Energy optimization by scratchpad memory banking for embedded systems

Noha Abuaesh
The American University in Cairo AUC

Follow this and additional works at: https://fount.aucegypt.edu/retro_etds

Part of the Computer Engineering Commons

Recommended Citation

APA Citation

MLA Citation

This Thesis is brought to you for free and open access by AUC Knowledge Fountain. It has been accepted for inclusion in Archived Theses and Dissertations by an authorized administrator of AUC Knowledge Fountain. For more information, please contact fountadmin@aucegypt.edu.
ENERGY OPTIMIZATION BY SCRATCHPAD MEMORY BANKING FOR EMBEDDED SYSTEMS

A Thesis Submitted to
The Department of Computer Science and Engineering

in partial fulfillment of the requirements for the degree of Master of Science

by Noha Abuaesh

under the supervision of Prof. Florin Balasa
December/2013
DEDICATION

To my beloved parents... May they be granted every happiness in this world and in the hereafter...
ACKNOWLEDGEMENT

“Praise to Allah, who has guided us to this; and we would never have been guided if Allah had not guided us.” –Quran 7:43

My endless thanks—if it will ever suffice-- to my wonderful parents; my devoted mom and compassionate dad, for their encouragement and sacrifice.

The thesis at your hands would not have been in this form if it wasn’t for the dedication and supervision of Prof. Florin Balasa; to whom I owe my sincerest gratitude.

I would like to appreciate the valuable trust put in me by the Department of Computer Science and Engineering by nominating me for a tuition-waiver fellowship; without which I wouldn’t have joined the graduate program in the first place.

My sincere thanks to everyone who helped, encouraged or prayed for me. I am forever indebted.

All praise to Allah, Lord of the worlds. To Him are my initial and final thanks.
ABSTRACT

In real-time data-intensive multimedia processing applications, data transfer and storage significantly influence, if not dominate, all the major cost parameters of the design space – namely power consumption, performance, and chip area. Hierarchical memory organizations are used in embedded systems to reduce energy consumption and improve performance by exploiting the non-uniformity of memory accesses, by assigning the frequently-accessed data to low levels of the hierarchy. Moreover, within a given level, energy can be further reduced and performance further enhanced by memory partitioning – whose principle is to divide the address space in several smaller blocks and to map these blocks to physical memory banks. Scratch-pad memories (SPMs) offer a good compromise – as on-chip storage in embedded systems – when taking into account performance, energy consumption, and die area. This thesis addresses the problem of optimizing the partitioning of SPMs.

Different from previous techniques, this approach has as main input the application code, rather than a memory access trace obtained by simulation.

The approach builds upon a framework that employs a formal model operating with integral polyhedra, using techniques specific to the data-dependence analysis employed in modern compilers. Thus, and unlike previous techniques, the problems of data assignment to the memory layers and banking the on-chip memory are addressed in a consistent way, based on the same formal model.

Another major difference is that the cost function takes into account all the three major design objectives, letting the designers decide on their relative importance for a specific project. The main design target is the reduction of the static and dynamic energy consumption in the memory subsystem, but the same formal model and algorithmic flow can be also applied to reduce the overall time of access to memories.

The proposed approach proved to be computationally fast and very efficient when tested for several data-intensive applications, whose behavioral specifications contain multidimensional arrays as main data structures.
TABLE OF CONTENTS

DEDICATION.................................................................................................................. II

ABSTRACT..................................................................................................................... IV

TABLE OF CONTENTS................................................................. V

TABLE OF FIGURES.............................................................................. VIII

LIST OF TABLES................................................................................ XI

LIST OF ALGORITHMS.......................................................................... XII

1 INTRODUCTION......................................................................................... 1

1.1 Problem Definition and Scope Limitations....................................................... 7

1.2 Literature Survey ............................................................................................ 9

1.3 Significance of the Study .............................................................................. 11

1.4 Research Motivation ....................................................................................... 14

1.5 Objectives of the Study .................................................................................. 15

1.6 Document Organization .................................................................................. 16

2 COLLECTING SIMULATION DATA....................................................... 17

2.1 About CACTI Simulator ................................................................................ 17

2.1.1 Definition .................................................................................................... 17

2.1.2 Quick History and Origin ......................................................................... 17

2.1.3 CACTI Forms ............................................................................................ 18

2.2 The Used Version is CACTI 6.5 (Offline Form) ............................................. 18

2.3 Using CACTI's Offline Version ....................................................................... 19

2.4 Using CACTI to Generate Batch Simulation Data ......................................... 19
3.1 Inputs .................................................................................................................. 24

3.2 The Recursive Algorithm by Benini et al. .......................................................... 25

3.3 Cost Calculation .................................................................................................. 27

3.4 Results and Discussion ....................................................................................... 28

3.5 Complexity and Analysis .................................................................................... 32

3.6 Limitations and Shortcomings ............................................................................ 34

4.1 About Dynamic Programming ............................................................................. 35

4.2 A Dynamic Programming Algorithm by Angiolini et al. ................................. 39

4.2.1 Overview of the Design Flow ......................................................................... 39

4.2.2 Cost Calculation ............................................................................................... 41

4.2.3 The Algorithm ................................................................................................. 43

4.2.4 Complexity and Analysis ................................................................................ 45

4.2.5 Limitations and Shortcomings ........................................................................ 47

5.1 Inputs .................................................................................................................... 49

5.2 The Algorithm ..................................................................................................... 50

5.3 Cost Calculation .................................................................................................. 52

5.4 Results and Discussion ....................................................................................... 53

5.5 Complexity and Analysis .................................................................................... 56
5.5.1  Space Complexity: ................................................................. 56
5.5.2  Time Complexity: ................................................................. 57

5.6  Limitations and Shortcomings ...................................................... 59

6  A PROPOSED BACKTRACKING APPROACH ................................. 61

6.1  Significance of Code Analysis ..................................................... 61
   6.1.1  Case 2-way partitioning ...................................................... 63
   6.1.2  Case 3-way partitioning ...................................................... 64

6.2  A Memory Management Framework ........................................... 66
   6.2.1  Input ............................................................................... 66
   6.2.2  Output ............................................................................. 67
   6.2.3  The Design Flow .............................................................. 67

6.3  A Backtracking-Based Partitioning Algorithm ............................... 74
   6.3.1  Inputs .............................................................................. 75
   6.3.2  The Algorithm ................................................................... 76
   6.3.3  Cost Calculation ............................................................... 79
   6.3.4  Results and Discussion ...................................................... 81
   6.3.5  Complexity and Analysis .................................................... 87

7  CONCLUSIONS ............................................................................. 89

7.1  Summary .................................................................................. 89

7.2  Obstacles and Constraints .......................................................... 91

7.3  Future Directions ....................................................................... 91

REFERENCES .................................................................................. 94

APPENDIX A – SIMULATION DATA ................................................. 101

APPENDIX B – TESTING RESULTS .................................................. 102
# TABLE OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A survey of power reduction techniques in microprocessor systems</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Memory hierarchy example</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Sample division of the memory address Space</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>(a) Architecture with monolithic on-chip SPM. (b) Architecture with on-chip SPM partitioned in 3 banks</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>A Venn diagram of the problem research area and its connection to related fields</td>
<td>8</td>
</tr>
<tr>
<td>1.6</td>
<td>Block diagram of a typical embedded processor configuration not making use of cache memory</td>
<td>9</td>
</tr>
<tr>
<td>1.7</td>
<td>Illustrative example of a typical affine specification</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>A diagram showing the inputs and outputs of CACTI</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Snapshot of a sample output file (cache32nm512:1024:16.txt) generated by the batch generator</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Workflow of the batch generator</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>An optimal 2-way partitioning is found by $\text{Min}<em>k {E</em>{\text{Bank1}} + E_{\text{Bank2}} } + \Delta E_{12}$. Optimal M-way partitioning is found using recursion</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Total energy consumption for different number of partitions shown for two different values of word width. While increasing the number of partitions reduces the total energy, the figure suggests insignificant progress in energy consumption by decreasing the word width</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Total execution times for different number of partitions are shown for two different values of word width</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>The recursion tree for computing scratchpad partitioning solution for an address range 1 to 4. The darkened subtrees are overlapping subproblems. Figure from [49]</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Schematic Design Flow presented by Angiolini et al. in [3]</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Principal structure of the dynamic programming algorithm presented by Angiolini et al.</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Principal structure of the P and S matrices presented in the dynamic programming algorithm</td>
<td>52</td>
</tr>
</tbody>
</table>
Figure 5.2 Energy costs obtained for different values of $\Phi$ and $G$. Even though energy cost is expected to increase with the partitions sizes increase; since accessing larger banks of memory consumes more energy than accessing banks of smaller sizes, the figure suggests a negligible difference between the energy results.

Figure 5.3 Performance costs obtained for different values of $\Phi$ and $G$. Performance exhibits a similar behavior to energy. Analogous to energy consumption, accessing larger banks of memory takes more time than accessing banks of smaller sizes. We may roughly estimate that optimizing for energy produces somewhat efficient solutions. This can also be noticed especially when compared with the results obtained from performance optimization tests; see page 100.

Figure 5.4 Memory requirements for an 8 KB trace ($N=8192$) tested with different granularities (or word widths). The memory requirements are the same for different minimum partition sizes.

Figure 5.5 CPU running time for an 8 KB trace ($N=8192$) tested with different word width. The worst time was 10 minutes taken by word width of 4 bytes.

Figure 6.1 Illustrative example of a typical affine specification with 4 loop nests. Accesses to signal A are circled.

Figure 6.2 The partitioning of signal A’s array space. The number of memory accesses is indicated for each partition – the darker ones being more heavily accessed.

Figure 6.3 Flowchart of a memory management system relying on electronic design automation (EDA) methodology presented by Balasa et al. in [13] for the high-level design of hierarchical memory architectures for data-intensive signal processing applications.

Figure 6.4 Part (a) shows the code excerpt containing accesses to signal X. Part (b) shows the mapping from the iterator space into the index space for the array reference bounded by the conditions in the if statement.

Figure 6.5 An example of an exact access map of a 2-D signal of size 350x350 elements is shown in part (a). Part (b) shows the computed 3-D map of memory accesses to the same signal, based on the decomposition of its index space into disjoint bounded lattices. The horizontal plane (XY) represents the array dimensions while the vertical Z axis is the access density.
Figure 6.6 The distribution of memory accesses to the scratchpad address space assuming the 2-D array A is assigned to the on-chip scratchpad column-by-column.

Figure 7.1 The distribution of memory accesses to the scratchpad address space assuming the 2-D array A is mapped into the on-chip scratchpad column-by-column. Dotted lines showing possible partitioning positions of the on-chip address space, if intermittent address ranges are to be allowed on the same memory bank.
LIST OF TABLES

Table 3.1 Testing results of the recursive algorithm with an 8KB benchmark. Entries left blank could not be generated due to intractable execution times. 30

Table 6.1 Testing results for running the backtracking algorithm presented in Algorithm 6.1 on different benchmarks [50]. Columns 3 to 6 show testing results obtained while running the algorithm with energy optimization as the only target, i.e. w1=1. Columns 3 and 5 display the results of running Benini’s algorithm. The results in columns 7, 8 and 9 show the energy, access time and area savings achieved by running the algorithm on M=8 with w1=w2=0.4 and w3=0.2 assuming a 32 nm technology. 83

Table 6.2 Brief information on the selected benchmarks used for testing the proposed backtracking algorithm. The results of the tests are shown in Table 6.2. 85

Table 6.3 Table 7.1 Summary of the discussed algorithms. 90

Table 7.2 Obstacles and constraints faced during the research. 91
LIST OF ALGORITHMS
Algorithm 3.1 Recursive Algorithm by Benini et al. [2] ......................... 26
Algorithm 4.1 Pseudo code for the Angiolini et al. algorithm.................. 43
Algorithm 5.1 Pseudo code for the novel dynamic programming algorithm. 51
Algorithm 6.1 Pseudo code of the proposed banking algorithm [41]......... 77
Algorithm 6.2 The recursive bank partitioning function used in the proposed algorithm[41]. ................................................................. 78
1 INTRODUCTION

An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints [51]. It is embedded as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a personal computer (PC), is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today. Embedded systems are controlled by one or more main processing cores that are typically either microcontrollers or digital signal processors (DSP). The key characteristic, however, is being dedicated to handle a particular task, which may require very powerful processors.

Embedded systems span all aspects of modern life and there are many examples of their use. Telecommunications systems employ numerous embedded systems from telephone switches for the network to mobile phones at the end-user. Computer networking uses dedicated routers and network bridges to route data. Consumer electronics include personal digital assistants (PDAs), mp3 players, mobile phones, video-game consoles, digital cameras, DVD players, GPS receivers, and printers. Many household appliances, such as microwave ovens, washing machines and dishwashers, are including embedded systems to provide flexibility, efficiency and features.

The design trade-off between performance and energy efficiency has become a major point of concern in embedded systems [52], [12]. This is mainly due to the fact that such systems appear in electronic products available on the market which are portable and battery-operated (such as cellular phones, laptop computers). This implies that their functionality be fulfilled with energy delivered by a battery of minimum weight and size [31].

In the hardware platform of an embedded system, three types of operations are responsible for energy consumption: (1) data processing, (2) data transfers, and (3) data storage [52], [12], [31]. Since the software component of an embedded system does not have a physical realization, suitable models for estimating the software impact on the hardware energy consumption have been proposed (e.g., [53]). The choice of the software implementation may
affect the energy consumption of all the three operations performed by the hardware platform. For instance, software compilation affects the instructions used by the computing elements (processors, DSPs), each one with a specific energy cost, consequently, having an impact on the energy consumption of data processing.

This thesis will focus on a class of techniques targeting mainly the reduction of power consumption in the hardware platform for data storage operations.

Many signal processing systems, particularly in the multimedia and telecommunication domains, are synthesized to execute data-intensive applications. Data-intensive algorithms for multimedia applications—possibly real-time as well— are typically specified in a high-level programming language where the code is organized in sequences of loop nests having as boundaries linear functions of the outer loop iterators, conditional instructions, and multidimensional signals whose array references have, possibly complex, linear indices. This class of specifications is often referred to as *affine* due to the fact that they contain array references whose index values can be typically represented by vector functions. The basic parameters of the design space in such applications-- namely power consumption, performance, and chip area -- are heavily influenced, if not dominated, by the data transfer and storage aspects[12].

The reduction of power consumption can be achieved by many ways; see Figure 1.1. However, the memory subsystem, in particular, is, typically, a major contributor to the overall energy budget of the entire system [31](and often a bottleneck for performance[32]). The *dynamic energy* consumption is caused by memory accesses, whereas the *static energy* consumption is due to leakage currents. One of the optimization techniques for memory subsystems is adopting hierarchical memory organizations. This organization reduces energy consumption by exploiting the non-uniformities of memory accesses by assigning the frequently-accessed data to low hierarchy levels, with the problem being how to optimally assign the data to the memory layers.
In the earlier days of digital system design, memory was expensive; so, researchers focused on memory size optimization. Nowadays, the cost per memory bit is very low due to the progress of the semiconductor technology and the consequent increase of the level of integration. Gradually, the memory size optimization decreased in importance, while performance and power consumption became the key challenges.

Memory latency; that is, the time it takes to access a particular location in storage, and energy consumption per access increase with the memory size. Hence, memory may become a bottleneck -- both in terms of energy and performance -- for applications with large storage requirements [52],[12], especially when the flat memory architecture -- that is, when data is stored in a single, off-chip memory -- is adopted. Therefore, reducing the
memory requirements of the target applications continues to be used as a first-phase design flow strategy for reducing the storage power budget and increasing performance. During this phase, the designer attempts to improve the temporal locality of data (that is, the results of a computation should be used as soon as possible by next computations in order to reduce the need for temporary storage) by performing code (especially, loop) transformations on the behavioral specifications [54],[55],[56],[57].

Data compression is another technique for reducing the storage requirements, which targets finding efficient representations of data, for example [58]. However, data memory size have steadily increased over time due to the fact that system applications grew more complex. Further efforts to tackle the memory bottleneck focused on (1) energy-efficient technologies and circuit design [59], [60] and (2) hierarchical memory architectures.

In general, lower levels in the hierarchy are made of small memories, close and tightly coupled to computation units; higher hierarchy levels are made of increasingly large memories, far from the computation units. The terms "close" and "far" imply here the effort needed to fetch or store a given amount of data from/to the memory. This effort can be expressed in units of time or units of energy, depending on the cost function.

![Figure 1.2 Memory hierarchy example](image)

Hierarchical memory organizations -- like the one in Figure 1.2 above--reduce energy consumption by exploiting the non-uniformities of memory accesses: most applications access a relatively small amount of data with high frequency, while the rest of the data are accessed only a few times[63]s. In a
hierarchical memory organization, the reduction of power consumption can be achieved by assigning the frequently-accessed data in low hierarchy levels.

As on-chip storage, the scratch-pad memories (SPMs) -- compiler-controlled static random-access memories (SRAMs), more energy-efficient than the hardware-managed caches -- are widely used in embedded systems, where caches incur a significant penalty in aspects like area cost, energy consumption, and hit latency. A detailed study[61] comparing the trade-offs of caches as compared to SPMs found in their experiments that the latter exhibit 34% smaller area and 40% lower power consumption than a cache of the same capacity. Even more surprisingly, the runtime measured in cycles was 18% better with a scratchpad using a simple static knapsack-based allocation algorithm. As a general conclusion, the authors of the study found absolutely no advantage in using caches [27], even in high-end embedded systems in which performance is important1.

Scratchpads are used to statically store a portion of the off-chip memory (or a memory farther in the hierarchy). This is in contrast with caches, that dynamically map a set of non-contiguous addresses from a slower, larger memory.

Different from caches, the scratchpad occupies a distinct part of the virtual address space, with the rest of the address space occupied by the main memory (see Figure 1.3). The consequence is that there is no need to check for the availability of the data in the SPM. Hence, the scratchpad does not possess a comparator and the miss/hit acknowledging circuitry[61]. This contributes to a significant energy--as well as area-reduction. Another consequence is that in cache memory systems, the mapping of data to the cache is done during the code execution, whereas in scratchpad-based systems this can be done at compilation time, using a suitable algorithm. Therefore, scratchpads are used as a L1 on-chip memory type for a large number of high-end embedded systems[32]; since most of the embedded systems perform a fixed and dedicated set of functions. For example, the microprocessor which controls the fuel injection system in a car will perform the same functions for its entire life-

---

1 Caches have been a big success for desktops though, where the usual approach to adding SRAM is to configure it as a cache.
time. A good example of an embedded processor system that uses scratchpad memory is the ARM7TDMI evaluation board (AT91EB01) [76]. The ARM7TDMI processor is a simple 32 bit RISC processor which implements the ARM Instruction Set Architecture (ISA) version 4T [75]. It is the most widely used processor core in contemporary low power embedded devices [32].

Moreover, within a given memory hierarchy level, power can be reduced by memory partitioning -- whose principle is to divide the address space in several smaller blocks, and to map these blocks to physical memory banks that can be independently enabled and disabled.

Memory partitioning—or banking-- is also a performance-oriented optimization strategy, because of the reduced latency due to accessing smaller memory blocks. What actually makes this class of techniques low-power is the opportunity of selectively shutting down the memory blocks that are not accessed; an operation that has little effect on performance [31].

Arbitrarily fine partitioning is prevented since an excessively large number of small banks is area inefficient, imposing a severe wiring overhead as well as complex circuitry for addressing; which tends to increase the die area and, hence, increase communication power and decrease performance. Figure 1.4 shows the extra overheads imposed by a 3-bank scratchpad memory architecture as compared with a single-bank scratchpad architecture.
This thesis addresses the problem of partitioning the SPM address space, focusing mainly on the reduction of energy consumption of the SPM. Different from previous techniques, we introduce a cost function that takes into account the three major design objectives -- energy consumption, performance, and die area -- letting the designers decide on their relative importance for a specific project. Different from previous techniques that have as main input the execution trace of the application [2], [3], [4], [18], [31], [67], we will also make use of a memory management system that starts from the behavioral specification, deciding in a preliminary phase the data assignment to the memory layers and the mapping of signals to the physical memories. When tested for several data-intensive applications, whose behavioral specifications contain multidimensional arrays as main data structures, the proposed approach proved to be computationally fast even for SPMs of a larger size, being able to explore banking solutions with a larger number of partitions.

1.1 Problem Definition and Scope Limitations

This research addresses the problem of optimizing the energy, performance and chip area in hierarchical storage organizations of embedded systems by scratchpad memory partitioning.

The thesis does not investigate the idleness of data stored in the banks, in order to be able to put the banks into a 'sleep' mode when they are not accessed for a longer time. This will be addressed as a future research.
The memory management goals of this research target especially applications in multidimensional signal processing (multimedia, telecommunication) that are *data-intensive* or even *data-dominated*—in the sense that their cost related aspects, namely power consumption and footprint are heavily influenced, if not dominated, by the data access and storage aspects.

For instance, in a System-on-a-Chip implementing an MPEG4 decoder multiplexing and de-multiplexing between multiple video and speech/audio streams [62], the 16 Mb embedded DRAM occupies about 40% of the chip, and the various caches and SRAM buffers occupy roughly 20% of the area. Thus, more than 60% of the active chip area is dedicated to embedded memories. The chip consumes 240 mW at 60 MHz, and the memory access power (both to embedded SRAM and DRAM memories) is the dominant contributor to the overall chip power budget.²

Figure 1.5 shows a Venn diagram of the related research areas and the location of our research on the map.

![Figure 1.5 A Venn diagram of the problem research area and its connection to related fields.](image)

The more common memory architecture of an on-chip scratchpad and an off-chip DRAM [23], as shown in Figure 1.6, is assumed. It is also assumed that the off-chip and on-chip storage requirements are computed for the multidimensional signal [11], [16], [29] and that the signal assignment to each layer has already been decided [17], [28], and that the mapping functions [19],

² The MPEG4 standard is one of the video coding methods used in wireless telephony, especially in the so-called *third generation* mobile telephony -- supporting transmission of data streams, as well as speech and data.
for all the signals in the behavioral specification are already known [28], [37]. The partitioning technique to be devised will be integrated in a larger CAD memory management system; targeting data-intensive applications tackling the major attributes of memory organizations for embedded systems, see section 6.2.3.

![Figure 1.6 Block diagram of a typical embedded processor configuration not making use of cache memory[23]. Components inside the dotted box are on-chip elements. This design is assumed throughout this research.](image)

The evaluation of the memory metrics—-that is, energy consumption, access time and chip area-- in the scratchpad are based on results achieved from CACTI 6.5[36]; an analytical tool that takes a set of cache, scratchpad, or DRAM parameters as input and calculates times of memory accesses for read/write operations, energy spent per access, static power, and chip area. The interface used on top of CACTI as well as the tool itself will be discussed in more detail in a separate chapter.

### 1.2 Literature Survey

As on-chip storage, the scratchpad memories (SPMs) -- compiler-controlled SRAMs that are more energy-efficient than the hardware-managed caches -- are widely used in embedded systems, where caches incur a significant penalty in aspects like area cost, energy consumption, and hit latency[27].

In spite of their superior power efficiency, SPMs are often a large contributor to the overall energy budget of the entire system: this is caused by the tendency of the designer to store on-chip as much data as possible in order to improve performance. Therefore, a lot of effort is still involved to devise solutions aiming to optimize the SPM partitioning and memory hierarchies --
in general. This issue has been widely acknowledged and several teams of researchers came up with a wide spectrum of solutions, at the beginning -- for the reduction of dynamic energy consumption, which expands due to memory accesses. Many software-based techniques have been initially developed for caches, and a comprehensive review can be found in [68]. These research works have been revisited and adapted to SPMs as well.

Partitioning of on-chip memories have been analyzed by several research teams, being typically used as an additional dimension of the memory design space. For instance, Ko et al. [15] and Shiue and Chakrabarty [20][21] studied power-efficient partitioned cache organizations, identifying cache sub-banking as an effective approach to reduce cache power consumption.

Logical partitioning consists of splitting the on-chip cache into a spatial and a temporal cache. Grun et al. statically mapped the data with high spatial and temporal correlation to either cache, using access profiles for embedded applications [33].

Coumeri and Thomas studied embedded static RAMs and described a partitioned SRAM model; called segmented configuration, analyzing partition matching against the access patterns of the application [22].

Kandemir et al. proposed a compiler-controlled dynamic scratchpad management using loop and data transformations [34], then a transformation called array interleaving [35]. They also exploited scratchpad bank locality for maximizing the idleness, thus ensuring maximal amortization of the energy spent on memory re-activation [5],[10]. Golubeva et al. proposed a leakage-aware approach based on traces of memory accesses [4].

Benini et al. proposed a recursive partitioning of the scratchpad address space, provided with a backtracking mechanism, that achieved a complete exploration of the banking solutions [2],[18]. A further optimization was described in [3]: the cost function was shown to exhibit properties that allow applying a dynamic programming paradigm.

Memory partitioning techniques that have been proposed differ in the hierarchy level targeted for partitioning. Farrahi et al. have studied the problem in the context of board-level memory optimization, where memory
blocks are large, off-chip dynamic random-access memories (DRAMs) that can be powered down when they are not storing live program variables [30]. This technique attempts to cluster data into memories so that memory chips are transitioned in and out of the shut-down mode as scarcely as possible.

After the scaling of the technology below 100 nm, the static energy due to leakage currents has become increasingly important. While leakage is a problem for any transistor, it is even more critical for memories: their high density of integration translates into a higher power density that increases temperature, which in turn increases leakage currents significantly.

Kandemir et al. exploited SPM bank locality for maximizing the idleness, thus ensuring maximal amortization of the energy spent on memory re-activation [34]. Hardware schemes putting a memory block into a dormant (sleep) state with negligible energy spending have been proposed [65], [66]. These schemes normally imply a time and an energy overhead: transitioning a memory block into and, especially, out of the dormant state consumes both energy and time.

A team of researchers proposed a leakage-aware approach based on traces of memory accesses: their method takes into account that putting a memory block into the dormant state should be done only if the cost in extra energy and decrease of performance can be amortized [4], [67].

1.3 Significance of the Study

The data-intensive algorithms for (real-time) multimedia applications are typically specified in a high-level programming language, where the code is organized in sequences of loop nests having as boundaries linear functions of the outer loop iterators, conditional instructions, where the conditions may be both data-dependent or data-independent and multidimensional signals whose array references have (possibly complex) linear indices. This class of specifications is often referred to as affine due to the fact that they contain array references whose indexes are affine functions of the loop iterators[52].

Sometimes, in image, speech, and numerical processing, there may be also indices containing modulo operators: using Hermite Normal Forms [64], such
specifications can be transformed into the affine class applying only integer arithmetic. The illustrative code with four nested loops shown in Figure 1.7 is an example of such a specification. The example shows a typical affine behavioral specification with 4 nested loops intensively accessing signal A array space. We will be using this illustrative example as the main benchmark for testing and comparing different methods in this research as it represents a typical affine behavioral specification; the class of applications targeted by our work. We also make the assumptions that the entire array A is stored in an on-chip scratchpad, with each element occupying only 1 byte, the mapping of the array into the scratchpad is column-by-column, and the on-chip memory has one read/write port.

```
C[0] = 0; // A[16][512]: input
for (int n=0; n < 3; ++n) {
    for (int j=96; j < 416; ++j) {
        B[n][j][0] = 0;
        for (int i=n; i < n+14; ++i) {
            for (int k=j-96; k <= j+96; ++k) {
            }
        }
    }
    C[320*n+j-95] = B[n][j][2702] + C[320*n+j-96];
}
out = C[960]; // out: output
```

Figure 1.7 Illustrative example of a typical affine specification.

Benini et al. proposed a recursive partitioning of the scratchpad address space that optimized the energy consumption in a banked scratchpad [2],[18]. The main input is the graph of the distribution of memory accesses to the scratchpad address space. The technique achieves a complete exploration of the banking solutions when the upper bound \( M \) of the number of banks is known. A backtracking mechanism – that relies on the monotonic increase of the energy consumption with the scratchpad size – prevents visiting the partitioning solutions that cannot be optimal in terms of their energy costs; however, the exploration is done such that the optimality of the final solution is guaranteed.
We implemented this *optimal* exploration model on a PC with an Intel Core 2 Quad 2.83 GHz processor and we applied it to the trace of *read/write* memory accesses to a scratchpad of 8 Kbytes obtained from the illustrative code in Figure 1.7.

The energy cost was computed using CACTI 6.5 [36] – an analytical tool that takes a set of scratchpad, cache, or DRAM parameters as inputs and calculates times of memory accesses for read/write operations, energy spent per access, static power, and chip area. The number of clock cycles – obtained by simulation – is 7,783,682 clocks. The number of accesses to A-elements is 5,187,840 accesses.

The energy consumption for a monolithic scratchpad ($M=1$) is 67.05 $\mu$J, assuming a technology of 32 $nm$ and a frequency of 400 MHz. For $M = 4$, the energetically-optimal bank partitioning obtained by the recursive partitioning proposed by Benini et al. for the scratchpad addresses of the bank boundaries: [0, 2655, 4078, 5533, 8192]; the corresponding minimum energy consumption was 32.34 $\mu$J. This result was obtained after a computation time of almost 1 hour, based on the analysis of 40.814 billion banking configurations. When the value of $M$ was increased beyond 4, the optimal exploration became computationally infeasible.

Our experiments show that the exploration for an energetically-optimal solution [18] is computationally expensive, possibly infeasible, for larger values of $M$ and/or larger values of the scratchpad size. The exploration algorithm has an exponential complexity, but the significant increase of the running times for $M \geq 4$ is mainly due to the large number of partitions that must be analyzed, although the backtracking mechanism prevents the visit of the entire search space.

A second experiment was performed using an exploration model based on dynamic programming [3]. The main input is the graph of memory accesses during the execution of the application code, using as main data structure an array whose dimensions are the sizes of the scratchpad and of the aforementioned graph (denoted $C$ and $N$, respectively). The array elements are profit values targeting, alternatively, performance optimization or energy...
optimization. The silicon area is indirectly taken into account by increasing the indexes of the profits computed during the dynamic programming algorithm by values depending on ratios of scratchpad areas. Since these ratios are, typically, non-integers (while indexes must be integers!), the algorithm needs tuning. Moreover, whenever the scratchpad model or technology is changed, the tuning should be done again. Although the time complexity is polynomial \( \theta(N.C^2) \) [3], [11], the computation time increases significantly with that is the size of the scratchpad. For instance, considering the execution trace of the code in Figure 1.7 such that \( N=C \), and iteratively doubling the scratchpad size from 512 to 8192 words, the computation times were 5.3, 46.1, 430.8, 3574.2 seconds, and over 8 hours, respectively.

1.4 Research Motivation

The main motivation of this research can be summarized as:

1. **The lack of a model designed for banking multidimensional signals.** Most models presented in previous research study the problem of scratchpad memory partitioning for embedded applications while dealing with the signals as a set of simple scalar variables. Even though these techniques can be applied to array signals as well—theoretically—by simply decomposing arrays to their scalar elements, this process can be time and space exhausting when practically applied to the class of data-intensive applications targeted by our research. This speculation is confirmed by the experiments we conducted for previous models whose results are presented in subsequent chapters of this thesis. It is also found that applying such techniques to data-intensive applications can be computationally infeasible.

2. **The lack of a banking model that takes as input the high-level behavioral specification.** Most models presented in previous research require a memory footprint (or trace) for the embedded specification prior to finding an optimized partitioning of the on-chip scratchpad memory. Finding such a trace requires in turn running a simulated execution of the given embedded program. Obviously, finding the memory trace of a data-intensive program through simulated execution is another time and space exhausting process, since it involves keeping track of accesses to a vast number of memory locations amid
execution. This motivated us to develop an approach that starts from a behavioral specification rather than from a memory trace.

3. **The lack of a banking model that allows for the optimization of the three factors: power, performance and/or chip area as possible targets for the scratchpad partitioning process.**

   From the literature survey, a general impression can be made regarding the main objectives for scratchpad memory partitioning in previous work. The most common objective for partitioning the on-chip scratchpad memory was to reduce the power consumed by the memory accessing. Performance is rarely considered as an alternative optimization target. Chip area is obviously not being studied with the same degree of concern. This motivated us to take directly into account energy consumption, performance, and area metrics when devising a banking model.

4. **The inefficiency of prior banking models when the number of banks is increased.**

   Preliminary results discussed in section 1.3 showed the incompetency of the previous methods when the specified number of banks increases. There is a need for a technique, different from the prior solutions that would be efficient with a larger number of banks.

5. **The lack of an efficient model that banks large-sized scratchpad memory.**

   The previous techniques explored in section 1.3 ensure the lack of an efficient and accurate banking model for scratchpad memories of larger sizes.

### 1.5 Objectives of the Study

The main objectives of this research are:

1. Study the state-of-the-art methods for on-chip memory banking.
2. Devise a technique for memory partitioning such that the exploration of the banking solutions be steered by the intensity of memory accesses within the array space of the multidimensional signals.
3. Evaluate the energy savings in the memory subsystem entailed by the novel approach, as well as the performance and chip area of the devised technique in comparison with other state-of-the-art approaches.
1.6 Document Organization

The rest of the document is logically divided into three parts, briefly described below.

The first part (Chapter 2) presents the memory simulation tool used to obtain energy, performance and area measurements in this research. The interface developed for the tool to achieve the experimental results is also introduced.

Chapters 3, 4, 5 and 6, standing for part two, are the core of the experimental work done. They show different approaches tackling the specific problem of scratchpad memory partitioning defined in the previous section; some of which were originally initiated in this research.

The third and last part of the thesis discusses the main conclusions attained by the research, draws attention to the limitations of the work and possible enhancements that may be applied in future work and nails new ideas for researchers interested in the field.

The appendices at the end are memory simulation data obtained for use in the experimental part. These tables were achieved after hours of running the interface tool presented in part three with different memory configurations and may be used as reference for future research.
2 COLLECTING SIMULATION DATA

This chapter gives a thorough overview of the simulation tool used to obtain memory data concerning power, access times and area used in this research. The used tool is CACTI 6.5 [36]. An interface to CACTI 6.5 has been implemented and was used to get the data displayed in Appendix A – Simulation Data. This chapter also presents a guide on how to use the developed interface to get batch simulation data and discusses possible sources of inaccuracy initiated by the tool and inherited in our work.

2.1 About CACTI Simulator

In this section, we display a quick review of CACTI in general; what it is, its origin, its forms and how to use it.

2.1.1 Definition

CACTI is a cache, scratchpad and RAM memory simulation tool [24]. It integrates models based on HSPICE\(^3\) simulations[9] for access time, cycle time, area, leakage, and dynamic power together and grants the confidence that tradeoffs between time, power, and area are all based on the same assumptions. Hence, the performance, energy and area results given by CACTI for different memory parameters may be considered mutually consistent [5]. According to CACTI developers, the tool is intended for the use of computer architects who need to better understand the performance tradeoffs inherent in memory system organizations.

Please note that the tool meant in our context is different from another well-known Cacti software; a network graphing tool topping Google's hits for "cacti". Probably it is more common than our simulator since it is a commercial tool, while our simulator is a research project.

2.1.2 Quick History and Origin

CACTI was originally developed as an analytical model for calculating the access and cycle times of direct-mapped and set-associative caches only in

---

\(^3\) HSPICE is an optimizing analog circuit simulator produced by Synopsys. It is used to simulate electrical circuits in steady-state, transient, and frequency domains. It provides fast, accurate circuit and behavioral simulation, facilitating circuit-level analysis of performance and yield, by using Monte Carlo, worst-case, parametric sweep, and data-table sweep analyses [74].
1994 by Steve Wilton and Norm Jouppi [9], from The Western Research Laboratory (WRL); a computer systems research group founded by Digital Equipment Corporation in 1982 and focusing on computer science research relevant to the design and application of high performance scientific computers. It was built as a research prototype developed and tested by designing, building, and using real systems.

Power analysis was later integrated in the tool by Reinman and Jouppi in year 2000 [5]. In 2001, Shirakumar and Jouppi integrated area analysis [7]. The tool did not include simulation models for memory types other than cache until 2007 when Manulimanoohar, Balasubramonia and Jouppi added scratchpad and DRAM to the supported types of memory [24]. The tool was later acquired by HP Labs in 2009[8], while still remaining as an open-source research project. Until the time of writing this thesis, CACTI is not a commercial product, and is not intended to become one[36].

2.1.3 CACTI Forms

HP Labs made available two forms of CACTI [36]: a web-based version and a C++ source code version.

The web interface version is frequently updated with the latest versions and bug fixes, and allows CACTI to be easily accessible to a larger user community. The web interface can be accessed online from [47].

In addition, CACTI source code is still available in C++ language for modification and integration into other tools especially for research support.

2.2 The Used Version is CACTI 6.5 (Offline Form)

Because the team at HP Labs announced that previous web versions of CACTI will not remain available online once newer versions are released [36], and because it is much more convenient for modification, integration and research support, we chose to use the offline source of CACTI for obtaining the simulation data in this research.

For consistency issues, it is recommended by CACTI's team in HP Labs (HPL) to use a single version of the source code since results for specific memory configurations and technologies change as new versions of CACTI are released and more bugs are fixed due to continuous upgrading [36]. The
latest version of CACTI as of the start of this work was 6.5. Hence, we used CACTI 6.5 in our study and continued using it even though newer versions may have become available by the end of this research.

CACTI 6.5 is a significantly enhanced version that includes major extensions over its predecessor--release notes can be found in [24]. CACTI 6.5 supports 32, 45, 68, and 90 nm technologies.

2.3 Using CACTI's Offline Version

To run the tool, you need to install it successfully on your machine. The only system requirement for running CACTI is a Linux system with a C compiler installed. The size of the tool is trivially small and it does not require a significant amount of memory for installation neither while running.

The tool takes as input a configuration file containing the parameters of the memory module to be simulated, like its type, size, technology, associativity—if applicable, ports and a lot of other attributes. The generated output is a file containing simulation data about the specified memory module; like access time, access energy(dynamic and static), area and other data; see Figure 2.1.

![Diagram showing the inputs and outputs of CACTI.](image)

Figure 2.1 A diagram showing the inputs and outputs of CACTI.

The configuration file must be in the same format of the given sample .cfg file provided with the tool. A miss-formatted configuration will generate an error and prevent the tool from correct execution. The command used for invoking CACTI with a given configuration file and saving the results in res.out is: ./cacti in.cfg -o res.out. The output file is generated in the same directory of CACTI or in the desired directory, if specified.

2.4 Using CACTI to Generate Batch Simulation Data

In our research, we are going to need simulation data for different SPM sizes—at least the sizes from 0 to 8K bytes must be available. Running CACTI individually for each and every configuration then extracting the resulting 8192 output files to get the needed data is a process that will
obviously exhaust our time and effort before we can even start tackling our main research problem. Thus, we had to think of a way to automate the gruesome process of collecting the simulation data needed for our research. Consequently, we implemented the batch generator program that aided us in completing the task.

It is true that we consider only scratchpad memories and assume a technology of 32 nm in this research; however, the work may be extended for other types and technologies as well in the future. The batch generator program is therefore highly re-usable for generating simulation data for any type of memory with any size and under any technology, under the condition that they are supported by CACTI.

To run the batch generator program implemented on top of CACTI, you need to:

1. Install any version of CACTI on your machine. Make sure it works correctly by testing it with any configuration file. See the previous section (Using CACTI's Offline Version) for more details about using CACTI for a single file.
2. Save the interface file: `cacti13.cpp` and the configuration file: `temp.cfg` together in the same directory.
3. Update the path to the installed version of CACTI in the file: `cacti_if.cpp`, line 178.
4. Compile `cacti_if.cpp` using the g++ command, as follows:
   ```
g++ cacti_if.cpp -o cacti_if
   ```
5. Run the compiled file with the following parameters:
   ```
./cacti13 [start] [end] [step] [technology] [type]
   ```
   where `start`, `end` and `step` represent memory sizes in bytes. The available technologies for the `technology` field are 32(default), 45, 68 and 90 nm. And, `type` can have any value from 1 to 3; such that 1 stands for cache, 2 for scratchpad (default) and 3 for main memory (or RAM).

The program gives the time elapsed for running. The final results are saved in a file named after the type, technology and sizes. For example, to simulate a cache memory module --Type 1 memory-- with sizes of 512 bytes, 528 B, 544 B, up to 1024 bytes--that is a step of 16 bytes--in the 32 nm technology, run the command:
```
./cacti 512 1024 16 32 1
```
which generates the file: `cache32nm512:1024:16.txt` containing the desired data shown in Figure 2.2. Upon wrong invocation, an error message that explains the command in detail appears.  

<table>
<thead>
<tr>
<th>Technology: 32 nm</th>
<th>Memory Type: Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (B)</td>
<td>Dyn. Energy (nJ)</td>
</tr>
<tr>
<td>512</td>
<td>0.00411748</td>
</tr>
<tr>
<td>528</td>
<td>0.00414604</td>
</tr>
<tr>
<td>544</td>
<td>0.00404604</td>
</tr>
<tr>
<td>560</td>
<td>0.0041746</td>
</tr>
<tr>
<td>576</td>
<td>0.0041746</td>
</tr>
<tr>
<td>592</td>
<td>0.00420316</td>
</tr>
<tr>
<td>608</td>
<td>0.00420316</td>
</tr>
<tr>
<td>624</td>
<td>0.00423172</td>
</tr>
<tr>
<td>640</td>
<td>0.00423172</td>
</tr>
<tr>
<td>656</td>
<td>0.00426028</td>
</tr>
<tr>
<td>672</td>
<td>0.00426028</td>
</tr>
<tr>
<td>688</td>
<td>0.00428884</td>
</tr>
<tr>
<td>704</td>
<td>0.00428884</td>
</tr>
<tr>
<td>720</td>
<td>0.0043174</td>
</tr>
<tr>
<td>736</td>
<td>0.0043174</td>
</tr>
<tr>
<td>752</td>
<td>0.00435996</td>
</tr>
<tr>
<td>768</td>
<td>0.00435996</td>
</tr>
<tr>
<td>784</td>
<td>0.00437452</td>
</tr>
<tr>
<td>800</td>
<td>0.00437452</td>
</tr>
<tr>
<td>816</td>
<td>0.00440308</td>
</tr>
<tr>
<td>832</td>
<td>0.00440308</td>
</tr>
<tr>
<td>848</td>
<td>0.00443264</td>
</tr>
<tr>
<td>864</td>
<td>0.00443264</td>
</tr>
<tr>
<td>880</td>
<td>0.0044602</td>
</tr>
<tr>
<td>896</td>
<td>0.0044602</td>
</tr>
<tr>
<td>912</td>
<td>0.00448876</td>
</tr>
<tr>
<td>928</td>
<td>0.00448876</td>
</tr>
<tr>
<td>944</td>
<td>0.00451732</td>
</tr>
<tr>
<td>960</td>
<td>0.00451732</td>
</tr>
<tr>
<td>976</td>
<td>0.00454588</td>
</tr>
<tr>
<td>992</td>
<td>0.00454588</td>
</tr>
<tr>
<td>1008</td>
<td>0.004578</td>
</tr>
<tr>
<td>1024</td>
<td>0.004578</td>
</tr>
</tbody>
</table>

Figure 2.2 Snapshot of a sample output file (cache32nm512:1024:16.txt) generated by the batch generator.

What actually takes place behind the scenes in the generator program is shown in Figure 2.3. First, all the configuration files of the memory modules in the specified range are generated using the same format as in the temp file provided with CACTI in step 2 above. These configuration files are saved in a folder named `cactiCfgs`. Then, CACTI is invoked with each and every configuration file.

4 If you notice any anomalies in the batch generator or face any difficulties using it, please send to noha_abuaesh@aucegypt.edu.
configuration file in `cacti_cfgs` and the resulting output is saved to another folder called `cacti_outs`. After CACTI is done processing all the configuration files and all output files are generated and saved in their corresponding folder, the program then extracts the performance, dynamic and static energy as well as area information from all the output files in the `cacti_outs` folder and stores them to the corresponding columns in the results file(`cache32nm512:1024:16.txt`). Finally, the elapsed time taken to find the required results since the command was initiated is displayed in the command window.

![Workflow of the batch generator.](image)

Figure 2.3 Workflow of the batch generator.

It may take considerably long time to generate large batches of simulation data. For example, generating the data for scratchpad sizes up to 8K bytes with a step of 4 bytes—that is, running CACTI for 1920 different memory configurations and extracting the results into one file—took more than an hour to complete. However, the program not only saves a much-much-longer time that would have elapsed if the process was otherwise done manually, but also
eliminates a tedious amount of file processing that the user would have went through in the other case.

Nevertheless, the efficiency of the simulation data batch generator is not really important since it will be run only once in a lifetime and once the tables in Appendix A – Simulation Data are generated with the needed memory settings; the generator will not be needed any more.

### 2.5 Limitations and Shortcomings

A major limitation that we faced while using CACTI for generating simulation data is that it did not retrieve data for memory modules of sizes less than 512 bytes. As a solution, we used extrapolated values for smaller memory sizes.

In addition to not being able to retrieve reliable simulation data for memory sizes below 0.5 KB, another difficulty is that CACTI as a tool is not very user friendly. Perhaps the reason is that it is basically intended for research purposes and not for commercial distribution. Even the batch generator we developed lacks a user-friendly interface. Nevertheless, we did not wish to put needless effort in developing a GUI since the command-based interface is not overly complicated and because once the tables in Appendix A – Simulation Data are generated, the tool will be needed no more.

As mentioned earlier, CACTI is based on HSPICE models. According to CACTI's technical report, the generated estimates are within 10% of HSPICE results for selected circuits used in testing CACTI's accuracy [9]. This inaccuracy will consequently be inherited in our work since CACTI's simulation data are the base for our experiments and results.

As announced in HP Labs official website [36], the systems built are research prototypes; they are not intended to become products; which means that they are continuously under development and are vulnerable to major changes. The continuous upgrading of CACTI implies variable simulation results for specific cache configurations and technologies as new versions are released and bugs are fixed. Therefore, the work depending on simulation data obtained from CACTI needs to be periodically updated with data attained by running the latest version of CACTI.
3 A RECURSIVE APPROACH

Early work done by Benini et al. [2], [18], [31], addressed the problem of scratchpad memory partitioning for energy optimization. They introduced a recursive method for computing the minimum energy partition of an on-chip memory into $M$ banks. The partitioning is carried out according to the dynamic memory access profile of the embedded application. Interestingly, their algorithm is guaranteed to find the global optimum taking into account the hardware and wiring overhead due to adding extra memory banks.

This chapter studies the recursive algorithm proposed by Benini et al and investigates the feasibility of applying it to data-intensive applications. The first section merely describes the algorithm according to Benini et al.'s published paper in 2000[2]. The second section critically analyzes the complexity of the algorithm. Finally, testing results are discussed and an overall evaluation of the approach is made in the last section.

3.1 Inputs

The proposed partitioning algorithm takes as input the memory read/write pattern resulting from simulation of the embedded program; represented in the $r$ and $w$ arrays that are used in cost calculation. It also requires a hard bound on the maximum number of partitions; $M$.

The dynamic access profile for the target embedded application is given as a pair of arrays $r = [r_0, r_1, ..., r_{M-1}]$ and $w = [w_0, w_1, ..., w_{M-1}]$ where $r_i$ is the number of reads to address $i$, and $w_i$ is the number of writes to address $i$. The total energy consumed by a memory containing a given range of addresses is a technology-dependent metric that can be expressed as a function $\text{MemE}(lo, hi, r, w)$, where $lo$ and $hi$ are the maximum and minimum addresses in the range.

Furthermore, an array $\Delta E = [\Delta E_{0,1}, \Delta E_{1,2}, ..., \Delta E_{M-1,M}]$ is defined to express the energy overhead of adding one more bank to a partitioned memory. In other words, $\Delta E_{i,i+1}$ is the amount of additional energy expected to be spent in selection logic and memory buses when moving from a memory organization with $i$ banks to one with $i + 1$ banks. The power savings obtained by partitioning must compensate the overhead. Clearly, the exact value of the
energy overhead is not known before the memory is completely designed. Hence, $\Delta E$ just provides a conservative bound: it is to prevent partitioning when power savings are dubious.

A memory partition is a set of memory banks that can be independently selected. Any address $0 < i < N$ is stored into one and only one bank. The total energy consumed by a partitioned memory is the sum of the energy consumed by all its banks. Given these definitions, the memory partitioning problem can be formulated as:

*Given* $w, r, \Delta, MemE$ and $M$, *find a partition of a* $N$-word memory *with at most* $M$ *banks that minimizes the total energy.*

We will first introduce the proposed solution to the memory partitioning problem, and then, in the subsequent section, we shall focus our attention on the cost metrics employed for estimating memory energy.

### 3.2 The Recursive Algorithm by Benini et al.

Algorithm 3.1 shows the pseudo code of the recursive function introduced by Benini *et al.* to solve the scratchpad partitioning problem. It defines the procedure $Part$ that receives as input the recursion index $n$, the current maximum depth of the recursion $M$, the starting memory address of the current block to be partitioned $i$, the current total energy $TotEnergy$, and the current energy budget $Budget$. The procedure is first invoked as $Part(1, 2, 0, MemE(0, N, r, w), 0)$, that is, with initial budget equal to the cost of a monolithic memory of $N$ words, and with total energy initialized to 0.

```
1 Part(n, M, i, Budget, TotEnergy)
2   Budget -= $\Delta E_{n+1}$
3   if Budget < 0 then
4       return
5   for cut := i to N do
6       CurrTotE := 0
7       MemEnergy := MemE(i, k, r, w)
8       NewB := Budget - MemEnergy
9       if NewB < 0 then
10          return
```
Algorithm 3.1 Recursive Algorithm by Benini et al. [2]

The algorithm is recursive; at a given recursion depth, it computes the optimal partition (of up to \( M \) blocks) of the memory portion between \( i \) and its upper limit \( N \). In Line 2, the currently available power budget, \( \text{Budget} \), is reduced by a factor corresponding to the energy penalty due to adding an extra memory bank. If this new budget becomes negative, no further solution can be found using \( n \) memory blocks, and execution resumes at the upper recursion level (Line 3). Figure 3.1 visualizes the process of finding the position \( i \) at which the initial cut (\( i, k \)) is made during the first call to \( \text{Part} \). A partitioning is beneficial only when:

\[
E_{\text{Bank1}} + E_{\text{Bank2}} + \Delta E_{1,2} < E_{\text{Monolithic_SPM}}
\]

If some budget is still available, the exploration of all possible partitions is started from the current cut \( i \) to the end of the memory (Line 5). A local energy cost is initialized at each iteration (Line 6), and the cost of the partition in the generic iteration \( \text{MemEnergy} \) is computed using the cost function \( \text{MemE} \) (Line 7). The resulting cost is subtracted from the current budget, and assigned to the budget of the iteration loop \( \text{NewB} \) (Line 8). This "local" budget is used to restrict the search space (Lines 9 and 10); the rationale here is that if the cost of the currently analyzed block \( ([i,k]) \) exceeds the current budget, it is useless to continue with this iteration. If this is not the case, the current energy cost is added to the current total, and considered for inclusion in a solution (Line 12). If bottom of the recursion is reached (Line 13), the
current solution is completed by adding the cost of the remaining portion of memory (from the current cut to the end - Line 14). This complete solution can be stored as the new best solution if its energy cost improves the current one and it does not exceed the available budget (Lines 15 to 17). In order to continue in the iteration of Line 5, the last selection from the current solution is popped (Line 18).

Figure 3.1 An optimal 2-way partitioning is found by Minₖ{E_{Bank1} + E_{Bank2}} + ΔE₁₂. Optimal M-way partitioning is found using recursion.

The discovery of a new minimum allows restricting the search space, in terms of a reduction of the total budget. This additional optimization is not shown in the pseudo-code for the sake of readability. If the recursion can proceed, the current index $i$ is pushed onto the solution stack (Line 20), and recur by adding another memory block. The current budget and current total energy of the solution built so far are forwarded to the next recursion level (Lines 21 and 22).

### 3.3 Cost Calculation

The cost function used by Benini et al. to drive the partitioning process, denoted $\text{MemE}$, mainly evaluates two components of the memory energy: 1) dissipation per cycle, and 2) dynamic access profile.

To measure the memory energy dissipation per cycle, the authors adopted the energy model proposed by Coumeri and Thomas in [22]. The energy measurements of this model are empirically derived from simulation and are expressed in terms of high-level parameters such as size and bit-width. In
general, the adopted energy model consists of distinct equations for read and write operations; due to the different energy cost of the two operations.

The second component of $MemE$—which is the memory access profile—can be computed using an instruction-level simulator for a given application provided with the chosen processor core, as suggested by the authors. Even though this requirement of simulated execution prior to running the partitioning algorithm may not be as harmful to the overall efficiency as exhaustively exploring the search space as will be seen later, it is still worth mentioning that this requirement adds up the embedded application's running time to the algorithm's complexity.

Finally, the total memory energy $MemE$ is then given by the energy cost per access of a memory within the given bounds, $hi$ and $lo$, multiplied by the number of accesses to addresses within those bounds. $MemE$ can be formulated as:

$$MemE(hi, lo, r, w) = E_r(hi - lo) \sum_{i=lo}^{hi} r[i] + E_w(hi - lo) \sum_{i=lo}^{hi} w[i]$$

(1)

where $E_r(d)$ and $E_w(d)$ represents the energy consumption for a single read and write access, respectively, in a memory of $d$ words.

As can be noticed from the expression of $MemE$ in the equation above, $MemE$ is monotonically increasing with respect to the memory size; a feature that is exploited in the next section to prove that the worst case is unlikely to occur in practice.

### 3.4 Results and Discussion

In their paper, the authors display the testing results of the memory partitioning; validated on a set of benchmark applications run on an ARM processor. The results showed significant energy reduction with respect to the case of a monolithic memory.

The authors claim an average energy savings of about 42% improvement over the monolithic architecture for the tested benchmarks with $M$ values of 2 and 3. However, for most of the benchmarks presented in their work, the testing results are not available for $M > 3$ partitions. The reason is most probably due to the huge inflation in execution time for every increase in $N$ and $M$ values.
In fact, it is important to mention that the optimal solutions obtained by their algorithm takes a relatively large execution time. The execution time is obviously proportional to the number of distinct addresses used by the program, a conclusion that can be drawn from a careful study of the pseudo code in Algorithm 3.1.

For example, the authors mentioned that in their experiments the execution times ranged from about two minutes for the smallest benchmark to about three hours for the largest one. Referring to the benchmarks they used, we find that the smallest benchmark, the lirDemo benchmark, has a size less than 1K words and was tested for $M=3$, while the largest benchmark, the DFT benchmark--sized above 15K words-- was tested for only 2 banks! A rational test would dictate setting the larger $M$ value for the larger benchmark. But since running with $M=2$ took a couple of hours, we may conclude here that testing the DFT benchmark for larger $M$ values is evidently intractable.

Besides, we cannot really consider the obtained results as optimal since it is not guaranteed to be given an optimal value for $M$ in the first place. Therefore, it is significantly misleading to describe their results as the absolute optimal since it is likely to get even better results by increasing $M$ value. Alternatively, it can be stated that these results are optimal only under the given $M$ values.

For instance, focusing on the DFT benchmark, we find that a bi-partition solution takes about 3 hours of execution and achieves about 39% energy savings over a monolithic architecture. Whereas a benchmark like lirDemo achieved about 45% savings by a 3-partition solution.

To make this point evident, we tried running the recursive algorithm on the 8 KB data-intensive benchmark discussed in section 1.3 with different values of $M$ and different word widths; see Table 3.1. The experiments made to obtain the results in Table 3.1 were carried out on a PC with an Intel Core 2 Quad 2.83 GHz processor.

In the first group of tests, the algorithm runs with the smallest possible granularity, that is, a single memory word. In other terms, we did not restrict the sizes of the memory banks a-priori to be for example multiples of a minimum number of words. This choice is the only one that is guaranteed to
find the global optimum in the most general case. We were able to go as far as $M=4$ in almost an hour of execution. For greater values of $M$, the execution had to be terminated after a couple of days!

<table>
<thead>
<tr>
<th>Word Width (B)</th>
<th>Max</th>
<th>Energy (uJ)</th>
<th>No. of Analyzed Partitions</th>
<th>CPU Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>67.05</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>40.95</td>
<td>5332</td>
<td>2</td>
<td>0.38</td>
</tr>
<tr>
<td>3</td>
<td>34.58</td>
<td>18.18M</td>
<td>3</td>
<td>2.15</td>
</tr>
<tr>
<td>4</td>
<td>32.34</td>
<td>40.814G</td>
<td>4</td>
<td>3188.24</td>
</tr>
<tr>
<td>5</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>67.05</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>41.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>34.64</td>
<td>71043</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32.41</td>
<td>9.965M</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>31.23</td>
<td>1.0345G</td>
<td>80.71</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>30.7577</td>
<td>87.857G</td>
<td>6925.9</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 3.1 Testing results of the recursive algorithm with an 8KB benchmark. Entries left blank could not be generated due to intractable execution times.

Even by extending the word width to 16 bytes, we were not able to go beyond $M=6$; which took slightly below 2 hours of execution time. Obviously, discretizing the search space with a minimum cut granularity would sensibly cut run times, at the expense of the optimality. However, the loss of optimality is negligible as shows from Figure 3.2.
Figure 3.2 Total energy consumption for different number of partitions shown for two different values of word width. While increasing the number of partitions reduces the total energy, the figure suggests insignificant progress in energy consumption by decreasing the word width.

The rapid growth of execution time is apparent with every increase of $M$. Figure 3.3 suggests increasing the word width to allow for finding solutions with larger number of partitions. It is also noticeable from the conducted tests that the solution quality slightly increases with the maximum number of banks—at least for the $M$ values shown in the results table. The execution time needed to find partitioning solutions with different number of partitions is greatly affected by the word width. Also, increasing the word width allows finding solutions with larger number of partitions. Therefore, we cannot determine whether a solution is actually optimal unless we have explored different values of $M$; which can theoretically be as big as the value of $N$ itself!
From the above discussion, we may wrap up that the recursive algorithm at hand suffers from questionable optimality and intractability that makes it extremely inefficient when applied for large-sized benchmarks or increased number of partitions. We may conclude from the tests above that the method is inefficient when (1) the number of banks increases over 4-5 banks, and (2) when the SPM size grows above 2 Kbytes.

An argument that the authors raised to justify the tremendous execution delay is that it represents a one-time cost because, for given a memory model and access profile, the optimizer has to be run only once. Nevertheless, it may be preferable to have an algorithm that generates rival results while executing in relatively much less time.

The next chapters investigate other approaches inducted by other researchers in an effort to consider efficiency along with the solution's optimality.

### 3.5 Complexity and Analysis

A careful analysis of the structure of the problem and its cost metrics reveals that the algorithm has an exponential complexity of $\theta(N^M)$. Yet, the authors claim that, even though the algorithm has an exponential worst case run-time—that is, in the worst case, it exhaustively explores all possible partitions--the algorithm performs very well in practice. Their reason was that the algorithm
relies on two key properties to speed-up the search when making the optimum cut at every recursive bi-partitioning. The properties are:

i. The total energy consumption of a memory bank monotonically increases with increasing memory size, if the addresses stored in a larger memory are a superset of the addresses stored in a smaller memory.

ii. The number of memory banks, \( M \), in a partitioned architecture is much smaller than the total memory size \( N \).

Of course this is true when considering the simple case of bi-partitioning (\( M = 2 \)). The optimum solution can be found in \( O(M) \) time by iteratively moving the lower bound, \( j \), of the first bank from 1 to \( N-2 \). The total memory energy can be computed as \( TotE2 = MemE(0, j, w, r) + MemE(j + 1, M - 1, w, r) \).

A bi-partition is considered better than the single-bank solution with energy \( TotE1 \) if \( TotE2 < TotE1 - \delta \). The number of iterations can be reduced if, for a given \( j \), it was found that \( MemE(0, j, w, r) > TotE1 - \delta \). This early stopping condition is motivated by property i above: If a memory containing the range of addresses \([0,j]\) consumes more than \( TotE1 - \delta \), further iterations can be avoided because \( MemE(0, k, w, r) > MemE(0, j, w, r) \) for every \( k > j \).

Therefore, the simple case of two-way partitioning indicates that property i can be effectively exploited to create bounds and prevent the exploration of search space regions that do not contain the global optimum.

The second property may be used to reduce the search space for greater values of \( M \). The partitioning algorithm is invoked \( M \) times, to compute partitions with an increasingly larger number of blocks; an extra block being added with each invocation. The algorithm, eventually, moves from coarse partitions to finer ones that have a larger hardware overhead. The coarse-granularity solutions are exploited to tighten the bounds on the search of fine-granularity partitions by means of the \( M \) parameter.

While the aforementioned features may help in diminishing the search space to a certain extent, we need to make sure that the execution times are practically feasible for data-intensive benchmarks, where values of \( N \) can be relatively large.
3.6 Limitations and Shortcomings

This algorithm has some limitations in implementation. The adopted cost function, to begin with, takes only energy into account while totally ignoring access time and chip area. Second, the method is not connected to the way the signals were assigned to the memory layers. Third, no mapping function is mentioned (that is, how were the signals stored in the SPM and at what addresses.) Fourth, after assuming the assignment and mapping function are accomplished, the read/write accesses at each address is considered known, without a clue on how it was found.

Focusing on the efficiency, the method becomes computationally expensive or even infeasible when: the size of the SPM is large (relative to the size of the word); or the upper bound M of the maximum number of banks is increased.

The next chapter introduces a novel approach that exploits the dynamic programming methodology to solve our partitioning problem.
4 A DYNAMIC PROGRAMMING APPROACH

After verifying that exhaustively exploring all possible partitionings does not yield an efficient algorithm for larger values of $M$ and $N$ (see the discussion of Benini et al.’s recursive algorithm in the previous chapter), we shall now explore a different technique to solve the problem: the dynamic programming technique.

First, we study the feasibility of using a dynamic programming approach to solve the partitioning problem. After ensuring the optimality and applicability of the technique, we review in this chapter a dynamic programming approach proposed by Angiolini et al.[3].

4.1 About Dynamic Programming

This section verifies the optimality and applicability of a dynamic programming method to our problem. At the end of this section, a brief comparison between dynamic and linear programming is made to justify the selection of the former over the later.

4.1.1 Proof of Optimality

In dynamic programming approaches, all possible cases are evaluated just like a brute force algorithm. It is essentially a “smart” recursion. Often extra work doesn't have to be repeated if solutions to subproblems are cached after they are solved. Therefore, the only difference from exhaustive algorithms which makes dynamic programming significantly more efficient is the reuse of solutions to subproblems that have already been computed. That is why dynamic programming is guaranteed to always achieve the global optimal solution to any problem that has proven to be applicable to this method of solution.

4.1.2 Feasibility of Applying Dynamic Programming to the Partitioning Problem

Dynamic programming solves problems by combining the solutions to sub-problems using a tabular method. Unlike the divide-and-conquer method, dynamic programming solves problems where sub-problems are not independent, that is, subproblems share subproblems [49]. It solves a
subproblem only once and saves its answer in a table, thereby avoiding the work of re-computing the answer every time the sub-problem is encountered.

In general, dynamic programming algorithms are typically applied to optimization problems; which is, in fact, the case in our scratchpad partitioning problem. In the scratchpad partitioning problem, each solution has "a value" representing the energy, performance and/or area costs of the partitioning represented in that solution. We wish to find a solution with the optimal value, that is, minimum cost. It is important to notice that we are interested in finding an optimal solution to the problem, since it is possible to have several solutions that achieve the optimal value.

It seems from this introduction that our scratchpad partitioning problem is a perfect fit for a dynamic programming approach. However, let us prove it from an engineering perspective before making a hasty judgment.

4.1.3 Does the scratchpad partitioning problem have the elements of dynamic programming?

According to T. Cormen et al. [49], there are two key ingredients that an optimization problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblem.

4.1.3.1 Optimal Substructure

A problem is said to have an optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. Whenever a problem exhibits optimal substructure, it is a positive sign that dynamic programming might apply. In dynamic programming, we build an optimal solution from optimal solutions to subproblems.

To discover the optimal substructure property, we need to prove that a given solution to a problem cannot be optimal unless it uses optimal solutions to the contained subproblems. Applying this method of proof to our scratchpad partitioning problem, we let $P_{ij}$ denote the optimal partitioning solution to the address range from $i$ to $j$. Therefore, for any partitioning for the address range $Q_{ij}$, the following is true:

$$\text{cost}(Q_{ij}) \geq \text{cost}(P_{ij}) \text{ where } Q_{ij} \neq P_{ij}$$
Assume that \(P_{ij}\) optimal solution divides the range \(i\) to \(j\) at address \(k\), and according to the optimal substructure property, solutions to the subproblems must be optimal as well, hence:

\[
Cost(P_{ij}) = cost(P_{ik}) + cost(P_{kj})
\]

where \(P_{ik}\) and \(P_{kj}\) are both optimal solutions for address ranges \(i\) to \(k\) and \(k\) to \(j\), respectively.

Now, using proof by contradiction, we try to prove that we can get an optimal solution that costs less than—or equal to—\(P_{ij}\) without using optimal solutions to the subproblems.

Assume that there exists another optimal solution \(Q_{ij}\) that makes a partition at \(k\) but does not take the optimal solutions for the subproblems \(i\) to \(k\) and \(k\) to \(j\).

That is:

\[
Cost(Q_{ij}) = cost(Q_{ik}) + cost(Q_{kj})
\]

where \(Q_{ik}\) and \(Q_{kj}\) are not optimal solutions for address ranges \(i\) to \(k\) and \(k\) to \(j\), respectively. That is, \(cost(Q_{ik}) > cost(P_{ik})\) and \(cost(Q_{kj}) > cost(P_{kj})\).

Consequently:

\[
cost(Q_{ik}) + cost(Q_{kj}) > cost(P_{ik}) + cost(P_{kj})
\]

\(cost(Q_{ij}) > cost(P_{ij})\) --Contradiction!

which contradicts our previous conjecture of \(Q_{ij}\) optimality. Thus, an optimal partitioning for any address range must use optimal partitioning for subranges, too. We have now proved the optimal structure property for the scratchpad partitioning problem.

4.1.3.2 Overlapping Subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be “small” in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.

---

\(^5\) In this calculation of partitioning cost, we are ignoring the cost of the overhead produced by adding an extra partitioning to the solution since, even if this overhead is not to be neglected, it does not harm our argument here since it will always be added for any partitioning whether it is optimal or not.
Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems. In contrast, a problem for which a divide and conquer approach is suitable usually generates brand-new problems at each step of the recursion.

Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

It is obvious that the scratchpad partitioning problem has overlapping subproblems. Figure 4.1 shows a trivial example of a memory space of only 4 addresses and the space subproblems under it. The dimmed subproblems denote an overlap with one of the undimmed subproblems; the overlapping of subproblems can be clearly noticed.

![Figure 4.1](image)

**Figure 4.1** The recursion tree for computing scratchpad partitioning solution for an address range 1 to 4. The darkened subtrees are overlapping subproblems. Figure from [49].

### 4.1.4 Dynamic Programming versus Linear Programming

A main advantage of dynamic programming is that it allows for any cost function to be used in optimization. Linear programming, on the other hand, restricts the optimization problem to the case where the optimization function is subject to only linear constraints. Linear programming becomes applicable only if we can specify the objective as a linear function of certain variables, and if we can specify the constraints on resources as equalities or inequalities on those variables [71]. Since our optimization problem is subject to non-linear constraints—namely, energy consumption, access time delay and chip
area—considering linear programming for our problem is therefore not a feasible option.

After proving that dynamic programming can actually find optimal solutions without restricting the optimization functions to be linear and that our scratchpad partitioning problem has both the optimal substructure and overlapping subproblems features, we can now confidently consider a dynamic programming approach to solve it.

4.2 A Dynamic Programming Algorithm by Angiolini et al.

In this section, we are going to dissect an interesting algorithm by Angiolini et al. in 2005 [3], [25] that deploys dynamic programming to solve our scratchpad partitioning problem. This algorithm is integrated in a complete and automated design, simulation, and synthesis flow; which we will thoroughly explore in the next section.

The research we are about to examine describes an algorithm to solve the mapping problem aside with the partitioning problem by means of dynamic programming applied to a synthesizable hardware architecture. In general, the algorithm works by mapping segments of external memory to physically partitioned banks of an on-chip scratchpad. According to the authors, this architecture provides significant energy savings.

An advantage of the algorithm is that it does not require any user-set bound on the number of partitions and takes into account partitioning overhead. Improving on previous solutions, execution time is polynomial in the number of memory locations, even in the most general solving policy.

Even though strategies to optimize memory requirements and speed of the algorithm are exploited, a major concern here is the feasibility of deploying the algorithm to data-intensive applications where the number of memory locations is relatively large.

4.2.1 Overview of the Design Flow

As in the design flow presented in chapter 2- A Memory Management Framework, the target hardware architecture suggested by Angiolini et al. is assumed to be fully open for design, up to the layout level.
According to [3], the scratchpad partitioner requires application-independent inputs, like the size of the target scratchpad (denoted \( C \)) and a quantitative estimation of the hardware overhead associated with any further partitioning of the scratchpad (denoted \( \tau \)). The authors recommend extrapolating the latter with back-annotation according to the designer’s priorities and adjust it to accurately reflect area, delay, or power overheads, or an average of them all as mentioned in [18] and related works.

![Schematic Design Flow](image)

**Figure 4.2 Schematic Design Flow presented by Angiolini et al. in [3]**

In addition to such designer-provided inputs, the partitioner is linked to the underlying development platform as shown in Figure 4.2. The task of the filters block in the diagram is not clearly mentioned in the paper. The reason behind the need of the filters may be due to extracting the memory access pattern from the execution footprint, unlike the method used in 6.2.3.1.

It is assumed that some preprocessing engine generates the execution traces of the target application. It is important to mention the effect of this step on the algorithm's overall complexity; adding up the entire application's running time to the algorithm's complexity.

Eventually, the traces are passed to the scratchpad partitioner for analysis. Once the best scratchpad partitioning is found, the second link to the design
platform comes into play. The algorithm’s output is fed back to the simulation engine, and a second simulation run is launched. Providing complete reporting on the efficiency of the implementation, specific detection routines make it possible to compare the results of the currently-generated scratchpad partitioning versus the partitioning solutions found in previous runs.

A core weakness in this flow is that the execution trace is obtained by simulation. It is extremely important to highlight the inconvenience arising from this process due to the need to carry out simulated execution every time a new partitioning is evaluated; multiplying the algorithm's complexity by the entire application's running time.

In addition, the assignment and mapping methods (that is, deciding which signals to be assigned to the scratchpad, and how their mapping is performed) are not mentioned even though these steps must be taken care of before tackling the partitioning problem.

4.2.2 Cost Calculation

Because of the way Angiolini et al. structured their dynamic programming algorithm, it is more convenient to first understand their method for evaluating solutions fitness before delving into the algorithm's details.

In their algorithm, the authors employed two different functions to evaluate the fitness of a given partition containing the address range from \(i\) to \(j\): a profit function and a cost function, denoted \(p(i, j)\) and \(w(i, j)\), respectively.

For \(p(i, j)\), the authors defined two different profit functions; one aimed for optimizing the dynamic energy profit \((p_{\text{energy}})\) and another for speed \((p_{\text{speed}})\), as follows:

\[
p_{\text{speed}}(i, j) = \sum_{h=i}^{j} \pi_h
\]

\[
p_{\text{energy}}(i, j) = \left( \sum_{h=i}^{j} \pi_h \right) \cdot (\Delta - \delta_{a_j-a_i+1})
\]

where:

\(\pi_i\) is the number of access to the \(i^{th}\) sorted trace entry.
\( \Delta \) is the amount of energy spent to access the external memory, and 
\( \delta_k, (k = 1, \ldots, C) \) is an array describing the energy spent to access scratchpad memory banks holding \( k \) words. 
\( \alpha_i \) is the actual memory address.

In other words, \( p_{\text{energy}}(i, j) \) represents the energy savings achieved by mapping the range onto the scratchpad in a separate partition, while \( p_{\text{speed}}(i, j) \) indicates the speed profit of such a mapping.

According to the paper, the user\(^6\) needs to select between these two profit functions depending on the application objectives; either optimizing energy consumption or optimizing speed but not both. This harsh trade-off gives no chance for a synergy between energy and performance; a shortcoming that we will try to overcome in the next section.

The other function introduced by Angiolini et al. is the cost function, \( w(i, j) \) that determine the validity of a given partition in terms of its physical area. It is calculated according to the following function:

\[
w(i, j) = \frac{\alpha_{a_j-a_{i+1}}}{\alpha_C} \cdot C + \tau \tag{4}
\]

where:

\( \alpha_k (k = 1, \ldots, C) \) is an array characterizing the silicon area corresponding to memory cuts holding \( k \) words,
\( C \) is the size of the scratchpad in 32-bit words.
\( \tau \) is the overhead associated with adding an extra partition.

The authors assumed \( \alpha_k = \{10 \, \mu m^2, 15 \, \mu m^2, 25 \, \mu m^2, 40 \, \mu m^2\} \) for memories capable of holding 1, 2, 4 and 8 words, respectively, while interpolating the missing values. They also assumed the partitioning overhead to be equivalent to one word; that is \( 10 \, \mu m^2 \).

In theory, the cost function symbolizes the area cost of a given memory partition based on the range length (in words), which gets subsequently biased by comparing the area of the target partition to the area of an unpartitioned scratchpad.

\(^6\) The word “user” here refers to the embedded system designer/engineer likely to use the partitioning software.
However, it is important to mention here that despite the authors' assumptions regarding the variables used to calculate $w(i,j)$, in practice, none of these variables is guaranteed to have an integer value except $C$, which represents the size of the given scratchpad in 4-Byte words. In fact, even if the values used for $\alpha_k$ and $\tau$ were integers, the result of the division $\frac{\alpha_j - \alpha_i + 1}{\alpha_C}$ is never integer—except in the case of an unpartitioned scratchpad where $\alpha_i = 1$ and $\alpha_j = C$. Consequently, a realistic value of $w(i,j)$ cannot be assumed to be integer. This fact will greatly affect the algorithm as will be seen later.

### 4.2.3 The Algorithm

The algorithm at hand takes as input the number $N$ of addresses in the sorted execution trace, along with the actual address $\alpha$ and the number of accesses $\pi$ associated with the $i$th sorted trace entry, the size $C$ of the scratchpad, and the penalty $\tau$ for mapping noncontiguous ranges in the scratchpad. Note that $\alpha$, $C$ and $\tau$ are all expressed in 32-bit words.

```
  for i := 1 to N do
    for j := i to N do
      if w(i, j) > C then
        break
      for k := 1 to C - w(i, j) do
        if P[k][i] + p(i, j) > P [k + w(i, j)][j] then
          P [k + w(i, j)][j] := P[k][i] + p(i, j)
  copyprofits(i)
```

**Algorithm 4.1** Pseudo code for the Angiolini et al. algorithm.

For each pair $[i, j]$, $1 \leq i \leq j \leq N$, let $p(i, j)$ and $w(i, j)$ be the profit and (space) cost associated with the range of addresses from the $i$th to the $j$th. The objective of the problem is to select a set of mutually disjoint address ranges, the overall profit of which is maximized, and the overall cost of which does not exceed $C$.

The algorithm's main structure is a matrix-shaped data structure with $C$ rows and $N$ columns (see Figure 4.3). Every cell of the matrix contains two pieces of information about a possible partitioning of the scratchpad: its overall profit (according to one of the two strategies: $p_{\text{energy}}(i,j)$ or $p_{\text{speed}}(i,j)$) and its list of ranges.
The cell in row \( k \) and column \( j \) contains data about the best partitioning found so far, starting from the actual address \( \alpha_1 \) up to the last address of which is less than or equal to the \( j \)th address of the trace \((\alpha_i)\) and the overall cost of which amounts to \( k \) 32-bit words in the scratchpad. The recursive definition of the profit values \( P[k][j] \) for the matrix \( P \) are given by:

\[
P[k][j] = \max \{P[k][j-1] \mid \forall i \leq j, w(i, j) \leq k\} \max P[k-w(i, j)][i]
\]

for \( j = 1, \ldots, N \) and \( k = 1, \ldots, C \), where \( P[k][0] = 0 \) for all \( k \).

In light of the pseudo-code in Algorithm 4.1 Pseudo code for the Angiolini et al. algorithm, we can roughly summarize the algorithm as follows: it actually checks for all possible scratchpad sizes (line 5) and all possible ranges from \( i \) to \( j \) (lines 1 and 2) whether it is more profitable to add the range \( i-j \) into one of the previously-created partitions or to create a new partition to hold it putting in mind that the total area cost after adding the new partition does not exceed the whole area allocated for the scratchpad in total(line 6). Accordingly, the most profitable action is taken. Line 8 makes sure that the highest profit found so far is copied to the \( i \)th column for the current \( j \)th row.

Unfortunately, the authors left out—probably for the sake of simplicity—mentioning the details of managing the ranges added to the partitions as a result from line 7; which involves defining and managing additional data structures. Besides, data replication is more likely to occur due to the blind addition of data ranges in \( P[k+w(i, j)][j] \) at lines 6 and 7.
In addition, a core weakness in the algorithm is that the cost function \( w(i,j) \) is used in the array index, even though an earlier inspection of \( w(i,j) \) concluded that it does not typically yield integer values. Unfortunately, \( w(i,j) \) cannot be used for the array index as suggested by the algorithm unless enumerated values are used. This will, certainly, add extra complications to the algorithm because then every time we need to test the condition in line 6 we'll have to first compute the area returned by \( w(i,j) \)--which is a float number--and then translate this value to an integer number. It is obvious that this quantization-like process introduces a needless trade-off between the computation accuracy and the algorithm's efficiency.

Despite the fact that the aforementioned definition of the cost function \( w(i,j) \) was found inapplicable, and cannot be actually used to obtain the published results, we used an alternative cost function to experiment with a 12-word sample benchmark. We assumed the scratchpad size to be 9 words; that is \( N=12 \) and \( C=9 \). The algorithm produced an optimal result in almost 0.22 seconds; which may seem promising at a first glance. But is the result obtained for this small benchmark sufficient enough to conclude the algorithm's applicability to practical-sized applications, not to mention data-intensive? To answer this question, we need to better investigate the algorithm's complexity.

4.2.4 Complexity and Analysis

Taking a superficial look at the pseudo code, it is easy to see that the algorithm has a time and space complexity of \( \theta(N \cdot C^2) \), that is, the polynomial complexity claimed by the authors. Note that the algorithm analyzes the target application’s execution trace, taking into account every possible address range as a candidate partition for the desired optimal scratchpad partitioning.

Even though the algorithm seems very attractive after this thorough analysis since it is guaranteed to always find the optimal solution in polynomial time and polynomial space, it makes a great difference to put in mind two essential facts:

4.2.4.1 Space Complexity:

As previously discussed, there is a need to store information about the profit gained by every partitioning represented by the processing array (P), plus
"some way" of tracing the actual ranges composing the partitioning itself represented by the address ranges array (S).

Every cell in P needs at least 32 bits if the profit is expressed in terms of intercepted accesses—that is optimization for performance—and no less than 64 bits if expressed in terms of energy to avoid unwanted clipping or rounding effects.

Besides, each cell in S is an array of memory ranges, with starting and ending addresses for each range. Therefore, every cell in S is at least 64 bits in a best case of a 1 partition configuration entailing the need to store two addresses (at least of size 32 bits each).

If we suppose that a single matrix of a special structure type to hold all this information altogether, and even by wisely using pointers to best utilize memory resources, it is impossible to take less than 16–20 B of memory per matrix cell of this type. For example, for an application—or a critical section of an application—with a 256kB memory trace and a 16kB scratchpad, the needed matrix data structure will cost about 4 to 5 GB as a whole; viewing it as a single matrix of 64 k columns and 4 k rows, totaling 256 M cells, each cell of at least 16 to 20 Bytes of memory in a best case.

This means that for applications whose memory traces are in the range of hundreds of Kilobytes, the memory requirement is in the order of Gigabytes! Clearly, this memory requirement is unacceptable for medium-sized applications, as processing on a common workstation would prove impossible. The expected conclusion of the algorithm's inapplicability to data-intensive applications, hence, follows.

4.2.4.2 Time Complexity:

The attempt to solve memory misuse with extensive list management and dynamic memory allocation—as suggested by the authors—imposes a heavy load upon processing time. However, the authors claim they were able to make certain optimizations which "significantly cut the overall memory requirements without slowing down execution to an unacceptable level."

One of these optimizations is adjusting the output to fit on power-of-two memory banks; which makes it possible to prune most of the exploration
space, thus achieving dramatic speedups. This feature is especially useful when mapping applications onto scratchpads implemented as memory macros; that normally come in power-of-two sizes.

Implementation-wise, some additional checks on the $w(i, j)$ cost function need to be performed to decide whether an iteration is to be skipped if its range does not match a power of two. By adopting this mapping strategy, complexity drops from $\theta(N.C^2)$ of an un-optimized implementation to $\theta(N.C.\log C)$ because the innermost loop at line 5 in an optimized pseudo code version of Algorithm 4.1 Pseudo code for the Angiolini et al. algorithm gets drastically reduced in scope. As a result, memory requirements and execution times are expected to dramatically diminish.

Yet, it is important to point out the loss of optimality caused by this option because an optimal solution may be carried out in partitions with sizes that are not powers of two. It is also possible to have a more profitable case of merging two partitions together; which becomes impossible since their added power-of-two sizes do not necessarily yield a power-of-two total. Even though the results the authors got by applying this strategy showed a speed and memory improvement of up to two orders of magnitude due to the dramatic pruning of the exploration space, performance profit results were globally worse. The number of intercepted memory accesses turned out to be up to 18% lower. And, the profit results did not make much difference from the energy profit perspective. In general, the conclusion that a full custom-memory subsystem design may be worth the extra development time in some circumstances was reached.

4.2.5 Limitations and Shortcomings

This algorithm suffers several implementation problems. First, the input trace is obtained by simulation; which is computationally expensive in the case of data-intensive applications. Second, the mapping of the signals to physical memory locations is left completely aside. There is no clear methodology on how this step is to be performed before tackling the partitioning problem. A third and more crucial setback of the algorithm is the use of the non-integer cost function $w(i, j)$ as an indexer in the processing matrix. The problem is not only that this cost function is not integer, but also that it is not easy to be
converted to integer since it depends on the energy, area and/or time model. If the model is changed, the values generated by the function will need to be re-analyzed and, very likely, modified.

Efficiency-wise, the memory requirement needed by the algorithm’s main structure is infeasible for data-intensive applications. As a result, processing this structure on a common workstation would prove impossible.

The next chapter introduces a novel approach that exploits the same methodology—dynamic programming—in a different way.
5 A PROPOSED DYNAMIC PROGRAMMING APPROACH

This chapter exhibits a novel dynamic programming algorithm to solve the scratchpad partitioning problem. Generally tuned toward data-intensive applications, the dynamic programming algorithm discussed in this chapter conquers many of the aforementioned shortcomings associated with Angiolini et al.’s algorithm.

The major accomplishments of this algorithm are mainly related to cost evaluation and overall efficiency. Cost evaluation in this approach is built upon memory simulation results of energy and time obtained from CACTI [36] as described in section 2.4. The drastic improvement of time and space complexity in this approach are a result of eliminating the need to compute the area at every step in the previous approach by, instead, translating a given area constraint to its equivalent SPM size (in words), as will be seen shortly. Also, the genuine transformation of the main processing matrix to contain drastically less number of elements cuts down the time needed to process each element.

5.1 Inputs

The algorithm takes as input the number $N$ of addresses in the sorted execution trace, along with the actual address $addr_i$ and the number of accesses $rw_i$ associated with the $i^{th}$ sorted trace entry and the maximum area allocated for the scratchpad in total, denoted $\theta$. It is assumed that $\alpha_i$ is expressed in 32-bit words, while $\theta$ is given in $nm^2$.

It is possible to determine the maximum number of partitions possible by finding the maximum value of $M$ that satisfies the formula:

$$\sum_{i=1}^{M} \left[ SPM\_area\left(\frac{\delta, N}{M}\right) + \Delta A_{i,i+1}\right] \leq \theta$$

(6)

where:

- $M$ is the number of maximum number of partitions; desired to maximize its value.
- $SPM\_area(n)$ is the area cost of a partition of size $n$ bytes. This function obtains its values from Cacti; the memory simulation tool.
- $\delta$ is the word size in bytes, assumed to be 4 bytes per word.
- $N$ is the total number of addresses in the input trace.
$\Delta A$ is the area overhead associated with an extra partition; obtained from Cacti.

$\theta$ is the given area constrained for the whole scratchpad memory module.

After finding the maximum number of partitions ($M$) possible under the given area constraint, the algorithm calculates the minimum partition size possible (denoted $\Phi$ and measured in 32-bit words) under the given area constraint by a simple division:

$$\Phi = \left\lfloor \frac{N}{M} \right\rfloor$$  \hspace{1cm} where $M, N, \Phi \in Z^+$

(7)

In other words, the algorithm translates the given area constraint (in $nm^2$) to a simpler, more quantized constraint to easily apply on the dynamic table calculations. This simpler constraint is the size of the partition that cannot be further partitioned. By doing that, we simplified the search process by pruning all inconvenient partitioning possibilities that yield a partition size smaller than $\Phi$—and that are most likely to violate the given area constraint.

Starting from this point, the given area will not be needed since we now know the smallest possible partition size in words ($\Phi$) beyond which the area constraint is violated. Therefore, the complications created by the area cost function in Angiolini et al.'s algorithm are now avoided.

Even though constraining the partition size to match the maximum area allowable for the scratchpad module may give the impression that the resulting area cost of the generated solution leaves a tiny margin—if at all—before exceeding the given area limit, this does not necessarily imply that optimization for area is not possible. On the contrary, the idea of having the area constraint as a hard limit for partition size gives a chance to best utilize the available area when area optimization is not the target, while still allowing for area optimization, if required, in cost calculation.

### 5.2 The Algorithm

After refining our inputs, our objective now is to select a set of mutually disjoint address ranges (each range forming a partition), the overall cost of which is maximized, and any partition size is not less than $\Phi$. Algorithm 5.1
Pseudo code for the novel dynamic programming algorithm. below shows a pseudo code of the proposed dynamic programming algorithm.

```
1  for j := N to 1 do
2      for i := 1 to N and k := j to N do
3          P[i][k] = rw(i,k) . cost(i,k)
4          if k-i < 2 . Φ then
5              continue
6          for l := i+ Φ to k - Φ do
7              if P[i][l] + P[l+1][k] < P[i][k] then
8                  P[i][k] := P[i][l] + P[l+1][k]
9                  S[i][k] := l
```

Algorithm 5.1 Pseudo code for the novel dynamic programming algorithm.

For each pair \([i, j]\), \(1 \leq i \leq k \leq N\), let \(rw(i, k)\) be the total number of accesses to the memory range starting from location \(i\) and ending at location \(k\), and let \(cost(i, k)\) be the cost associated with a single access to a memory location in the range of addresses from the \(i^{th}\) to the \(k^{th}\). See the next section for more details on cost calculation.

The algorithm's main structures are two \(N \times N\) matrices, \(P\) and \(S\), representing the costs and partitioning positions, respectively, see Figure 5.1. Every cell in \(P\) contains the overall cost according to one of the strategies mentioned above. The cell in row \(i\) and column \(k\) contains the cost of the best partitioning found so far, starting from the \(i^{th}\) address up to the \(k^{th}\) address. Initially, all entries of the matrix \(S\) are set to 0. If that best partitioning was achieved by making a partition rather than having it as a single bank, the matrix \(S\) is updated with data about the partitioning position that charges this cost.

The entries in \(P\) and \(S\) are filled out diagonally in an ascending order of range size represented by the index \(j\) in line 1. At any time of processing, \(j\) is always equal to \(k-i\), which indicates the current range size being explored.
In light of the pseudo-code in Algorithm 5.1 Pseudo code for the novel dynamic programming algorithm, we can roughly summarize the algorithm as follows: it actually checks for all possible partition sizes starting from the minimum partition size, $\Phi$, up to the maximum possible partition size, $N$, (line 1) and all possible ranges from $i$ to $k$ that give this size (line 2) whether storing this range in one partition (line 3) is more profitable than partitioning at any possible partitioning position (lines 6 through 9). Lines 4 and 5 make sure that the partition at hand can be partitioned to smaller banks without violating the constraint before entering the comparison loop.

### 5.3 Cost Calculation

The cost function is represented by the value returned from querying Cacti memory simulator database—see Appendix A – Simulation Data. The query to Cacti’s database could be for energy cost, performance cost and/or area cost per memory access. The cost function referenced in line 3 of the pseudo code deploys these value(s) to set the costs of unpartitioned ranges in the processing matrix $P$ for a memory bank of the size $(k-i)$.

The cost function gives the flexibility to set the values directly as energy, performance or area costs, or a weighted sum of any of them in case of multiple objectives, depending on the user's optimization target.
5.4 Results and Discussion

The results in this section were obtained by running a C++ implementation of the algorithm on a PC with an i5 core at a 2.5 GHz processor. The program was tested using the same benchmark used to test the previous approaches with different values of $\Phi$ and different granularities. The optimal partitioning cut sets found by this algorithm are shown in the table in
Appendix B – Testing Results, along with data captured for each generated solution; like the total energy consumption, the total time cost of the partitioning and the total area needed for the partitioning. The memory simulation data were acquired for the 32 nm\(^2\) technology, scratchpad memory type.

The granularity (denoted \(G\)) specifies the exploration accuracy for each partition possibility. In general, specifying a granularity can be viewed as increasing the word width. Increasing the granularity of the search not only cuts down the memory requirements as mentioned above, but also significantly accelerates the algorithm's execution time.

![Figure 5.2 Energy costs obtained for different values of \(\Phi\) and \(G\). Even though energy cost is expected to increase with the partitions sizes increase; since accessing larger banks of memory consumes more energy than accessing banks of smaller sizes, the figure suggests a negligible difference between the energy results.](image)

Even though this section discussed the results obtained while running for energy optimization since the main objective of our problem is to optimize energy consumption, our cost function can be oriented to optimize for any of the parameters of interest: energy, performance and area.
Appendix B – Testing Results shows the results obtained for running this algorithm for performance optimization. Optimizing for area alone was expected to give a monolithic solution. Tests asserted this expected conclusion because, in practice, memory modules of bigger sizes are denser than smaller modules in terms of area [3], so it's always better for area to use less memory banks.

Figure 5.3 Performance costs obtained for different values of $\Phi$ and $G$. Performance exhibits a similar behavior to energy. Analogous to energy consumption, accessing larger banks of memory takes more time than accessing banks of smaller sizes. We may roughly estimate that optimizing for energy produces somewhat efficient solutions. This can also be noticed especially when compared with the results obtained from performance optimization tests; see page 102.

However, we may want to optimize for area and another parameter at the same time; for example, optimizing for area and energy at the same time. A method to account for multiple competing objectives in the cost function is discussed in [50]. The method can be mainly summarized in setting weights for each optimization parameter (energy, performance and area) according to the relative improvement required for the corresponding objective. In such a case the solution may not always give a monolithic solution—depending on the specified weights— even though optimization for area is accounted for.
5.5 Complexity and Analysis

One of the main accomplishments we claim for this approach over the previous dynamic programming technique is the enhanced space and time complexity. The following two sections justify this claim and explain techniques that can help in further improving the algorithm's complexity.

5.5.1 Space Complexity:

The memory requirements for this algorithm are influenced by the sizes of matrices $P$ and $S$. Even though Algorithm 5.1 shows that $P$ and $S$ densities are always less than 0.5, we are going to consider the memory taken by the structures as a whole, considering even the sparse elements.

Suppose we use a single matrix whose elements are data structures holding profit and address information at the same time, and suppose also that we need 4 bytes to store addressing information and 8 bytes to store the profit. A single element in this matrix, therefore, takes 12 bytes of memory. Consequently, the memory required to process an N-word trace is $12 \times N \times N$ bytes, resulting in a space complexity of $\theta(N^2)$. Comparing this complexity with Angiolini et al.'s algorithm, we notice that our algorithm improves the memory requirement by an order of magnitude.

It is important to mention that the memory actually needed for processing is far less than the amount considered above, since the parts of the matrix actually used in processing highly depend on $\Phi$, the minimum partition size. The sparse parts of the matrix may be eliminated by performing additional vectors—or pointers—manipulation that may affix simple calculations to elements dereferencing.

Another dodge that significantly reduces the memory usage is playing with the word width, or the granularity of the search. This means that instead of allocating an $N \times N$ matrix, we can allocate only $\frac{N}{G} \times \frac{N}{G}$, where $G$ is the selected granularity. While not sensibly improving the overall complexity, this reduction does make a significant outcome in practical measurements of both memory consumption and time of execution, as can be noticed from Figure 5.4 and Figure 5.5.
Experiments showed that the granularity can have any value starting from 1 (which means all partitioning possibilities are tested) up to the minimum partition size ($\Phi$). A granularity value greater than $\Phi$, in addition to its negative effect on the solution's quality, overrides the advantage of $\Phi$ in the first place. Figure 5.4 shows the memory requirements obtained from processing an 8 KB trace (N=8192) tested with different word widths. The memory requirements are the same for different minimum partition sizes since the allocation of the matrices depend on the trace size and the granularity of the search.

![Figure 5.4 Memory requirements for an 8 KB trace (N=8192) tested with different granularities (or word widths). The memory requirements are the same for different minimum partition sizes.](image)

### 5.5.2 Time Complexity:

A look at the pseudo code presented in Algorithm 5.1 reveals a worst case time complexity of $\theta(N^3)$. This efficiency may seem similar to that of Angiolini's algorithm discussed earlier in this chapter with the case of $C = N$. However, a more careful investigation shows that we may comfortably say that the efficiency is better than $O(N^3)$.

Let us start by examining Figure 5.1 of the main processing structure in our dynamic programming approach. For instance, more than half of the processing matrix is not processed--the area below the dotted line--due to the fact that matrix cells where $k-i < \Phi$ are not explored. In other words, a worst case scenario when the area constraint is loose enough to allow partitions as small as 1 word in size, the processing does not exceed the main diagonal ;
which is less than half of the matrix. Of course, the tighter the area constraint, the larger the minimum partition size, the more sparse the matrix $P$ becomes and, consequently, the faster the execution.

Now, observe the pseudo code in Algorithm 5.1 Pseudo code for the novel dynamic programming algorithm. Lines 1, 2 and 6 are guaranteed to loop exactly $N-1$ times in a worst case scenario with $\Phi=1$. Loop $j$ in line 1 loops for $N-\Phi$ iterations. For each iteration of $j$, loop $i$ and $k$ in line 2 is executed $N-j$ times—that is, $N-\Phi$ times at first and keeps decreasing till it gets executed only once when $j$ reaches $N$. Finally, the inner most loop of $l$ at line 6 executes $k-i-2\Phi$ under the condition that $k-i > 2\Phi$. This means that loop $l$ is skipped for unpartitionable sizes and otherwise loops for a maximum of $N-2\Phi$ iterations. This gives an overall efficiency of $\theta((N - \phi)^3)$. As a result and as will be confirmed by the testing results in the coming section, a larger value of $\Phi$—reflecting a tighter area constraint—speeds up the execution.

A single run with granularity of 2 bytes was performed and it took 3 hours to find a solution with almost the same energy consumption as the higher granularity runs. Based on this result, we decided to stop our tests at granularity=8 bytes since it was not reasonable to continue testing with finer granularity.
Other readings to measure the algorithm's efficiency were also collected, like the CPU time taken to find the partitioning as well as the memory space needed to complete the processing. See Figure 5.3 below.

### 5.6 Limitations and Shortcomings

Even though this algorithm has recorded significant improvements over the previously discussed algorithms in terms of solution quality with respect to time and space efficiency, we are still ajar from finding an algorithm that finds a universal optimal solution in reasonable time.

The computation time, to begin with, drastically increases when the SPM size is large relative to the minimum bank size and granularity. However, with the help of the granularity trick, we were able to find "a solution" in reasonable time. There is no guaranty to unconditional optimality unless if granularity and minimum partition size were set to 1; in which case the program's time and space requirements significantly increase. The reason is because of the constraint on partitions sizes. The preliminary step of translating the area constraint into a partition size constraint, despite its relaxing effect on the problem, prevents the algorithm from exploring possibly-optimal solutions with smaller partition sizes and thus giving a chance that optimal solution is missed.

Another drawback of the current implementation of the algorithm is limiting the granularity \( G \) to be a common factor between \( N \) and \( \Phi \). This is because we restrained the size of our processing matrix to be \( \frac{N}{G} \times \frac{N}{G} \). As a result, the algorithm is prevented from exploring cut positions that are not divisible by \( G \). However, while significantly improving the algorithm's performance, testing results suggest negligible energy degradation due to increasing granularity within the given value of \( \Phi \), see Figure 5.2.

Another limitation is the inability to determine the idleness of data stored in the banks, a technique that helps in further saving energy by putting the banks to sleep when they are not accessed to save static energy.

The question that remains open is whether it is possible to devise an algorithm that finds a universally-optimal solution to our partitioning problem without unrealistic demands of time and computing resources? The next chapter...
discusses an interesting algorithm that approximates solutions by means of a backtracking approach.
6 A PROPOSED BACKTRACKING APPROACH

Exhaustive exploration of the partitioning solution space has shown to be computationally expensive for non-data-intensive benchmarks [2],[3],[18]. The computation and memory space expenses proved to be visionary when dealing with data-intensive applications using the exhaustive partitioning algorithms discussed in the previous chapters. In addition, major obstacles—in terms of computational efficiency—have been observed when applying previous exhaustive methods for data-intensive applications.

Therefore, it may be more convenient when partitioning for data-intensive applications to follow an approach that is steered by an analysis of the intensity of read/write accesses within the array space of the multi-dimensional signals rather than observing scalar memory locations individually. In this chapter, we present a refinement of the brute force approach that is based on the code analysis and makes better use of backtracking.

6.1 Significance of Code Analysis

To better understand the potential of using an approach that is steered by an analysis of the intensity of read/write accesses within the array space of the multi-dimensional signals rather than observing scalar memory locations individually, let us consider our illustrative example of a typical affine behavioral specification. Observing the code in Figure 1.7, we characterize 4 nested loops intensively accessing signal A array space. Figure 6.1 below highlights the loop nests and accesses to signal A in the code.
We build on the assumptions made earlier in section 1.3; that is, assuming that the entire array A is stored in an on-chip scratchpad, with each element occupying only 1 byte, the mapping of the array into the scratchpad is column-by-column, and the on-chip memory has one \textit{read/write} port.

Consequently, the dynamic energy consumption $E_{\text{dyn}}$ for a monolithic on-chip scratchpad of 8 Kbytes—needed to store the 512 x 16 elements of A—can be computed multiplying the number of \textit{read} accesses to A by the dynamic energy spent per each access. The number of accesses to the A-elements is 5,187,840; which can be easily computed by summing the access contributions to each part of the array space in Figure 6.2, or directly from the code in Figure 1.7. The dynamic energy per read access can be estimated using CACTI 6.5 [36]-- an analytical tool that takes a set of cache/memory parameters as input and calculates its access time, power, cycle time, and area.

Choosing, for instance, the technology 32 nm and the memory type \textit{“ram”} (that is, scratch RAM, without tag array), and one \textit{R/W} port, CACTI yields 0.01245 nJ dynamic energy per \textit{read} access and, consequently, $E_{\text{dyn}}$=64.59 uJ for the monolithic scratchpad.

The leakage power of the assumed 8-Kbyte scratchpad is-- according to CACTI -- 0.126703 mW. As the access to the fast on-chip memory is typically less than 1 clock cycle, and since our tool finds that the storage requirements for the arrays B and C is of 1 byte each due to the disjoint lifetimes of their elements (therefore, they can be stored in data path registers), the number of clock cycles necessary for the execution of the illustrative code in Figure 6.1 is 7,783,682 (as each assignment in the fourth loop takes 3 clock cycles -- a \textit{read} from the scratchpad, the addition in parallel with the second \textit{read}, and the subtraction). Assuming a frequency of 400 MHz, the execution time of the code results to be 19.46 ms. Then, the static energy consumption of the on-chip scratchpad is: $E_{\text{st}} = 0.126703 \text{ mW} \times 19.46 \text{ ms} = 2.47 \text{ uJ}$.
Figure 6.2 The partitioning of signal A's array space. The number of memory accesses is indicated for each partition – the darker ones being more heavily accessed.

Since the leakage energy in our current tests is very small in comparison to the dynamic energy consumption, it will be ignored for the time being from further discussion. However, with the scaling down of the technology, this term will gain in significance in the overall energy budget, so the next version of partitioning will take it into account.

6.1.1 Case 2-way partitioning

Now, if the on-chip scratchpad is partitioned into two banks, one bank -- of 0.5 Kbytes -- containing the first column A1 of the array A (see Figure 6.2), and the other bank of 7.5 Kbytes-- containing the rest of A's array space, the dynamic energy consumption of the two banks is:

$$E_{\text{dyn}}(0.5; 7.5) = 61,760 \text{ accesses} \times 0.00411748 \text{ nJ} + 5,126,080 \text{ accesses} \times 0.0120162 \text{ nJ} = 61.85 \mu J$$

$$E_{\text{st}}(0.5; 7.5) = (0.0291453+0.103089) \times 19.46 = 2.57 \mu J$$
Therefore, the dynamic energy consumption decreases by 4.25% relative to the monolithic scratchpad architecture, but there is an energy overhead of 0.18 \( uJ \) introduced by the multi-bank architecture. There is also a slight increase in the overall leakage energy consumption; but this term is still negligible. Other cases of two-bank partitioning can decrease the dynamic energy consumption even more significantly: assuming the first bank stores A1-A2, respectively, A1-A2-A3, and exactly half of the array space, similar computations yield the following results (the arguments are bank sizes in Kbytes): 

\[
E_{\text{dyn}}(1.0 ; 7.0) = 185,280 \text{ accesses } \times 0.004578 \text{ nJ} + 5,002,560 \text{ accesses } \times 0.0116018 \text{ nJ} = 58.04 \text{ uJ}
\]

\[
E_{\text{dyn}}(2.0 ; 6.0) = 555,840 \text{ accesses } \times 0.00549564 \text{ nJ} + 4,632,000 \text{ accesses } \times 0.0107731 \text{ nJ} = 52.96 \text{ uJ}
\]

\[
E_{\text{dyn}}(4.0 ; 4.0) = 2,593,920 \text{ accesses } \times 0.00733036 \text{ nJ} = 37.58 \text{ uJ}
\]

The static energy increases to 2.89 \( uJ \).

### 6.1.2 Case 3-way partitioning

If the on-chip scratchpad is partitioned into three banks, we exemplify for the situation when the first bank (2 Kbytes) stores A1-A2-A3, the second bank (4 Kbytes) stores A4-A5-A6, and the third bank (2 Kbytes) stores A7-A8-A9. Then:

\[
E_{\text{dyn}}(2.0 ; 4.0 ; 2.0) = 555,840 \text{ accesses } \times 0.00549564 \text{ nJ} + 4,076,160 \text{ accesses } \times 0.00733036 \text{ nJ} = 35.99 \text{ uJ}
\]

Note that employing different signal-to-memory mapping functions, which do not entail storing the 2-D signal A column-by-column, the three banks could store A1-A2-A3-A4, A5, and A6-A7-A8-A9 respectively. In such a scenario, the resulting dynamic energy consumption becomes even lower:

\[
E_{\text{dyn}}(2.75 ; 2.5 ; 2.75) = 667,584 \text{ accesses } \times 0.00618114 \text{ nJ} + 3,852,672 \text{ accesses } \times 0.00595243 \text{ nJ} = 31.18 \text{ uJ}
\]

However, the energy overhead\(^7\) in this latter case is higher (1.35 vs. \( 0.54 \text{ \mu J} \)) due to a more complex decoding logic. If we split in half the lattices

\(^7\) The decoding circuits were synthesized using the ECP family of FPGA’s from Lattice Semiconductor and, for the energy overhead, we used the Power Calculator from Lattice Diamond[48].
A4-A5-A6, the best energy consumptions (dynamic, static, and overhead) -- restricted to the disjoint lattices partitioning the array space -- obtained for 4, 6, 8 banks, respectively, are:

\[ E(2.0 ; 2.0 ; 2.0 ; 2.0) = 33.10 \text{ } \mu J \]
\[ E(1.0 ; 1.0 ; 2.0 ; 2.0 ; 1.0 ; 1.0) = 33.56 \text{ } \mu J \]
\[ E(0.5 ; 0.5 ; 1.0 ; 2.0 ; 2.0 ; 1.0 ; 0.5 ; 0.5) = 34.85 \text{ } \mu J \]

The reduction of the dynamic energy consumption becomes insignificant (28.51, 27.49, 27.32 \( \mu J \)) with the increase in the number of banks beyond a certain value. Contrary to this “saturation” effect, the static energy consumption continues to increase (to 5.37 \( \mu J \) in the last case), as well as the energy overhead (to 2.16 \( \mu J \)) since the decoding circuitry becomes more complex.

The optimal bank partitioning for different numbers of banks is included in the partitioning of the index space in disjoint lattices, making thus the exhaustive exploration quite unnecessary.

Performing exhaustive explorations for different values of \( M \)-- the maximum number of scratchpad banks. For \( M=4 \), the optimal bank partitioning was obtained for the scratchpad addresses of the bank boundaries: \([0, 2655, 4078, 5533, 8192] \)\(^8\); the corresponding minimum energy consumption was 32.34 \( \mu J \), only slightly better than the energy consumption of 33.10 \( \mu J \) obtained for 4 banks of 2 \( Kbyes \) each (see above).

The result of the exhaustive exploration was obtained after 1-hour computation time, based on the analysis of 40.814 billion banking configurations. When the value of \( M \) was increased beyond 4, the exhaustive exploration became computationally infeasible.

The main conclusion of this case study is that the exhaustive exploration is computationally expensive--possibly infeasible-- and the resulting energy reduction does not justify such a brute-force exploration. This represents a

\[^8\] Memory generators do not allow all possible values for bank boundaries, so the search can be constrained by practical requirements: for instance, a memory generator may yield storage blocks with only a multiple of 16 bytes. More general, a user-specified minimum granularity for the bank sizes could be imposed within the CAD tool.
motivation of a banking algorithm that guides the exploration by the intensity of memory accesses in the array space (see Figure 6.2.)

6.2 A Memory Management Framework

This section presents an electronic design automation (EDA) methodology for the high-level design of hierarchical memory architectures in embedded data-intensive applications, mainly in the area of multidimensional signal processing. The framework presented in this chapter employs a formal model originally established by Balasa et al. [13] operating with integral polyhedra, using techniques specific to the data-dependence analysis employed in modern compilers.

The section briefly addresses the problems of data assignment and mapping before an optimized banking of the on-chip memory can be tackled in a consistent way.

The main design target of this model, as stated by its authors, is the reduction of the static and dynamic energy consumption in the memory subsystem, but the same formal model and algorithmic flow can also be applied to reduce the overall time of access to memories and consider the chip area.

6.2.1 Input

The expected input to the model is data-intensive algorithms for multimedia applications—possibly real-time applications too—that are typically specified in a high-level programming language where the code is organized in sequences of loop nests having as boundaries linear functions of the outer loop iterators, conditional instructions, and multidimensional signals whose array references have, possibly complex, linear indices. This class of specifications is often referred to as affine[52] due to the fact that they contain array references whose indexes are affine functions of the loop operators.

Sometimes, in image, speech, and numerical processing, there may be also indices containing modulo operators using Hermite Normal Forms [40]. Such specifications can be transformed into the affine class applying only integer arithmetic. Such applications are typical in multidimensional (M-D) signal processing, particularly in the multimedia and telecommunication domains. These behavioral specifications describe the processing of streams of data
samples, so their source codes can be imagined as surrounded by an implicit loop having the time as iterator. Consequently, each signal in the algorithm has an implicit extra dimension corresponding to the time axis.

These embedded systems are often designed under stringent energy consumption constraints, to limit heat generation and battery size. The illustrative code with four nested loops shown in Figure 1.7 is an example of such a specification.

The approach presented in this chapter starts from a behavioral specification—rather than from a memory trace as compared to other approaches discussed in subsequent chapters—and takes directly into account energy consumption, performance, and area. Therefore, since the memory optimization targeted by this system starts from a given high-level behavioral specification, it may be classified as a compiler-assisted approach. Different from the prior solutions, and as will be evident by the end of this research, this technique is efficient and accurate for scratchpads of a larger size and/or with a larger number of banks.

### 6.2.2 Output

The system provides a complete specification of the on- and off-chip memory configuration, including signal assignment to different layers (on- and off-chip) as well as the mapping of signals to the physical memory locations and a near-optimal banking—or partitioning—of the on-chip scratchpad memory. Even though the memory configurations provided by this model are typically not guaranteed to be energy-optimal, but the authors claim near-optimal estimations.

### 6.2.3 The Design Flow

In brief, the flow of the proposed memory management methodology before scratchpad partitioning is shown in Figure 6.3.
The whole process that finds an optimized memory configuration can be summarized in three main steps:

6.2.3.1 Step 1: Decomposition of the array space of every indexed signal into disjoint bounded lattices:

The first step to analyze the input code is to construct a polyhedral library containing the images of the affine vector functions – called lattices:

\[ x \rightarrow T \cdot i + u \] over integral polytopes [39]. In other words, every lattice in the constructed polyhedral library is a bounded polyhedron restricted to points having integer coordinates – since the loop iterators have integer values. This decomposition into lattices can be performed analytically, by recursively intersecting all the array references of each signal [11][16].

This division of the signals is based on the frequency of access to each lattice, as mentioned earlier. We compute the number of memory accesses for each of the disjoint lattices by operating with the polyhedra and lattices extracted from the application code: we identify every array reference containing a selected lattice, and then we determine the expressions of the loop iterators addressing the lattice as part of each array reference. The sizes of these sets are the amounts of accesses to the given lattice, as part of each array reference.
containing it. The summation of all the individual contributions yields the total number of accesses to the array elements in the lattice.

Figure 6.4(a) shows a sample code that accesses the array X in the scope of nested loops and conditional instruction. The reference to the array, $X[3*i+2*j][4*i+3*j][k]$, have as iterator space P derived from the loop boundaries and logical expressions of the conditions.

After the elimination of redundant inequalities; that is, the points of integer coordinates inside the pyramid located in the center of Figure 6.4(b), the indices $x$ and $y$ of the $X$-elements of the array reference $X[3*i+2*j][4*i+3*j][k]$ can be modeled as the linearly bounded lattice:

$$LBL_X = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 3 & 2 & 0 \\ 4 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix}$$

where $x$, $y$ and $z$ are the three indexes of the array reference and $T$ is the $3 \times 3$ matrix.

**Figure 6.4** Part (a) shows the code excerpt containing accesses to signal X. Part (b) shows the mapping from the iterator space into the index space for the array reference bounded by the conditions in the if statement.
It can be shown that the points of the index space are included in the integral polytope:

\[
\{ \ 2 \geq 3x - 2y \geq 0, \ 3 \geq -4x + 3y \geq 0, \ 4 \geq z \geq 0, \ 4 \geq 6x - 4y + z, \ 12 \geq -16x + 12y + 3z \ \}
\]

\( LBL_X \) is the quadrilateral from the left-hand side of Figure 6.4(b). Not all the integral points of the quadrilateral are in the set of points \( LBL_X \); but rather only the points shown as darkened dots in the figure have coordinates that are indices of the array reference.

Calculating the number of memory accesses to the array reference \( X[3*i+2*j][4*i+3*j][k] \) in the example shown above, we find it is equal to 24 accesses; represented as darkened dots in the iterator space shown in the left-hand side of Figure 6.4(b) and are bounded by the pyramid \( aABCD \) from the right-hand side of Figure 6.4(b).

Collecting the number of memory accesses to all the lattices of an M-D signal, we build a map of the memory accesses to the array space of the signal. This computed map is an approximation of the exact map that can be generated by a simulated execution since the access distribution within each lattice is considered uniform; that is, equal to the average value of accesses to every individual array element in this lattice. Figure 6.5 shows an example of exact and computed access maps of a 2-D signal of size 350x350 elements from a code excerpt omitted for simplicity.
Figure 6.5 An example of an exact access map of a 2-D signal of size 350x350 elements is shown in part (a). Part (b) shows the computed 3-D map of memory accesses to the same signal, based on the decomposition of its index space into disjoint bounded lattices. The horizontal plane (XY) represents the array dimensions while the vertical Z axis is the access density.

Computing such approximate maps of accesses has an important advantage: the usually time-expensive simulation is not needed any more, being replaced by algebraic computations. Moreover, we can control the granularity of the array space decomposition into disjoint lattices: a finer granularity entails a computed map of accesses gradually closer to the exact map.

In embedded systems, an address generation unit (AGU) is typically implemented to compute arithmetic expressions in order to generate sequences of addresses [42]. Sets of array elements organized as lattices are a good input for the design of the AGU.

6.2.3.2 Step 2- Signal assignment to the memory layers:

This step aims for selecting the lattices having the highest access numbers, whose total size does not exceed the maximum scratchpad size--assumed to be a design constraint-- and assign them to the on-chip layer. The other signals will be assigned to the off-chip memory layer. Afterwards, the signals are mapped to the physical memory blocks in each layer [19], [40]. This design step determines mapping functions for all the signals in the specification and,
also, generates the distribution of memory accesses graph, like the one in Figure 6.6.

Quite obviously, the most desirable scenario – in point of view of both energy consumption and performance – is that all the signals be stored into the scratchpad memory layer. This is usually not possible because, most frequently, the size of this on-chip memory is small relative to the storage requirement of the entire code. In data-intensive applications, like the example at hand, the main data structures in the behavioral specification are multidimensional arrays. The problem is to automatically identify those parts of arrays that are more intensely accessed, in order to their assignment to the energy-efficient data storage layer—that is, the on-chip scratchpad-- such that the dynamic energy consumption in the hierarchical memory subsystem be reduced.

The number of storage accesses for each array element can certainly be computed by simulating the execution of the code. For each pair of possible indexes of the signals' elements, the number of memory accesses is counted and recorded. The drawbacks of such an approach are twofold. First, the simulated execution may be computationally expensive when the number of array elements is very large, or when the application code contains deep loop nests. Second, even if the simulated execution is feasible, such a scalar-oriented technique would not be helpful since the addressing hardware of the data memories would be difficult to design.

Instead, let us now make use of step 1, assuming that the array space of each multidimensional signal is partitioned into disjoint lattices. This is exactly like transforming the code of the application such that neither two array references have any array element in common. This allows computing exactly the number of memory accesses to the different parts of the arrays.

The technique is based on a simple observation: the most intensely-accessed parts of the array space of a multidimensional signal are typically covered by more than one array reference. Actually, in many cases, the more array references cover a certain element, the more accessed that element is.
An assignment algorithm mapping signals to the memory layers such that the dynamic energy consumption, performance and chip area of the hierarchical memory subsystem be optimized is presented in [28]. The technique in [13] estimates the dynamic energy per read/write access, as well as the access times using CACTI 6.5 [36]— an analytical tool that takes a set of scratchpad, DRAM, or cache parameters as inputs and calculates memory access time, power, and area. More details about using CACTI 6.5 simulator will be discussed in the next chapter.

### 6.2.3.3 **Step 3- Mapping Signals to Physical Memory Blocks in Each Layer**

This design step determines mapping functions for all the signals in the specification and, also, generates a graph of distribution of memory accesses in the scratchpad. According to [13], this design phase has the following goals: (a) to map the signals—that are already assigned to the memory layers—into amounts of data storage as small as possible, both for the scratchpad and the external memory; (b) to compute these amounts of storage after mapping on both memory layers—that is, to find an allocation solution—and be able to determine the memory location of each array element from the specification—that is, to find an assignment solution; (c) to use mapping functions simple enough in order to ensure an address generation hardware of a reasonable complexity; (d) to ascertain that scalar signals—that is, decomposed array elements—simultaneously alive are mapped to distinct storage locations.

The mapping of signals onto the on-chip scratchpad and off-chip memory—see Figure 1.6—is performed such that the overall amount of data storage be low and the address generation functions be simple. Different from other techniques[43]-[46], the mapping technique in [13] is designed to work in hierarchical memory organizations operating with parts of the arrays, represented by disjoint lattices, that can be assigned to different physical memories. The polyhedral framework, common to all the design phases in the system (data assignment to the memory layers, signal/array mapping onto the external memory and the scratchpad, followed by the banking of the latter), entails a high computation efficiency since all the phases rely on similar polyhedral operations.
6.2.3.4 Step 4- On-chip partitioning:

After assigning the signals from the application code to the memory layers, and after mapping these signals onto the physical memories on- and off-chip, further optimization of energy consumption and performance can be obtained by partitioning the on-chip memory.

We have now completed our quick overlook on the phases preceding the partitioning phase in the proposed memory management system. The next section shows a novel partitioning algorithm that makes use of the framework in addressing the problem.

6.3 A Backtracking-Based Partitioning Algorithm

This section shows a recent research [14][41] that addresses the problem of optimal memory partitioning of the on-chip scratchpad memory for data-intensive embedded applications.

The algorithm presented in this section relies on a recursive technique with backtracking. Backtracking is basically a form of recursion. In general, the usual scenario in backtracking is having a number of options at every step and having to choose one of them. After making a choice, we are left with a new set of options; just what set of options we get depends on what choice we made. This procedure is repeated over and over until we reach a final state [73].

This programming technique can be used for optimization problems, solving puzzles (like jigsaw and other) and creating computer opponents in thinking games like chess [72]. The main gist of backtracking is may be explained as walking through “a tree of possibilities” with branches and sub-branches. A branch in our case means splitting an SPM bank into two at a certain position. When we reach a branch that harms or does not achieve further optimization in our cost function, we go back along the same track as we came, and try another branch; that is splitting the bank at a different position. In general, sometimes the complete tree is searched and backtracking does not occur in some backtracking algorithms.

There are different types, or styles for achieving backtracking algorithms, like recursion, looping or using a recursive language like Prolog, for instance [72].
In our proposed algorithm, we use the first style. We developed a recursive function called *Partitioning* that will be discussed in detail.

The algorithm presented in this section relies on a recursive technique with backtracking. Unlike all the previous techniques, the cost function takes into account all the three major design objectives – the energy consumption, performance, and die area.

In contrast to Benini’s exploration that yields optimal solutions but whose performance degrades exponentially, the banking algorithm presented in this section yields near-optimal solutions in significantly less time due to an exploration mechanism based on the intensity of memory accesses in the array space of the multidimensional signals from the application code—refer to chapter 6.2 for more details about the framework integrated with this algorithm.

### 6.3.1 Inputs

This algorithm is integrated in the design flow presented in chapter 6.2. It assumes that the preliminary steps mentioned in 6.2.3 are performed prior to running the banking algorithm. Consequently, the output generated from these steps are assumed to be input to the algorithm at hand. The inputs of the partitioning algorithm are:

- **Input 1:** The maximum number of scratchpad banks, denoted $M$.
- **Input 2:** Three arrays $E$, $T$ and $A$ defined as follows: $E = [\Delta E_{1,2}, \Delta E_{2,3}, \ldots, \Delta E_{M-1,M}]$ whose elements $\Delta E_{k,k+1}$ are the energy overheads resulting from moving from a scratchpad with $k$ banks to one with $k+1$ banks; $T = [\Delta T_{1,2}, \Delta T_{2,3}, \ldots, \Delta T_{M-1,M}]$ and $A = [\Delta A_{1,2}, \Delta A_{2,3}, \ldots, \Delta A_{M-1,M}]$ whose elements are time and area overheads, respectively, resulting from moving from a scratchpad with $k$ banks to one with $k+1$ banks.

The elements of these arrays were estimated synthesizing decoding circuits, using the ECP family of FPGA’s from Lattice Semiconductor. The Power Calculator from Lattice Diamond[48] was used to estimate the energy overheads.

- **Input 3:** An array $A = \{addr_0, addr_1, \ldots, addr_n\}$ of ordered addresses such that a linearly bounded lattice $L_k$ partitioning the index space be mapped at the
scratchpad addresses \{addr_k, \ldots, addr_n\}. It is assumed that the storage requirement after mapping each lattice is computed at Step 2 in the preliminary steps mentioned in 6.2.3.

- **Input 4:** An array \( RW = [rw_0, \ldots, rw_n] \) whose elements represent the numbers of read/write accesses for each lattice mapped onto the scratchpad. For the illustrative example, they are the numbers inside the lattices in Figure 6.2, computed as explained in Step 1 in the preliminary steps mentioned in 6.2.3.

The algorithm then produces as output an array of scratchpad addresses delimiting the banks, and the values of energy, total access time, and area for the minimum relative cost.

### 6.3.2 The Algorithm

Algorithm 6.1 shows the pseudo-code of the banking algorithm we propose. The algorithm starts from the monolithic architecture and searches for the optimal partitioning of the scratchpad memory in no more than \( M \) memory banks, such that the borderlines between banks are addresses in the given array \( A \) described above. This ensures that any lattice of signals is entirely stored in one bank.

Lines 1 through 4 ensure convenient initialization of the variables; initially, the scratchpad being monolithic, \( \text{crtBestPartitioning} \) is set to \{addr0, addrn\}. The variables \( \text{MinEnergy} \), \( \text{MinTime} \) and \( \text{MinArea} \) register the total energy, time and area, respectively, consumed by the corresponding set of scratchpad addresses from \( \text{crtBestPartitioning} \).

Line 5 iterates through all possible positions of partitioning, that is, the addresses enlisted in the input array \( A \); where \( n \) is the number of addresses in \( A \). The function \( \text{SPM_energy(start_addr, end_addr, RW)} \) called in line 6 uses CACTI[36] and the array \( RW \) to compute both the static and dynamic energy consumed in a bank bordered by the argument addresses. Similarly, \( \text{SPM_time(start_addr, end_addr, RW)} \) in line 7 computes the total access time in a scratchpad bank starting and ending at the argument addresses, and \( \text{SPM_area(start_addr, end_addr)} \) called in line 8 represents the bank area containing the address range delimited by the argument addresses.
If the current cut introduces more cost than the minimum cost found so far, it means that further partitioning of the scratchpad will only be more costly, and thus the search is terminated and the solution is displayed. The three variables: $w_1$, $w_2$ and $w_3$ used in cost calculation in line 9 represent relative importance of optimizing the three objectives: energy, time and area, respectively. More details about cost calculation is discussed in the next section.

```
1 minEnergy := SPM_energy(addr_0, addr_n, RW)
2 minTime := SPM_time(addr_0, addr_n, RW)
3 minArea := SPM_area(addr_0, addr_n)
4 crtBestPartitioning := {addr_0, addr_n}
5 for i := 1 to n do
6   E := SPM_energy(addr_0, addr_i, RW)
7   T := SPM_time(addr_0, addr_i, RW)
8   A := SPM_area(addr_0, addr_i)
9   if $w_1*E/minEnergy+w_2*T/minTime+w_3*A/minArea \geq 1$ then
10      break
11     Partitioning(2, M, i, E+\Delta E_{1,2}, T+\Delta T_{1,2}, A+\Delta A_{1,2})
12     pop(SolutionStack);
13     print crtBestPartitioning, minEnergy, minTime, minArea
```

Algorithm 6.1 Pseudo code of the proposed banking algorithm [41].

If otherwise, further partitioning of the scratchpad does not tend to harm the cost, a recursive function called Partitioning is called to continue the exploration of the possible partitioning to be made. The definition of the function Partitioning is shown in Algorithm 6.2 The recursive bank partitioning function used in the proposed algorithm [41].

The function takes as a formal parameter the current number of banks reached so far, searches for the optimal solution such that the first bank ends at addr$_0$. EnergyConsumed accumulates the amount of energy consumed from the start of the scratchpad till the borderline addr$_i$. Similarly, TimeConsumed accumulates the total access time and AreaConsumed accumulates the chip area. These three parameters are used to compute the relative costs of partitioning solutions. A variable crtBestPartitioning, in line 12, records the set of addresses in $A$ corresponding to the most energetically-efficient partition reached in any moment of the exploration.
function Partitioning(m, M, i, EnergyConsumed, TimeConsumed, AreaConsumed)

EnergyConsumed += ΔE_{m-1,m}
TimeConsumed += ΔT_{m-1,m}
AreaConsumed += ΔA_{m-1,m}

E := EnergyConsumed + SPM_energy(addr_i, addr_n, RW)
T := TimeConsumed + SPM_time(addr_i, addr_n, RW)
A := AreaConsumed + SPM_area(addr_i, addr_n)

if w1*E/minEnergy+w2*T/minTime+w3*A/minArea < 1 then
    minEnergy := E
    minTime := T
    minArea := A
    crtBestPartitioning := top(SolutionStack)U{addr_i}

if m < M then
    push(SolutionStack, top(SolutionStack)U{addr_i})
    for k := i+1 to n do
        E := EnergyConsumed + SPM_energy(addr_i, addr_n, RW)
        T := TimeConsumed + SPM_time(addr_i, addr_n, RW)
        A := AreaConsumed + SPM_area(addr_i, addr_n)
        if w1*E/minEnergy+w2*T/minTime+w3*A/minArea ≥ 1 then
            break
    Partitioning(m+1, M, k, E + ΔE_{m+1}, T + ΔT_{m+1}, A + ΔA_{m+1})
    pop(SolutionStack)

Algorithm 6.2 The recursive bank partitioning function used in the proposed algorithm[41].

Again, the backtrack mechanism is incorporated (lines 19 and 20) to prevent the search from being directed towards more expensive partitioning solutions by backtracking as early as possible when the relative cost is increased beyond the found minimum. Furthermore, the algorithm uses a solution stack to keep record of the exploration of the solution space. By the end of the recursion, the best partitioning solution is saved at the top of the stack. Consequently, in line 22, the best solution found by the algorithm is popped from the top of the stack.
6.3.3 Cost Calculation

To best understand the cost calculation used in this algorithm, let us take a simple example of a range of \( N \) contiguous addresses mapped to the on-chip scratchpad memory: \( \{0, 1, \ldots, N-1\} \). Assume that memory is word-addressable and the word width is known and is normally imposed by the chosen core processor. We will be using the convention used by the authors to denote the dynamic energy consumed by mapping the address range of size \( j \) starting from address \( i \) in \( x \) partitions by \( E_x^{\text{dyn}}(i,j) \).

Therefore, the dynamic energy \( E_1^{\text{dyn}}(0,N) \) consumed by a monolithic scratchpad is calculated by:

\[
E_1^{\text{dyn}}(0,N) = E_R(N) \sum_{i=0}^{N-1} \text{read}[i] + E_W(N) \sum_{i=0}^{N-1} \text{write}[i]
\]

where \( E_R(N) \) and \( E_W(N) \) are the energies consumed per read and write accesses, respectively, to a scratchpad of \( N \) words—these metrics are technology-dependent. In addition, \( \text{read}[i] \) and \( \text{write}[i] \) represent the number of accesses to word \( i \) and, consequently, the sums represent the total numbers of read/write accesses to the on-chip memory locations \( 0, 1, \ldots, N-1 \).

If the address space of the on-chip scratchpad is arbitrarily partitioned in two ranges \( \{0, 1, \ldots, k-1\} \) and \( \{k, k+1, \ldots, N-1\} \), then the dynamic energy consumed in a two-bank scratchpad is:

\[
E_2^{\text{dyn}}(0,k,N) = E_1^{\text{dyn}}(0,k) + E_1^{\text{dyn}}(k,N-k)
\]

The first two arguments of \( E_2^{\text{dyn}} \) are the start addresses in words of the two banks, the third being the total size.

With the scaling of the technology below 100 nm, the static energy due to leakage currents has become increasingly important. While leakage is a problem for any transistor, it is even more critical for memories: their high density of integration translates into a higher power density that increases temperature, which in turn increases leakage currents significantly. The static energy consumed in the two-bank scratchpad, having the address space partitioned as above, is the sum of the static energy in each bank:

\[
E_2^{\text{st}}(0,k,N) = E_1^{\text{st}}(0,k) + E_1^{\text{st}}(k,N-k)
\]
Neither term depends on the number of memory accesses. The partitioning is energetically beneficial if \( E_2^{\text{dyn}}(0, k, N) + E_2^{\text{st}}(0, k, N) + \Delta E_{12} < E_1^{\text{dyn}}(0, N) + E_1^{\text{st}}(0, N) \), where \( \Delta E_{12} \) is the energy overhead required by the extra logic and interconnections necessary to move from the monolithic scratchpad architecture to a two-bank architecture.

Similarly, the total access time of a monolithic scratchpad can be expressed, akin to the dynamic energy, as:

\[
T_1(0, N) = T_R(N) \sum_{i=0}^{N-1} \text{read}[i] + T_W(N) \sum_{i=0}^{N-1} \text{write}[i]
\]

where \( T_R(N) \) and \( T_W(N) \) are the times of read and write accesses, respectively, to a scratchpad of \( N \) words. These are technology dependent metrics, too. Denoting \( T_2(0, k, N) \) as the total access time of a two-bank scratchpad, whose start addresses of banks are \( 0 \) and \( k \), the partitioning is beneficial in performance point of view if \( T_2(0, k, N) + \Delta T_{12} < T_1(0, N) \), where \( \Delta T_{12} \) is the time overhead required by the additional logic of the two-bank scratchpad architecture.

The die area of a two-bank scratchpad, having the address space partitioned as above, is the sum of the areas of each bank:

\[
A_2(0, k, N) = A_1(0, k) + A_1(k, N - k)
\]

The partitioning is beneficial in area point of view if \( A_2(0, k, N) + \Delta A_{12} < A_1(0, N) \), where \( \Delta A_{12} \) is the area overhead required by the extra logic and interconnections of the two-bank scratchpad architecture. Calculating the energy, time and area overheads for every extra partitioning differentiates this approach from previously discussed approaches.

To better understand the importance of accounting for the overheads, let us refer to Figure 1.4. The figure shows the more complex architecture of a multi-bank scratchpad versus the monolithic architecture. The additional components and interconnects induced by the address and data buses, the decoder, the control signals may introduce a non-negligible overhead on power consumption, time delay and chip area that must be compensated by the savings entailed by bank partitioning. These savings can be made, for
example, by the average power and time decrease in accessing the memory hierarchy, because a large fraction of accesses is typically concentrated on a smaller, more energy-efficient bank. In addition, the memory banks that stay idle long enough can be disabled through their chip-select (CS) pins.

Due to the different metrics of the three design objectives: energy consumption, performance, and die area, a relative cost may be used to introduce an order relation over the set of two-way partitions. Selecting three weights \( w_1, w_2 \) and \( w_3 \) between 0 and 1, such that \( w_1 + w_2 + w_3 = 1 \) for deciding the relative importance of the three objectives, a two-way partition characterized by the triplet energy-performance-area \((E_1, T_1, A_1)\) is better than another two-way partition characterized by \((E_2, T_2, A_2)\) if \( w_1 \frac{E_1}{E_2} + w_2 \frac{T_1}{T_2} + w_3 \frac{A_1}{A_2} < 1 \).

### 6.3.4 Results and Discussion

An EDA framework for memory management has been implemented in C++, incorporating the formal model for data assignment to memory layers specified by the memory management system required for this partitioning approach and discussed earlier in chapter 6.2. The main input of the developed software tool is an algorithmic specification of the signal processing application, expressed in a subset of the C language.

First, we will use our benchmark shown in Figure 6.1 (as it shows a typical affine behavioral specification; the class of applications targeted by this research) to demonstrate the application of the memory management system. The first three steps of the system—constructing the polyhedral library, assigning the lattices to memory layers and mapping the signals to physical memory blocks—are only abstractly demonstrated since their details are beyond the scope of our research problem.

Assuming that the entire array \( A \) is stored in an on-chip scratchpad, with each element occupying only 1 byte. We assume further that the given constraint for the scratchpad size is exactly equal to the size of signal \( A \), that is 8K bytes, since array \( A \) has 16x512 1-byte elements. The mapping of the array into the scratchpad memory is assumed to be column-by-column and the on-chip scratchpad memory is assumed to have one read/write port.
1. Constructing the Polyhedral Library of the Signals

The first step is to analyze the input code is to construct a polyhedral library containing all the lattices composing the signals used in the code. Constructing such a library involves partitioning of the array space of signals – which can be performed analytically by intersecting recursively the lattices of the signal’s array references. It allows to compute exactly the number of memory accesses when addressing the different parts of the arrays.

This process is carried out on all the signals in the code, resulting in a computed map of memory accesses for each signal. Figure 6.2 displays such a computed map of memory accesses for the signal $A$, where the lattices are labeled and the numbers of memory accesses to each lattice are shown below the corresponding label.

For simplicity, let us consider only the decomposition of lattice $A_5$ of signal $A$’s array space, shown in Figure 6.2. In order to compute the number of read accesses to the lattice $A_5$, the algorithm finds all the inclusions into $A$’s array references, determines the subsets of iterators for which the elements of the partition are addressed, and computes the sizes of these subsets.

$A_5$ coincides with the array reference $A[4+i\%8][j]$, therefore the corresponding subset of iterators is $\{ 2 \geq n \geq 0, 415 \geq j \geq 96, n + 13 \geq i \geq n, j + 96 \geq k \geq j - 96 \}$. Also, $A_5$ is strictly included in the array reference $A[i][k]$, the corresponding subset of iterators being $\{ 2 \geq n \geq 0, 415 \geq j \geq 96, 11 \geq i \geq 4, j + 96 \geq k \geq j - 96, 415 \geq k \geq 96 \}$.

The total accesses to the two integral polytopes composing $A_5$ – 2,593,920 and 1,258,752 – are the numbers of read accesses to $A_5$ as part of the two array references; that is, a total of 3,852,672 accesses.

2. Assigning Signals to Memory Layers

Of course, storing on-chip all the signals is clearly the most desirable scenario in point of view of energy consumption--and sometimes performance too. This is usually not possible since the SPM size is upper-bounded.

In our research, we study algorithms that make different assumptions concerning the size of the scratchpad memory. As will be seen in subsequent
chapters, some assume it to be a given constraint—either directly as an SPM size constraint or indirectly by calculating it from another constraint given for the area available for the on-chip memory module. Other approaches exhaustively study the energy cost incurred with all possible scratchpad sizes up to a given size. And, finally, we will come across a technique that finds out an optimized scratchpad size by computing the ratio between the expected dynamic energy reduction and the scratchpad size after mapping; the value of the SPM size maximizing this ratio is selected, the goal being to obtain the maximum benefit in energy point of view for the smallest scratchpad size.

Anyway, we notice that the array is not uniformly accessed during the code execution. Figure 6.6 displays the intensity of accesses of the elements of signal $A$. For each possible address in $A$'s address space – between 0 and 8K, relative – of the $A$’s elements the number of memory accesses was recorded on the vertical axis. One can see the elements near the center of the array space are accessed with high intensity, whereas the elements at the periphery of the array space are accessed with a significantly lower intensity. This can also be noticed from Figure 6.2.

![Figure 6.6](image)

**Table 6.1**

<table>
<thead>
<tr>
<th>SPM Address Space</th>
<th>SPM Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>3000</td>
<td>3000</td>
</tr>
<tr>
<td>4000</td>
<td>4000</td>
</tr>
<tr>
<td>5000</td>
<td>5000</td>
</tr>
<tr>
<td>6000</td>
<td>6000</td>
</tr>
<tr>
<td>7000</td>
<td>7000</td>
</tr>
<tr>
<td>8000</td>
<td>8000</td>
</tr>
<tr>
<td>9000</td>
<td>9000</td>
</tr>
</tbody>
</table>

Figure 6.6 The distribution of memory accesses to the scratchpad address space assuming the 2-D array $A$ is assigned to the on-chip scratchpad column-by-column.
We may now make use of the map of memory accesses found in the previous step for all the signals by selecting the lattices having the highest access numbers, whose total size does not exceed the maximum scratchpad size; which is usually a design constraint.

In our example, we assume for simplicity that signal A's lattices were found to be the most accessed lattices and are therefore assigned to the on-chip scratchpad. The other signals (signals B and C) assumed to have less-accessed lattices will be assigned to the off-chip DRAM. As a consequence of these assumptions, the entire array A will be assigned to the on-chip scratchpad layer.

3. Mapping Signals to Physical Memory Locations
Since the actual mapping of the assigned signals to the physical memory locations has little effect on our main topic of on-chip memory partitioning, we assume, for the sake of simplicity, that this step is taken care of and that appropriate mapping of the on-chip and off-chip signals is found.

4. Partitioning of Signals Assigned to the On-Chip Layer (Signal A)
After being assigned to the off- and on-chip memory layers, the linearly bounded lattices are mapped to the external DRAM and scratchpad; so, the distribution of the memory accesses to the scratchpad address space is known.

The first group of tests were run on the same benchmark discussed in section 1.3 for the sake of comparative evaluation. Our algorithm[41] was tested against Benini et al.'s[2] algorithm with a word width of 1 byte; the latter performing an exhaustive exploration of the scratchpad address space. While Benini's algorithm found the energetically-optimal solution of 34.58 μJ for $M = 3$ in 2.48 seconds, after analyzing 18.18 million banking configurations, a 4-bank solution of 32.34 μJ was found for $M = 4$, after an exploration of over 40 billion configurations in 1 hour, still larger than 30.76 μJ; the energy of a 6-bank solution found by Balasa's algorithm for M=6. In addition, Benini's algorithm suggested that the full exploration—that is a word width of 1 byte--proves to be computationally infeasible for $M > 4$.

Likewise, by setting the word width to be 16 bytes –thus reducing the scratchpad size in words from 8192 to 512— we find that Benini's recursive
approach finds the energetically-optimal partition of 30.7577 μJ and $M = 6$. It does so after analyzing over 87 billion partitions in almost 2 hours of computational time. However, the backtracking algorithm presented in this chapter was able to give a rival solution that consumes 30.76 μJ in only 0.14 seconds; that is, several orders of magnitude faster! This solution was reached after analyzing 1,343,663 different partitions, which means that over 86 billion possible partitions were pruned from the search and, therefore, drastically cutting down the time of computation needed to find a solution. This test was run on the same benchmark with the maximum number of partitions set to $M = 8$ and the array space of signal A partitioned in 30 disjoint lattices; the lattices A4, A5, and A6 from Figure 6.2 being decomposed by vertical cuts into 8 smaller lattices each. The algorithm was configured for energy minimization by setting $w1=1$, $w2=0$ and $w3=0$.

Another bunch of tests were run on different benchmarks, assuming a 32 nm technology. Table 6.2 summarizes the testing results of these experiments. The experiments were carried out on a PC with an Intel Core 2 Quad 2.83 GHz processor.

<table>
<thead>
<tr>
<th>Application</th>
<th>Address space</th>
<th>CPU/M (M=6) [sec]</th>
<th>Energy savings vs. ([T] (M=8))</th>
<th>Energy savings</th>
<th>Access time savings</th>
<th>Area savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian blur filter</td>
<td>14.3816</td>
<td>7.305 48</td>
<td>54.19</td>
<td>7.36 %</td>
<td>53.32 %</td>
<td>51.35 %</td>
</tr>
<tr>
<td>Motion detection</td>
<td>9.600</td>
<td>5.289 28</td>
<td>32.25</td>
<td>9.69 %</td>
<td>55.78 %</td>
<td>33.37 %</td>
</tr>
<tr>
<td>Motion estimation</td>
<td>1.024</td>
<td>7.36 52</td>
<td>5.32</td>
<td>7.51 %</td>
<td>47.21 %</td>
<td>45.02 %</td>
</tr>
<tr>
<td>Durbin’s algorithm</td>
<td>512</td>
<td>247.05</td>
<td>2.30</td>
<td>7.28 %</td>
<td>46.52 %</td>
<td>43.92 %</td>
</tr>
<tr>
<td>SVD updating algorithm</td>
<td>16.384</td>
<td>7.524 13</td>
<td>65.23</td>
<td>10.94 %</td>
<td>59.88 %</td>
<td>57.81 %</td>
</tr>
<tr>
<td>Voice coding kernel</td>
<td>2.048</td>
<td>1.371 56</td>
<td>10.08</td>
<td>10.21 %</td>
<td>57.48 %</td>
<td>55.08 %</td>
</tr>
</tbody>
</table>

Table 6.2 Testing results for running the backtracking algorithm presented in Algorithm 6.1 on different benchmarks [50]. Columns 3 to 6 show testing results obtained while running the algorithm with energy optimization as the only target, i.e. $w1=1$. Columns 3 and 5 display the results of running Benini’s algorithm. The results in columns 7, 8 and 9 show the energy, access time and area savings achieved by running the algorithm on $M=8$ with $w1=w2=0.4$ and $w3=0.2$ assuming a 32 nm technology.

Most of the benchmarks in column 1 are chosen real life digital signal processing applications developed by iMAC. Error! Reference source not found. gives brief information on the selected benchmarks.

<table>
<thead>
<tr>
<th>Application</th>
<th>Details</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Blur Filter</td>
<td>A medical image processing application which extracts contours from tomograms in order to detect brain tumors.</td>
<td>iMAC.</td>
</tr>
<tr>
<td>Motion Detection</td>
<td>Used in the transmission of real-time video signals on data networks.</td>
<td>iMAC.</td>
</tr>
<tr>
<td>Motion Estimation</td>
<td>The kernel of an MPEG4 motion estimation algorithm for moving objects.</td>
<td>E.Chan and S.Panchanathan [70].</td>
</tr>
<tr>
<td>Durbin’s Algorithm</td>
<td></td>
<td>iMAC</td>
</tr>
<tr>
<td>SVD Updating</td>
<td>Finds efficient solutions for several approximation problems in image</td>
<td>M.Moonen, P.V. Dooren, and</td>
</tr>
</tbody>
</table>
For $M < 4$ and $w1=1$ (that is aiming to optimize only the energy consumption), this algorithm is basically identical to the recursive exploration algorithm presented by Benini et al. [2][18]. For $M \geq 4$ and with energy optimization as the only target, the additional exploration constraint that no disjoint lattice assigned to the scratchpad can cross a bank boundary ensures the effectiveness of the approach, as shown in column 4 in Table 6.2 above that it took relatively lower CPU time to find a solution for $M=8$.

As shown in Table 6.2, column 2 displays the size of the scratchpad address space, that is, the size of the on-chip scratchpad memory. Column 3 reports the computation times for a full exploration—word width=1 byte—implemented as presented in Benini's algorithm and targeting energy reduction, but using simulation values obtained from CACTI 6.5[36] for power estimation. The maximum number of banks was set to $M = 4$.

Column 4 reports the computation times in seconds for our banking algorithm with $M = 8$. For each benchmark, the same signal-to-memory mapping was used in order to have identical traces of memory accesses for both techniques. Not only the computation times of the backtracking algorithm were significantly better, but for all the benchmarks, it found partitions of more than 4 banks which were better in terms of energy consumption (Column 5) than the 4-bank solutions found by the recursive technique. The energy savings versus a monolithic scratchpad are also displayed in Column 6.

Columns 7 to 9 display the savings of energy, access time, and silicon area versus a monolithic scratchpad when $w1=w2=0.4$ and $w3=0.2$. The energy and time savings are satisfying compared to a monolithic architecture. This is expected since the test was mainly run to optimize both metrics; $w1=w2=0.4$. However, a surprising result was the chip area that achieved an average of about 15% saving over a single bank scratchpad. This result is most
unexpected for two reasons: first, a scratchpad module that is larger in size is more compact than smaller scratchpad modules and, second, partitioning a scratchpad introduces extra wiring and decoding circuits that will only increase the chip area. That is why area savings of a partitioned configuration over a monolithic scratchpad is totally unexpected.

### 6.3.5 Complexity and Analysis

The algorithm at hand may seem similar to that of Benini et al. in that it starts exploring the search space exhaustively as long as the backtracking condition is not met. This phony similarity may suggest an analogous exponential complexity of the algorithm. However, in fact, a core difference between this algorithm and the one discussed in chapter 3 is the backtracking condition itself.

In Benini's algorithm, the condition under which backtracking occurs is by exceeding a monolithic scratchpad energy cost, at the time when any partitioned scratchpad will be energetically-less costly than the given energy threshold. This means that the Benini algorithm is literally exhaustive since the backtracking condition in Benini et al.'s algorithm is hardly ever met.

Whereas in our backtracking algorithm, the backtracking condition updates whenever a less costly partitioning is found; which means that with every update of the cost threshold, the search condition gets tighter and the chance of ignoring partitioning possibilities in the search space increases.

In addition, this algorithm is based on the intensity of memory accesses in the array space of the multidimensional signals mapped on-chip. In other words, the exploration of the search space is not blindly trying every possible address in the address space of the signal, but rather is steered by a more sensitive address set corresponding to lattice bounds. This address set used for steering the search process is extremely less in size than the signal size. The difference, in other words, is that Benini's algorithm is exponential in the size of the whole address space—being usually in the range of thousands or even millions of addresses—while Balasa's algorithm is exponential in the number of lattices. So, although the backtracking exploration algorithm still has an exponential complexity like Benini's, the additional exploration constraint that
no disjoint lattice assigned to the scratchpad can cross a bank boundary significantly reduces the search space, while yielding near-optimal results.

We may conclude that even though our algorithm has an exponential worst case complexity like Benini's, the first is far more efficient in practice than the second. This difference is also apparent from the experimental results obtained by our algorithm, discussed below. Experiments shown in the next section proved that the backtracking algorithm proposed in this chapter not only is much faster since it needs to explore a smaller solution space, but it can also find energetically better partitioning solutions for a number of banks when the exhaustive exploration is computationally-infeasible.

This makes the total number of possible partitions to be explored in a worst case scenario: \( \frac{(N-1)!}{(B-1)!(N-B)!} \), where \( N \) is the input size in words and \( B \) is the size of the array \( A \) previously defined as a set of addresses identifying the memory bank boundaries.

Another major distinction of this algorithm is the cost function. Our algorithm, and unlike all the previous methods, accounts for optimizing the three design objectives altogether rather than optimizing for only one objective while totally neglecting the others. This was made possible by the use of the sophisticated cost function proposed by the authors; which was found advantageous in terms of energy reduction and computational effort.

By the end of this chapter, we have now completed the exploration and analysis of four different approaches attempting to solve the banking problem. The following chapter concludes our research by comparing the results obtained for the studied approaches.
7 CONCLUSIONS

7.1 Summary

This thesis addresses the problem of partitioning the SPM address space, focusing mainly on data-intensive embedded applications. Table 7.1 shows a summary of the algorithms studied and introduced in our work to solve the research problem.

A recursive method introduced by Benini et al.[2][18] addressed the problem of scratchpad memory partitioning for computing the minimum energy partition of an on-chip scratchpad into M banks for energy optimization. Carried out according to the dynamic memory access profile of the embedded application and guaranteeing to find the global optimum taking into account the hardware and wiring overhead due to adding extra memory banks, this algorithm has an exponential complexity of $\theta(N^M)$.

Since dynamic programming (DP) is a well-known methodology for solving optimization problems, and asserting that our partitioning problem is solvable using this class of techniques, we studied a DP technique presented by Angiolini et al.[3]. This algorithm does find a global optimum solution without requiring any user-set bound on the number of partitions and takes into account partitioning overhead, with an efficiency that is polynomial in the sizes of the memory trace and the SPM--$\theta(N.C^2)$. However, the algorithm suffers crucial problems due to using the float cost function for indexing.

Another DP approach inspired by the matrix multiplication algorithm in [49] was proposed, avoiding the weak points of the previous method and optimizing its space requirements.

Finally, a novel recursive approach with backtracking is presented. This approach is steered by an analysis of the intensity of read/write accesses within the array space of the multi-dimensional signals rather than observing scalar memory locations individually. Different from previous techniques, a cost function that takes into account the three major design objectives--energy consumption, performance, and die area-- was introduced. Different from previous techniques that have as main input the execution trace of the application, the approach made use of a memory framework that starts from
the behavioral specification, deciding in a preliminary phase the data assignment to the memory layers and the mapping of signals to the physical memories. The proposed approach proved to be computationally fast even for SPMs of a larger size, being able to explore banking solutions with a larger number of partitions.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Algorithm 3.1</th>
<th>Algorithm 4.1</th>
<th>Algorithm 5.1</th>
<th>Algorithm 6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methodology</td>
<td>Recursive</td>
<td>Dynamic Programming</td>
<td>Dynamic Programming</td>
<td>Recursive with Backtracking</td>
</tr>
<tr>
<td>Input</td>
<td>Memory Trace</td>
<td>Memory Trace</td>
<td>Memory Trace</td>
<td>Behavioral Specification</td>
</tr>
<tr>
<td>Output</td>
<td>Partitioning Configuration</td>
<td>Partitioning Configuration</td>
<td>Partitioning Configuration</td>
<td>Mapping, Assignment and Partitioning Configuration</td>
</tr>
<tr>
<td>Cost Metrics</td>
<td>Energy</td>
<td>Energy or Performance</td>
<td>Energy, Performance or Area</td>
<td>Weighted Sum of Energy, Performance and Area</td>
</tr>
<tr>
<td>Measurements for Cost Calculation</td>
<td>Coumeri and Thomas Model [22]</td>
<td>N/A</td>
<td>CACTI 6.5 Simulator [36]</td>
<td>CACTI 6.5 Simulator [36]</td>
</tr>
<tr>
<td>Optimality</td>
<td>Yes, for a given M value.</td>
<td>Yes.</td>
<td>Yes, if Gran=1.</td>
<td>Near-optimal, for a given M value.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>$\theta(N^M)$</td>
<td>$\theta(N.C^2)$</td>
<td>$\theta(N^2)^9$</td>
<td>$\theta(A^M)^{10}$</td>
</tr>
</tbody>
</table>

Table 7.1 Summary of the discussed algorithms.

Memory metrics—that is, energy consumption, access time and chip area—for different scratchpad configurations were measured using CACTI 6.5[36]; an analytical tool that takes a set of cache, scratchpad, or DRAM parameters as input and calculates times of memory accesses for read/write operations, energy spent per access, static power, and chip area.

9 This is a worst case complexity, as practical tests of the algorithms proved to be far more efficient in practice since the matrix elements actually getting processed are less than half the size of the matrix.

10 Given that the size of $A$(the number of boundaries of the partitions yielded from code analysis is roughly a couple of tens) is usually much less than $N$(the size of the execution trace can be up to a couple of Millions). Also, this complexity is cut down in practice due to the wise use of backtracking.
7.2 Obstacles and Constraints

As it is the case with any project, we encountered a few obstacles during the execution of this thesis. Table 7.2 shows some of these obstacles, how we tried to get over them and their effect on the overall outcome of the research.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Circumvention</th>
<th>Effect on Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Implementations of previous techniques not available for comparison purposes.</td>
<td>We tried our best to follow the description published by the models' authors. Also, the implemented model is verified by comparing the running results of our implementation with the published results for the example benchmarks. After gaining confidence that the implemented model behaves as specified in the papers, we used it for testing our affine application.</td>
<td>Even though our implementation may have identical output to the one published by its owners for the sample benchmarks, this is not guaranteed to be the case for all other benchmarks. Also, other statistics (like the actual elapsed time) will most probably differ since the details of the original implementations are not known. However, the overall complexities of the algorithms are verified.</td>
</tr>
<tr>
<td>2. Simulation tool did not provide data for smaller sized memory banks.</td>
<td>Extrapolation was used.</td>
<td>Expected decrease of accuracy for the results obtained for memory banks less than 512 bytes.</td>
</tr>
<tr>
<td>3. Thesis writing process took a longer time than originally planned.</td>
<td>Extended an extra semester.</td>
<td>The extended period allowed a few more iterations on writing and enhancing the thesis document.</td>
</tr>
</tbody>
</table>

Table 7.2 Obstacles and constraints faced during the research.

7.3 Future Directions

This work is far from complete. The following are some possible enhancements that can be further investigated to develop this research.
7.3.1.1 **Idleness Investigation**

The thesis does not investigate the idleness of data stored in the banks, in order to be able to put the banks into a 'sleep' mode when they are not accessed for a longer time.

7.3.1.2 **Support for Parallel Programming**

Exploiting multiple memory banks is a challenging problem for compilers in parallel applications due to the instruction-level parallelism, small numbers of registers, and highly specialized register capabilities of most DSPs [1]. Memory hierarchy can be designed to support higher memory bandwidth by allowing multiple data memory accesses to occur in parallel.

7.3.1.3 **Genetic Programming**

The search space of all possible memory partitions can be easily proved by counter-example that total energy is not a single-minimum function over the solution space; there are many local minima. This observation seems to indicate that the memory partitioning problem can be most conveniently solved with heuristic techniques, such as genetic algorithms or randomized search, that do not guarantee global optimality.

![Figure 7.1 The distribution of memory accesses to the scratchpad address space assuming the 2-D array A is mapped into the on-chip scratchpad column-by-column. Dotted lines showing possible](image-url)
partitioning positions of the on-chip address space, if intermittent address ranges are to be allowed on the same memory bank.

In addition, the idea of including intermittent data ranges in the same partition is not considered in any of the discussed work. A possible reason may be to avoid complicating the decoding circuits of the scratchpad memory and hence save energy, area and enhance overall performance. However, it may be beneficial to investigate the idea and incorporate the decoding logic as a parameter in evaluating the cost of a partitioning.

7.3.1.4 Support for Multi-Task Processors

Even though our work is basically meant for specialized processing DSPs—that is processors mainly designed to process limited DSP functions--, it may be possible to extend the research problem to account for a broader range of functions. In other words, instead of limiting the banking solution to a specific task, the spectrum of functions supported by the banking solution may be broadened if the banking algorithm can be devised in a way that takes into account multiple behavioral specifications instead of only one.

For example, the proposed memory management system in Figure 6.3 may be tuned to find memory hierarchies for more flexible DSP processors; like the TMS320C3x for instance. An expected drawback of this enhancement is harming the optimization targets for the individual functions. However, it may prove beneficial on the long run. Perhaps a study of the potential amortized analysis of the different functions performed by the processor needs to be made before deciding on the final banking solution.
REFERENCES


[12] P.R. Panda, F. Catthoor, N. Dutt, K. Dankaert, E. Brockmeyer, C. Kulkarni, and P.G. Kjeldsberg, Data and memory optimization techniques for embedded systems,


[23] P. R. Panda, N. D. Dutt and A. Nicolau, On-chip versus off-chip memory, the data partitioning problem in embedded processor based systems, *ACM Transactions on*


This appendix lists memory data concerning power, access times and area used in this research and obtained using an interface to CACTI 6.5.

The data can be found at the following folder:

Appendices\CACTI Results\CACTI Simulation Data-SPM.xlsx

The Excel file contains four tabs for data obtained with different technologies: 32nm, 45nm, 68nm and 90 nm.
APPENDIX B – TESTING RESULTS

The optimal partitioning cut sets found by the algorithm in chapter 5 are shown in this appendix along with data captured for each generated solution; like the total energy consumption, the total time cost of the partitioning and the total area needed for the partitioning. The memory simulation data were acquired for the 32 nm² technology, scratchpad memory type.

The results in this appendix were obtained by running a C++ implementation of the algorithm on a PC with an i5 core at a 2.5 GHz processor. The program was tested using the same benchmark used to test the previous approaches with different values of Φ and granularity.

The data can be found at the following folder:

Appendices\DP Algorithm 5.1 Results\Algorithm 5.1 Results.xlsx

The Excel file contains three tabs for data obtained with each optimization target: energy, performance and area.