers for Parallel Computing*, 2002.

[56] G.Chen, G.Chen, O.Ozturk, and M.Kandemir. An adaptive locality-conscious process scheduler for embedded systems. In *In Proc. 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05), San Francisco, California*, March 2005.

[57] Ferid Gharsalli, Samy Meftali, Frederic Rousseau, and Ahmed A. Jerraya. Automatic generation of embedded memory wrapper for multiprocessor soc. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 596–601, 2002.

[58] Brent Goplen and Sachin Sapatnekar. Thermal via placement in 3d ics. In *ISPD '05: Proceedings of the 2005 international symposium on Physical design*, pages 167–174, 2005.

[59] 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.

[60] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 107–116, 2000.

[61] 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.

[62] Qubo Hu, Arnout Vandecappelle, Martin Palkovic, Per Gunnar Kjeldsberg, Erik Brockmeyer, and Francky Catthoor. Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 606–611, 2006.

[63] Hao Hua, Chris Mineo, Kory Schoenfliess, Ambarish Sule, Samson Melamed, Ravi Jenkal, and W. Rhett Davis. Exploring compromises among timing, power and temperature in three-dimensional integrated circuits. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 997–1002, 2006.

[64] W.-L. Hung, G. M. Link, Yuan Xie, N. Vijaykrishnan, and M. J. Irwin. Interconnect and thermal-aware floorplanning for 3d microprocessors. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 98–104, 2006.

[65] S. Im and K. Banerjee. Full chip thermal analysis of planar (2-d) and vertically integrated (3-d) high performance ics. In *International IEDM Technical Digest*, 2000.

[66] 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, 2006.

[67] ITRS. International technology roadmap for semiconductors, 2004.

[68] I. Kadayif, M. Kandemir, G. Chen, O. Ozturk, and U. Sezer. Optimizing array-intensive applications for on-chip multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):396–411, 2005.

[69] 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.

[70] 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, 2002.

[71] M. Kandemir, F. Li, G. Chen, G. Chen, and O. Ozturk. Studying storage-recomputation tradeoffs in memory-constrained embedded processing. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, 2005.

[72] M. Kandemir, O. Ozturk, and V. S. Degalahal. Enhancing locality in two-dimensional space through integrated computation and data mappings. In *In Proc. 20th International Conference on VLSI Design (VLSI'07), Bangalore, India*, January 2007.

[73] M. Kandemir, O. Ozturk, M. J. Irwin, and I. Kolcu. Using data compression to increase energy savings in multi-bank memories. In *Euro-Par*, pages 310–317, 2004.

[74] M. Kandemir, O. Ozturk, and M. Karakoy. Dynamic on-chip memory management for chip multiprocessors. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 14–23, 2004.

[75] 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.

[76] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th conference on Design automation*, pages 690–695, 2001.

[77] M. Kandemir, T. Yemliha, S. W. Son, and O. Ozturk. Memory bank aware dynamic loop scheduling. In *In Proc. of Design, Automation and Test in Europe (DATE'07), Nice, France*, 2007.

[78] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *Journal of Supercomputing*, 24(1):43–67, 2003.

[79] H. Koc, O. Ozturk, M. Kandemir, S. H. K. Narayanan, and E. Ercanli. Minimizing energy consumption of banked memories using data recomputation. In *In Proc. International Symposium on Low Power Electronics and Design (ISLPED'06), Tegernsee, Germany*, 2006.

[80] H. Koc, S. Tosun, O. Ozturk, and M. Kandemir. Task recomputation in memory constrained embedded multi-cpu systems. In *In Proc. IEEE Computer Society Annual Symposium on VLSI 2006 (ISVLSI 2006), Karlsruhe, Germany*, 2006.

[81] 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.

[82] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.

[83] 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.

[84] J. S. Lee, W. K. Hong, and S. D. Kim. Design and evaluation of a selective compressed memory system. In *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, page 184, 1999.

[85] S. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR-95-09-01, 1995.

[86] F. Li, G. Chen, M. Kandemir, O. Ozturk, M. Karakoy, R. Ramanarayanan, and B. Vaidyanathan. A process scheduler-based approach to noc power management. In *In Proc. 20th International Conference on VLSI Design (VLSI'07), Bangalore, India*, January 2007.

[87] Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and management of 3d chip multiprocessors using network-in-memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 130–141, 2006.

[88] Wei Li. *Compiling for numa parallel machines*. PhD thesis, Cornell University, Ithaca, NY, USA, 1993.

[89] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal Parallel Program*, 22(2):183–205, 1994.

[90] Y. Li and W. Wolf. Hardware/software co-synthesis with memory hierarchies. *IEEE Transactions on Computer-Aided Design of Integrated Circuit and Systems*, October 1999.

[91] Zhuoyuan Li, Xianlong Hong, Qiang Zhou, Shan Zeng, Jinian Bian, Hannah Yang, Vijay Pitchumani, and Chung-Kuan Cheng. Integrating dynamic thermal via planning with 3d floorplanning algorithm. In *ISPD '06: Proceedings of the 2006 international symposium on Physical design*, pages 178–185, 2006.

[92] Zuoyuan Li, Xianlong Hong, Qiang Zhou, Jinian Bian, Hannah H. Yang, and Vijay Pitchumani. Efficient thermal-oriented 3d floorplanning and thermal via planning for two-stacked-die integration. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2):325–345, 2006.

[93] 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.

[94] Sung Kyu Lim. Physical design for 3d system on package. *IEEE Des. Test*, 22(6):532–539, 2005.

[95] 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.

[96] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hllberg, Johan Hgberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[97] Peter Marwedel, Lars Wehmeyer, Manish Verma, Stefan Steinke, and Urs Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 4–11, 2004.

[98] Samy Meftali, Ferid Gharsalli, Frederic Rousseau, and Ahmed A. Jerraya. An optimal memory allocation for application-specific multiprocessor system-on-chip. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 19–24, 2001.

[99] 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.

[100] S. H. K. Narayan, O. Ozturk, M. Kandemir, and M. Karakoy. Workload clustering for increasing energy savings on embedded mpsocs. In *IEEE International SOC Conference*, 2005.

[101] S. H. K. Narayanan, O. Ozturk, and M. Kandemir. Compiler-directed power density reduction in noc-based multi-core designs. In *In Proc. the 7th International Symposium on Quality Electronic Design (ISQED'06), San Jose, CA*, 2006.

[102] 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.

[103] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.

[104] M. F. P. O'Boyle and P. M. W. Knijnenburg. Integrating loop and data transformations for global optimization. *Journal of Parallel and Distributed Computing*, 62(4):563–590, 2002.

[105] 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.

[106] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, 1996.

[107] O. Ozturk, G. Chen, and M. Kandemir. Compiler-guided data compression for reducing memory consumption of embedded applications. In *Proceedings of the Conference on Asia South Pacific Design Automation*, January 2006.

[108] O. Ozturk, G. Chen, and M. Kandemir. A constraint network based solution to code parallelization. In *In Proc. Design Automation Conference (DAC)[Nominated for Best Paper Award]*, 2006.

[109] O. Ozturk, G. Chen, and M. Kandemir. Multi-compilation: capturing interactions among concurrently-executing applications. In *In Proc. ACM International Conference on Computing Frontiers, Ischia, Italy*, May 2006.

[110] O. Ozturk, G. Chen, M. Kandemir, and M. Karakoy. Cache miss clustering for banked memory systems. In *In Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD'06), San Jose, CA*, 2006.

[111] O. Ozturk, G. Chen, M. Kandemir, and M. Karakoy. An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In *In Proc. IEEE Computer Society Annual Symposium on VLSI 2006 (ISVLSI 2006), Karlsruhe, Germany*, 2006.

[112] O. Ozturk and M. Kandemir. Energy management in software-controlled multi-level memory hierarchies. In *GLSVSLI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pages 270–275, 2005.

[113] O. Ozturk and M. Kandemir. Integer linear programming based energy optimization for banked drams. In *ACM Great Lakes Symposium on VLSI*, pages 92–95, 2005.

[114] O. Ozturk and M. Kandemir. Nonuniform banking for reducing memory energy consumption. In *DATE*, pages 814–819, 2005.

[115] O. Ozturk and M. Kandemir. Data replication in banked drams for reducing energy consumption. In *In Proc. the 7th International Symposium on Quality Electronic Design (ISQED'06), San Jose, CA*, 2006.

[116] O. Ozturk, M. Kandemir, G. Chen, and M. J. Irwin. Customized on-chip memories for embedded chip multiprocessors. In *Proceedings of the Conference on Asia South Pacific Design Automation*, pages 743 – 748 Vol. 2, January 2005.

[117] 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 [Nominated for Best Paper Award]*, pages 401–406, 2004.

[118] O. Ozturk, M. Kandemir, and M. J. Irwin. Bb-gc: basic-block level garbage collection. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, 2005.

[119] O. Ozturk, M. Kandemir, and M. J. Irwin. Increasing on-chip memory space utilization for embedded chip multiprocessors through data compression. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 87–92, 2005.

[120] O. Ozturk, M. Kandemir, and M. J. Irwin. Using data compression in an mpsoc architecture for improving performance. In *GLSVSLI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pages 353–356, 2005.

[121] O. Ozturk, M. Kandemir, M. J. Irwin, and I. Kolcu. Tuning data replication for improving behavior of mpsoc applications. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 170–173, 2004.

[122] O. Ozturk, M. Kandemir, M. J. Irwin, and S. Tosun. On-chip memory management for embedded mpsoc architectures based on data compression. In *IEEE International SOC Conference*, 2005.

[123] O. Ozturk, M. Kandemir, M. J. Irwin, and S. Tosun. Multi-level on-chip memory hierarchy design for embedded chip multiprocessors. In *In Proc. The Twelfth International Conference on Parallel and Distributed Systems (ICPADS'06) [Best Paper Award]*, July 2006.

[124] O. Ozturk, M. Kandemir, and M. Karakoy. Selective code/data migration for reducing communication energy in embedded mpsoc architectures. In *In Proc. GLSVLSI, Philadelphia, PA*, 2006.

[125] O. Ozturk, M. Kandemir, and M. Karakoy. Compiler-directed variable latency aware spm management to cope with timing problems. In *In Proc. IEEE/ACM International Symposium on Code Generation and Optimization (CGO'07), San Jose, CA*, 2007.

[126] O. Ozturk, M. Kandemir, and I. Kolcu. Shared scratch-pad memory space management. In *In Proc. the 7th International Symposium on Quality Electronic Design (ISQED'06), San Jose, CA*, 2006.

[127] O. Ozturk, M. Kandemir, and S. W. Son. Ilp-based management of multi-level memory hierarchies. In *In Proc. 4th Workshop on Optimizations for DSP and Embedded Systems (ODES'06), Manhattan, New York, NY*, March 2006.

[128] O. Ozturk, M. Kandemir, S. W. Son, and I. Kolcu. Managing spm space based on inter-application data sharing. In *In Proc. 4th Workshop on Optimizations for DSP and Embedded Systems (ODES'06), Manhattan, New York, NY, March*, 2006.

[129] O. Ozturk, M. Kandemir, and S. Tosun. An ilp based approach to address code generation for digital signal processors. In *In Proc. GLSVLSI, Philadelphia, PA*, May 2006.

[130] O. Ozturk, H. Saputra, M. Kandemir, and I. Kolcu. Access pattern-based code compression for memory-constrained embedded systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 882–887, 2005.

[131] Preeti Ranjan Panda and Lakshmikantam Chitturi. An energy-conscious algorithm for memory port allocation. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 572–576, 2002.

[132] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Architectural exploration and optimization of local memory in embedded systems. In *ISSS '97: Proceedings of the 10th international symposium on System synthesis*, pages 90–97, 1997.

[133] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the European Conference on Design and Test*, 1997.

[134] Kiran Puttaswamy and Gabriel H. Loh. Implementing caches in a 3d technology for high performance processors. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 525–532, 2005.

[135] Kiran Puttaswamy and Gabriel H. Loh. Thermal analysis of a 3d die-stacked high-performance microprocessor. In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 19–24, 2006.

[136] Anand Ramachandran and Margarida F. Jacome. Xtream-fit: an energy-delay efficient data memory subsystem for embedded media processing. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 137–142, 2003.

[137] G. Reinman and N. P. Jouppi. Cacti 2.0: An integrated cache timing and power model. Technical report, Compaq, February 2000.

[138] 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.

[139] 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.

[140] Kaushik Roy. Process variations in nanoscale technologies: Failure analysis, self-calibration, process-compensation, and fault tolerance. GSRC Quarterly Workshop, June 2005.

[141] Wen-Tsong Shiue and Chaitali Chakrabarti. Memory exploration for low power, embedded systems. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 140–145, 1999.

[142] K. Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Hotspot thermal modeling simulator.

[143] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2002.

[144] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004.

[145] S. Tosun, N. Mansouri, M. Kandemir, and O. Ozturk. Constraint-based code mapping for heterogeneous chip multiprocessors. In *IEEE International SOC Conference*, 2005.

[146] S. Tosun, N. Mansouri, O. Ozturk, and M. Kandemir. An ilp formulation for task scheduling on heterogeneous chip multiprocessors. In *In Proc. he 21st International Symposium on Computer and Information Sciences (ISCIS'06), Istanbul, Turkey*, November 2006.

[147] C. Tsai and S. Kang. Cell-level placement for improving substrate thermal distribution. *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems*, 19:253–266, 2000.

[148] Yuh-Fang Tsai, Yuan Xie, N. Vijaykrishnan, and Mary Jane Irwin. Three-dimensional cache design exploration using 3dcacti. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 519–524, 2005.

[149] Edward Tsang. A glimpse of constraint satisfaction. *Artificial Intelligence Review*, 13(3):215–227, 1999.

[150] B. Tunstall. *Synthesis of Noiseless Compression Codes*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1967.

[151] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, 2003.

[152] Balaji Vaidyanathan, Wei-Lun Hung, Feng Wang, Yuan Xie, Vijaykrishnan Narayanan, and Mary Jane Irwin. Architecting microprocessor components in 3d design space. In *IEEE VLSI Design Conference*, pages 103–108, 2007.

[153] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2004.

[154] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratch-pad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109, 2004.

[155] L. Wang, W. Tembe, and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. In *Proceedings of the 9th International Conference on Compiler Construction*, pages 141–156, 2000.

[156] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[157] 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.

[158] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded risc architecture. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 81–91, 1992.

[159] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[160] Yuan Xie, Gabriel H. Loh, Bryan Black, and Kerry Bernstein. Design space exploration for 3d architectures. *J. Emerg. Technol. Comput. Syst.*, 2(2):65–103, 2006.

[161] 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.

[162] 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.

[163] 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, 2003.

[164] L. Xue, M. Kandemir, G. Chen, F. Li, O. Ozturk, R. Ramanarayanan, and B. Vaidyanathan. Locality-aware distributed loop scheduling for chip multiprocessors. In *In Proc. 20th International Conference on VLSI Design (VLSI'07), Bangalore, India*, January 2007.

[165] L. Xue, O. Ozturk, F. Li, , and I. Kolcu. Dynamic partitioning of processing and memory resources in embedded mpsoc architectures. In *Design Automation and Test in Europe (DATE'06), Munich, Germany*, 2006.

[166] T. Yemliha, G. Chen, O. Ozturk, M. Kandemir, and V. S. Degalahal. Compiler-directed code restructuring for operating with compressed arrays. In *In Proc. 20th International Conference on VLSI Design (VLSI'07), Bangalore, India*, January 2007.

[167] Hao Yu, Yiyu Shi, Lei He, and Tanay Karnik. Thermal via allocation for 3d ics considering temporally and spatially variant thermal power. In *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*, pages 156–161, 2006.

[168] 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.

[169] Tianpei Zhang, Yong Zhan, and Sachin S. Sapatnekar. Temperature-aware routing in 3d ics. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 309–314, 2006.

Vita

Ozcan Ozturk

Education

The Pennsylvania State University University Park, Pennsylvania 2002–Present
Ph.D. in Computer Engineering, expected in May 2007
Area of Specialization: Compiler Optimization

University of Florida Gainesville, Florida 2000–2002
MS. in Computer Engineering

Bogazici University Istanbul, Turkey 1996–2000
BS. in Computer Engineering

Awards and Honors

Best Paper Award from the Twelfth ICPADS Conference 2006
Nominated for Best Paper Award in DAC 2006
Selected by SIGDA to present at the PhD Forum of DAC 2005
Nominated for Best Paper Award in DAC 2004
Scholarship from Swiss Federal Institute of Technology of Lausanne 2003

Research Experience

Doctoral Research The Pennsylvania State University 2002–Present
Thesis Advisor: Prof. Mahmut Kandemir
Compiler Directed Memory Hierarchy Design and Management in Chip Multiprocessors.

Graduate Research Swiss Federal Institute of Technology of Lausanne (EPFL), 2003
Research Advisor: Prof. Paolo Ienne
Optimization of a MachSUIF compiler backend for the ARM architecture.

Teaching Experience

Teaching Assistant The Pennsylvania State University 2002–2003
Instructor for Business Programming Applications and C++ Programming for Engineers courses.

Teaching Assistant University of Florida 2000–2002
Tutored and assisted Database Management Systems and Applications of Discrete Structures courses.