Compilation and Scheduling Techniques for Embedded Systems

Hassan Salamy
Louisiana State University and Agricultural and Mechanical College

COMPILATION AND SCHEDULING TECHNIQUES FOR EMBEDDED SYSTEMS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
Hassan Salamy
B.E., Lebanese American University, 2003
M.S. in E.E., Louisiana State University, 2006
August 2009

To my family, friends, and mentors... needless to say to you the reader...

Acknowledgments

I would like to thank my advisor Dr. Jagannathan Ramanujam (Dr. Ram) for his continuous support academically and financially. His hints and ideas when I was stuck will always be appreciated and acknowledged. Dr. Ram gave me much freedom in choosing my projects and was always very flexible in deadlines. I will always be thankful for the friendship that I have built with him during the years I spent at LSU as a graduate student. Thank you Dr. Ram for giving me the opportunity to go to different conferences that would not have been possible without your financial and academic support. Such conferences played crescent role in my overall academic experience. Also, I would like to thank my committee members, Dr. J. Trahan, Dr. R. Vaidyathanan, Dr. D. Carver, and Dr. M. Vicente, for serving on my committee and reading this thesis.

This dissertation would not have been possible without the unconditional love and support of my parents. Thanks for their financial support, I was able to get through the first year of my studies here at LSU. Their love and prayers have always been a motivation for me to give my best possible effort. Thank you for being such a great parents and mentors. I would also like to express my love to my two sisters Samar and Samah and their five angels Hadi, Mortada, Jawad, Hana'a, and Sara.

A special thank you is to my brother Rami who has always believed in me. He has always been proud of me and he has started calling me doctor since the time I sent my graduate application out. Thank you for being such a great brother and a great friend. You have always been a big motivator for me throughout my academic life.

Last but not least, I would like to thank all the great friends and colleagues at the Electrical Engineering Department at LSU that made my stay a pleasant one. Throughout the five years of graduate school, I was able to build strong friendships with few people in the Islamic Center of Baton Rouge and in Baton Rouge. I will always appreciate your friendship.

To all people that have loved and supported me either directly or indirectly, thank you and May Allah bless you.

Table of Contents

List of Tables

List of Figures

Abstract

Embedded applications are constantly increasing in size, which has resulted in increasing demand on designers of digital signal processors (DSPs) to meet the tight memory, size and cost constraints. With this trend, memory requirement reduction through code compaction and variable coalescing techniques are gaining more ground. Also, as the current trend in complex embedded systems of using multiprocessor system-on-chip (MPSoC) grows, problems like mapping, memory management and scheduling are gaining more attention.

The first part of the dissertation deals with problems related to digital signal processors. Most modern DSPs provide multiple address registers and a dedicated address generation unit (AGU) which performs address generation in parallel to instruction execution. A careful placement of variables in memory is important in decreasing the number of address arithmetic instructions leading to compact and efficient code. Chapters 2 and 3 present effective heuristics for the simple and the general offset assignment problems with variable coalescing. A solution based on simulated annealing is also presented.

Chapter 4 presents an optimal integer linear programming (ILP) solution to the offset assignment problem with variable coalescing and operand permutation. A new approach to the general offset assignment problem is introduced. Chapter 5 presents an optimal ILP formulation and a genetic algorithm solution to the address register allocation problem (ARA) with code transformation techniques. The ARA problem is used to generate compact codes for array-intensive embedded applications.

In the second part of the dissertation, we study problems related to MPSoCs. MPSoCs provide the flexibility to meet the performance requirements of multimedia applications while respecting the tight embedded system constraints. MPSoC-based embedded systems often employ software-managed memories called scratch-pad memories (SPM). Scheduling the tasks of an application on the processors and partitioning the available SPM budget among those processors are two critical issues in reducing the overall computation time.

Traditionally, the step of task scheduling is applied separately from the memory partitioning step. Such a decoupled approach may miss better quality schedules. Chapters 6 and 7 present effective heuristics that integrate task allocation and SPM partitioning to further reduce the execution time of embedded applications for single and multi-application scenarios.

Chapter 1
Introduction

Aside from science fiction movies, the idea of a human-independent robots that perform daily life chores flexibly to a large extent or work effectively side by side with people is still a dream. The belief that artificial intelligence will be at a stage where a machine is capable of performing many operations flexibly independent of the control of human beings is still out of reach. The intelligent machines field has witnessed good advancements over the years but is yet to reach a stage of maturity, where the transistor-based (or other) devices surpass human intelligence and become smart enough to improve their own design with minimal or no interference from human beings. The human mind is still too complex to be understood by the field of medicine and so is the ability to build a machines that can imitate humans with accuracy. Many of these abstract tasks are far more complicated to be handled efficiently by machines as the step by step logic provided by digital computers is still far away from imitating the human brain.

However, aside from the world ruled by machines, the last fifty years have shown that computers are extremely capable in two important areas:

1. data manipulation such as word processing, spread sheets, web search, and database management; and

2. mathematical calculations such as those in engineering simulations, aircraft design, and real-time digital signal processing.

Microprocessors perform these tasks with different levels of efficiency; however, it is still expensive to make a device that is optimized for tasks in both areas.

General-purpose processors are extremely useful in handling a large scope of versatile tasks with a degree of efficiency. However, sometimes the degree of efficiency is more important than versatility. It is a hard problem to build a general purpose processor that is optimized for a large set of tasks. Therefore, special-purpose processors have secured themselves the essential role as the building block in most of the embedded systems in the form of electronic devices that we come across daily in our lives. As the name suggests, a special-purpose processor is a processor specifically designed to solve a small set of tasks very efficiently.

In today's world, embedded systems are found everywhere from homes, offices and hospitals to cars, planes and consumer electronic devices. An embedded system is a system that is designed for a particular purpose. The special-purpose computer in such a system is usually designed to perform one or a few dedicated functions often with real-time computing constraints. The processor (or computer) here is usually embedded as part of a complete device including hardware and software interacting with other devices such as mechanical parts, etc. In contrast, a general-purpose computer such as a personal computer usually serves a number of purposes such as data processing, internet surfing, spread sheet manipulation, etc.

Since an embedded system is dedicated to a specific task, the design engineers of an embedded system can reduce the size and cost of the product and increase the reliability and performance. Embedded systems are usually mass-produced to benefit from economies of scale. Embedded systems are found in a range of items from portable devices such as digital cameras and cellular phones to large stationary installations like washing machines and traffic lights.

1.1 Embedded Systems Design

Embedded systems can be specified in different levels of abstraction. Gajski's Y-chart [25] has identified the different views of these abstraction levels and their relationship using three domains. The three domains are defined as follows.

- Behavior: This is a set of performance specifications with operational characteristics. Subsystems are presented in their functional forms.

- Structure: Different subsystems (e.g., processors, memory units, and controllers) that help to realize the behavior and their interconnections are specified.

- Geometry: This refers to the physical implementation of the system that realizes a structural definition.

The Y-chart uses five levels of abstraction defined below.

- System: At this level, the system is described as a set of subsystems to be mapped to hardware or software components.
- Algorithmic: It represents the algorithmic step-by-step solution and the interaction between different subsystems.
- Register-Transfer (RT): The system is defined as a set of communicating register transfer logics (RTL) such as ALUs, MUXs, and registers.
- Logic: It represents the hardware implementation of the logic functions as a set of logic gates and flip-flops along with their interconnections.
- Circuit: This is the actual hardware level implemented from transistors, capacitors and resistors on a board.

Embedded systems typically have tight constraints on design and functionality. Such contraints add significant burden on the design and life cycle of an embedded system. System optimization at different levels can have a large impact on meeting the set of tight constraints. We list some of the characteristics and constraints that an embedded system engineer often works with.

- Real-time operation: Often, one of the most critical constraints in an embedded system is its real-time system operation, i.e., the time at which the output from the system is ready in response to an input. In some systems, if the results are not available at a certain time, then the system is considered to have failed [24]; such systems are called *hard real-time systems.* In other cases, a delay in the availability of results leads to degraded system performance [24]; such systems are called *soft real-time systems.* Signal processing and mission critical systems are examples of embedded systems with significant real-time operation constraints.
- Size and weight: Embedded systems often have tight contraints on size and weight, which are typically derived from the larger system that they are part of; the size and weight also impact

the overall cost [28]. Memory utilization has a direct impact on area and hence size [11]. Therefore, reducing memory size is critical in controlling the size and the cost of embedded systems.

- Safety and reliability: Many embedded systems used in critical applications requiring high levels of reliability and fault tolerance. One way to improve fault tolerance is by including redundant components in the design. Unfortunately, adding redundant components to an embedded system adversely affects size and cost.
- Operation in harsh environments: Some embedded systems operate in environments, where they need to be protected from extreme heat, fire hazards, lightning, vibrations, shock, etc.
- Low-energy operation: The battery life of some embedded systems is one of the critical design issues. In some cases, frequent battery replacement is difficult. Thus, a design aimed to reduce energy consumption is essential.
- Low system cost: Many embedded systems, e.g., consumer electronic devices, are mass-produced and therefore must be inexpensive in price and cheap to manufacture. In some cases, lower cost can be achieved partly through reducing memory requirements.
- Time-to-market: In several embedded system markets such as consumer electronics, the time to market (a commonly quoted number is six months) is essential for companies to establish and maintain competitiveness. Often a missed deadline can have a major impact on a company's market share of the product. The use of high-level software tools in design help in reducing time-to-market but may have an adverse effect on design quality. In addition, design for debuggability helps in improving design correctness while meeting time-to-market constraints.

In this thesis, we deal with two kinds of processors used often in embedded systems, namely, digital signal processors (DSPs) and multi-processor system-on-chip (MPSoC).

1.2 Digital Signal Processors

A digital signal processor (DSP) is an example of an embedded processor, i.e., a processor used in an embedded system. DSPs are specifically designed to handle digital signal processing tasks. An example of a digital signal processor is the charge coupled device (CCD) image sensor in digital cameras. CCD image sensors are electronic devices that are capable of transforming a physical image (or light pattern) into an electric charge pattern. These sensors require real-time operation.

DSPs are often used in real-time processes and thus must have predictable execution times compared to general purpose processors. For instance, most people will not mind waiting for a *Word* document to be converted to *pdf* format, no matter how long it takes. However, DSPs are often used in applications where the processing is continuous and thus the execution time must be predictable. DSPs have seen tremendous growth in the last decade, and have been used in a wide variety of devices such as MP3 players, wireless phones, and scientific instruments.

1.3 DSPs with Address Generation Units

Digital signal processors form the core functions in many portable electronic devices designed under tight constraints, namely, cost, size and weight, while meeting constraints on high levels of performance and real-time constraints. Such designs usually have limited memory as a way to meet cost and size constraints. In contrast, the memory requirement for the execution of signal and video processing codes on embedded processors is significant. Moreover, since the program code resides in the on-chip ROM, the size of the code directly translates into silicon area. As a result, code minimization becomes an important step in reducing the amount of memory needed.

Many DSPs such as the Texas Instruments TMS320C5x, NEC 77110, Motorola DSP56000, Analog Devices ADSP21xx and ST D950 have address generation units (AGUs) [40]; see Figure 1.1. The architectures of such DSPs are very irregular and only indirect memory addressing is supported. In such architectures, the AGU is responsible of calculating the effective address of a memory location that will be accessed, since the base-plus-offset addressing mode is often not supported in these DSPs. An extra instruction is needed in general to add (resp. subtract) an offset to (resp. from) the current address in the address register to compute the next address. However, such architectures support auto-increment/decrement of the address register, as part of executing the current instruction. When there is a need to add an offset of 1 or subtract an offset of 1 from the current address, this can be done in parallel with the same LOAD/STORE instruction using auto-increment/decrement. This does not require an extra address arithmetic instruction in the code and therefore can decrease code size. Statistics show that the programs for DSPs can have up to 50% address arithmetic instructions [75].

One solution for minimizing the number of instructions needed for address computation in *scalar* based codes is to perform *offset assignment* of the variables (OA). Offset assignment refers to the problem of placing the variables in memory to maximally utilize auto-increment/decrement operations and thus reduce code size. It is referred to as *simple offset assignment (SOA)* when there is only one address register (AR), and as *general offset assignment (GOA)* in the case of multiple available ARs. Work so far has addressed SOA and GOA assuming a memory location is allocated for each variable for the duration of the entire program. Allocating variables with non-overlapping lifetimes to the same memory location is referred to as *variable coalescing.* The SOA and GOA problems studied in Chapters 2, 3 and 4 include the case of variable coalescing.

As many DSP algorithms have an iterative pattern of references to array elements within loops, an efficient generation of memory addresses for array references in loops is an important step for generating efficient code for array-intensive embedded applications. This problem is referred to as *address register allocation (ARA),* and is studied in Chapter 5. In both OA and ARA considered in this dissertation, the auto-modify range is [-1,1]. OA and ARA are both NP-complete [47, 7].

1.4 Multi-Processor System on Chip

System designers are finding it increasingly difficult to achieve more performance out of single processors due to clock and power constraints. As a result, achitectures with multiple processors on a single chip have become a viable solution to achieving higher level of performance to solve a broad range of problems from both high-end and low-end computing. Multiprocessor systems-on-chip (MPSoCs) that include a large number of different processing cores are now common for a variety of reasons, especially in embedded systems.

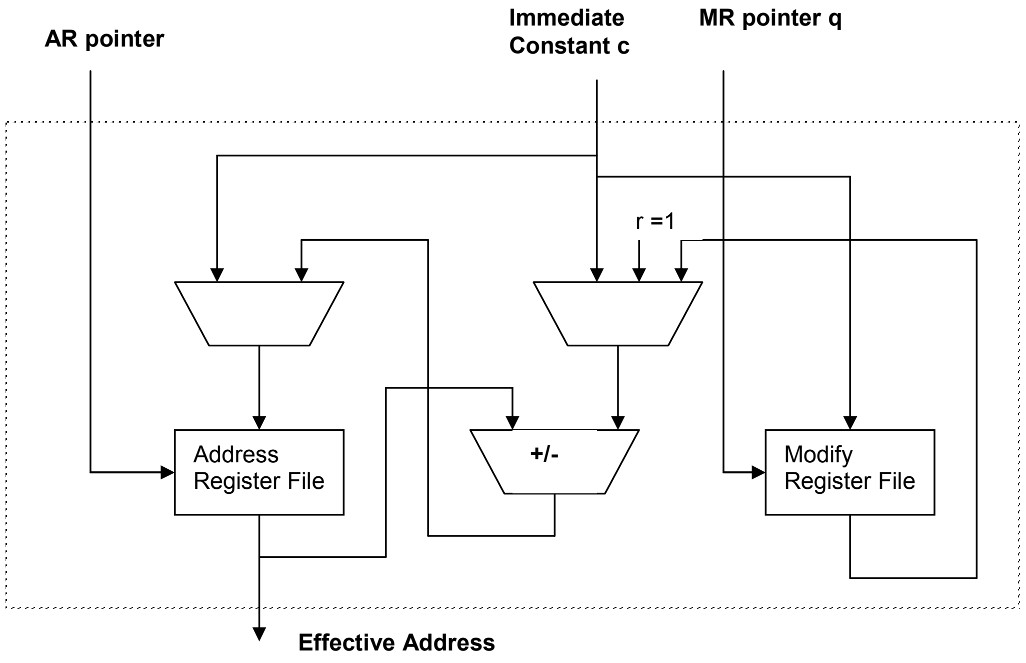


FIGURE 1.1. A typical Address Generation Unit (AGU) contains a modify register file, address register file, and ALU.

- The design, validation and verification of a chip multiprocessor consisting of multiple simple processor cores is easier than that of a complex single-processor system [36, 21, 54, 56].
- An MPSoC can be clocked at a reduced frequency, and this can lead to reduced power consumption without significant performance loss.
- Since each processor in an MPSoC can be indidually controlled, there are opportunities for energy reduction.
- Architecturally, an MPSoC design can result in better utilization of the available chip area.

Generally speaking, an MPSoC consists of multiple heterogeneous processing elements (PEs), memory hierarchies, and I/O components interconnected by complex communication architectures; see Figure 1.2. Such architectures provide the flexibility of simpler design, high performance and optimized energy consumption. An example of an MPSoC is the *Nomadik* multimedia application processor from ST Microelectronics [4]. This MPSoC is deployed in 2.5G/3G mobile phones and personal digital assistants (PDAs).

While embedded systems become increasingly complex, the processor-memory speed gap has continued to increase; over the last several years, increase in memory access speed has failed to

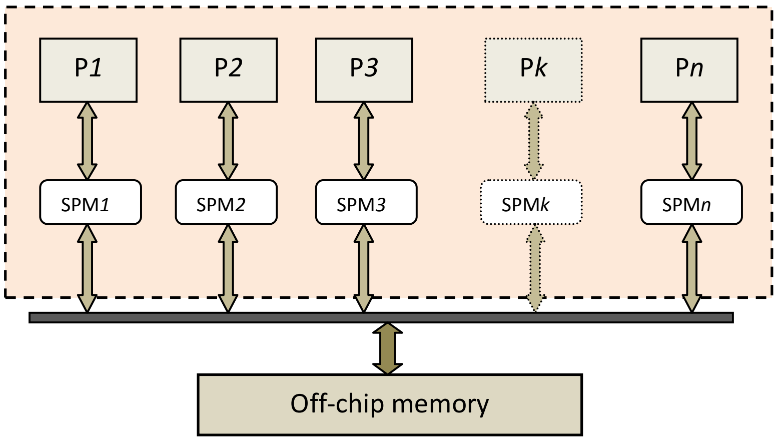


FIGURE 1.2. An architectural model example with *n* processors, SPM budget, off-chip memory and interconnection buses.

keep up with the increase in processor speed. This makes the memory access latency a major aspect in scheduling embedded applications on embedded systems. This increasing processor-memory speed gap is more of a problem in the case of MPSoCs due to the heavier contention on the network and the use of shared memories in some cases.

Hardware-based caches were always an attractive solution to bridge the speed gap between the processor and the memory. However, hardware based caches have many disadvantages.

1. Caches are among the major energy consumers among components of computing system.
2. Often, higher cache miss rates may occur due to the lack of predictability of future accesses and caches are subject to conflict, capacity and compulsary misses.
3. Using caches in systems with real-time requirements is not effective since their impact on worst-case execution time (WCET) is generally hard to predict. Caches incur unpredictable data access time (or latency), which is unacceptable for real-time embedded applications.
4. With caches, effective data prefetching to hide latency is harder to achieve since programs often fail to expose useful spatial locality in the data accesses.

Due to those and other reasons, recent research shows that caches are not ideally suited for multimedia applications with regular data access patterns. Execution time predictability is a critical issue for real-time embedded applications; this makes the use of data caches not suitable since it is hard to model the exact behavior and to predict the execution time of programs. To alleviate such problems, many modern MPSoC systems use software-controlled memories known as *scratchpad memories* (SPM).

An SPM is fully software-controlled and hence the execution time of an application on such memories can be predicted with accuracy since data movement is also software controlled. Unfortunately, scratchpad memories are expensive and hence they are usually of limited size and as a result not all the application data variables can be stored in the on-chip scratchpads. Many multiprocessor system-on-chip models use a memory hierarchy with slow off-chip memory (DRAM) and fast on-chip scratchpad memories. Such a hierarchy means that proper allocation of variables to the on-chip memory plays an essential role in reducing the off-chip accesses. The execution time of a program on a processor depends on how much SPM is allocated to that processor as accessing an element from the off-chip memory is usually in the order of 100 times slower than accessing elements stored locally in the on-chip memory.

There is a large number of complex embedded applications consisting of multiple concurrent real-time tasks [48]. These tasks can then be divided into subtasks by application designers. The computation time for each task depends on the amount of SPM allocated to the processor executing this task. The problem of task scheduling and memory allocation on MPSoCs is NP-complete [38]. Traditionally, these two steps—task scheduling and memory allocation—are done separately, where tasks are first scheduled and then the SPM budget is partitioned among the processors. Such a decoupled approach may not result in better schedules in terms of minimizing the computation time of the whole application. The appropriate configuration of a processor's scratchpad memory depends on the tasks scheduled on that processor. Therefore, the integration of those two steps is critical to improve the performance.

Unlike current approaches that have studied the task scheduling and memory partitioning problems as two separate problems, we solve those two problems in an integrated fashion. Chapter 6 deals with developing heuristics for the single application scenario that perform task scheduling and SPM memory partitioning in an integrated fashion where the private on-chip memory budget allocated to a processor is decided dynamically as tasks are mapped to this processor. Chapter 7 extends the heuristics in Chapter 6 to deal with more than one application concurrently using the MPSoC.

1.5 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 presents our solution to the simple offset assignment problem with variable coalescing (CSOA). We present an effective heuristic and test it on a bunch of real life embedded applications from *MediaBench.* We further improve the CSOA results using a simulated annealing approach. Our approach to the general offset assignment problem with variable coalescing (CGOA) is presented in Chapter 3. An effective heuristic for the CGOA problem is designed and tested. Results are further improved by using a simulated annealing approach.

In Chapter 4, optimal integer linear programming (ILP) for the offset assignment problem with variable coalescing is presented. Variable permutation is also utilized to find the best possible legal access sequence for the best cost. A new approach to the general offset assignment is presented in this chapter where the main idea is to partition the access sequence rather than the variables.

Chapter 5 presents the address register allocation (ARA) problem for array-intensive embedded applications. An optimal ILP and sub-optimal genetic algorithm (GA) solutions are presented. The solutions were further extended to allow code restructuring as a process to decrease the overall cost.

In Chapter 6, we present our work of developing an effective heuristic for the task scheduling/memory partitioning problem for a multi-processor system-on-chip where a single application is using the MPSoC at a time. In Chapter 7, the task scheduling/memory partitioning problem for MPSoC is studied in the case of multiple applications executing at the same time.

Conclusions and pointers to possible future work are presented in Chapter 8.

Chapter 2
Simple Offset Assignment with Variable Coalescing

As mentioned in the previous chapter, embedded systems are designed under tight constraints that vary from size and cost to safety and reliability. Cost and size are major factors in the design cycle of an embedded system as such systems are usually mass produced and are embedded in larger artifacts. Memory usually constitutes a big portion of the size and cost of the embedded system. As the embedded applications are growing in size so is the necessity for a bigger memory. Thus the ability to reduce the memory requirement becomes essential in keeping up with the application code size and the tight constraints on size and cost. One way to do so is through offset assignment and variable coalescing that will be thoroughly defined and explained in this chapter.

2.1 Offset Assignment

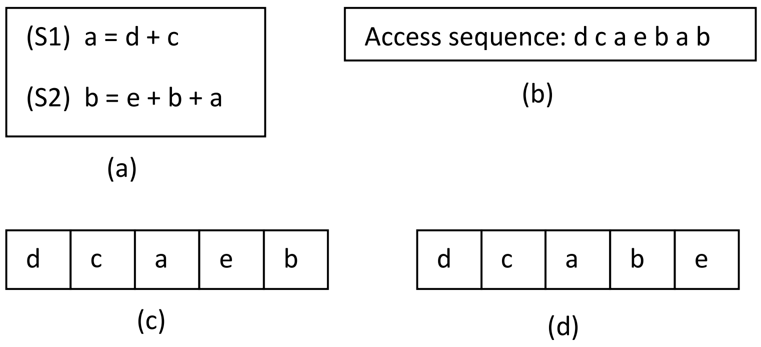
Embedded system applications are getting larger in size and this is exerting more pressure over the memory design of a DSP as usually the code is usually loaded into the ROM. And as a result, the size of the code translates into physical area. With all the tight size and cost constraints, the size of the embedded code becomes of substantial importance. Reducing the code size has great implications on the DSP design. Thus reducing the embedded application code size is a priority. As mentioned in the previous chapter, one way to reduce the size of an embedded application is known as offset assignment. Such a technique is utilized to decrease the number of explicit address arithmetic instructions. Such instructions usually constitute 20% - 30% of the total code instructions [75]. Variable coalescing is a technique also used to decrease such instructions as well as the memory requirement to store the variables. Variable coalescing usually increases the proximity between the variables in the memory and as a result less memory is needed to store the variables as well as fewer explicit address arithmetic instructions are needed.

The offset assignment problem refers to the placement of the variables in a program in the memory so that the number of explicit address arithmetic instructions is minimized. Different placements will lead to different generated code. Consider the simple example in Figure 2.1 with the code and the access sequence in Figure 2.1 (a), (b) and two different variable placements in Figure 2.1 (c), (d). The assembly code generated for the placements in Figure 2.1(c), (d) are respectively shown below with the instructions in bold are the explicit address arithmetic instructions. This simple example shows that a better placement of the variables in the memory will lead to fewer address arithmetic instructions which decreased from two in Figure 2.1(c) to one in Figure 2.1(d). This reduction is significant in real embedded benchmarks as the number of explicit address arithmetic instructions may account for 50% of the number of instructions.

```
LDAR AR0,&d          LDAR AR0, &d
LOAD *(AR0)+         LOAD *(AR0)+
ADD *(AR0)+          ADD *(AR0)+
STOR *(AR0)+         STOR *(AR0)
LOAD *(AR0)+         ADAR AR0,2
ADD *(AR0)           LOAD *(AR0)-
SBAR AR0,2           ADD *(AR0)-
ADD *(AR0)           ADD *(AR0)+
ADAR AR0,2           STOR *(AR0)
STOR *(AR0)
```

Simple offset assignment (SOA) refers to the case where there is only one address register. In SOA, each memory location or slot is assigned only one variable. Simple offset assignment with variable coalescing (CSOA) refers to the case where more than one variable can be mapped into the same memory location. Variable coalescing is intended to decrease the memory requirement by further decreasing the number of address arithmetic instructions as well as decreasing the memory requirement for storing the variables. Two variables can be coalesced if their live ranges do not overlap at any time which means that at any time, those two variables are not needed to be simultaneously live.

Definition 2.1: *An interference graph (IG) is a graph with a node to represent each variable in the access sequence and an edge between two nodes means the live ranges of the corresponding variables overlap and thus they interfere.*



```
(S1)  a = d + c

(S2)  b = e + b + a
```

(a)

Access sequence: d c a e b a b

(b)

d	c	a	e	b

(c)

d	c	a	b	e

(d)

FIGURE 2.1. (a) C code. (b) Access sequence. (c) Variable placement 1. (d) Variable placement 2.

In CSOA, an interference graph (IG) is constructed by examining the live ranges of all the variables. Each node in the graph represents a variable, and an edge between two nodes means they interfere and thus they cannot be coalesced. Coalescing two or more variables means that those variables will share the same memory location at different times as their live ranges are non-overlapping at all times of the program run.

Two variables can be coalesced if they meet all the following feasibility conditions:

- the two variables do not interfere;
- after coalescing, no node in the access graph has more than two selected edges incident on it; and
- the resulting access graph is still acyclic considering only the selected edges.

So instead of always selecting an edge as in SOA, CSOA can, in any iteration, either select an edge or coalesce two variables that meet the feasibility conditions.

Definition 2.2: *An access sequence (AS) is the order the variables are accessed in a certain program.*

Consider for example the following two statements in C code:

$a = b + c$

$b = a - d$

The access sequences of variables in statements S_1 and S_2 are $b\ c\ a$ and $a\ d\ b$, respectively. Thus the access sequence for the example C code is $b\ c\ a\ a\ d\ b$.

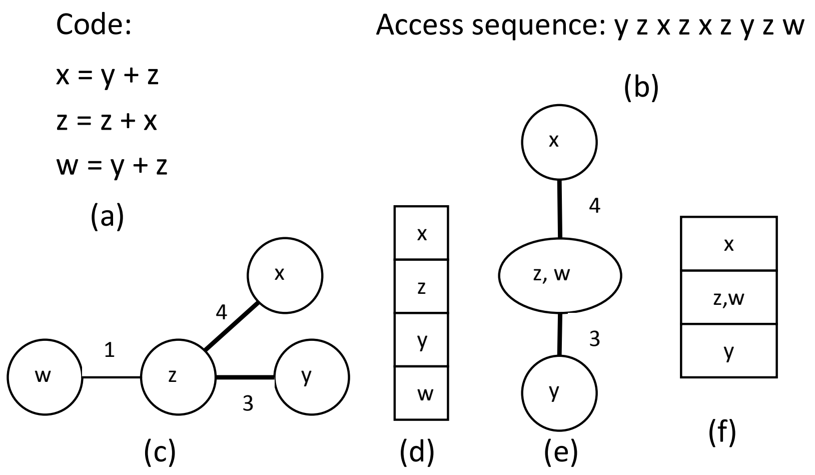


FIGURE 2.2. (a) C code. (b) Access sequence. (c) Access graph solution with no coalescing. (d) A Memory layout for OA. (e) Access graph solution with coalescing. (f) A memory layout for OA with variable coalescing.

Definition 2.3: *The access graph (AG) is a graph with a node for each variable and an edge of weight w between nodes u and v meaning that variables u and v appear consecutively w times in the access sequence.*

Consider the simple example in Figure 2.2 assuming only one available AR. One way to solve SOA is as follows. Given an access sequence of the variables, construct the access graph. Edges are then selected in decreasing order of their weights provided that choosing an edge does not introduce a cycle and it does not result in a node of degree more than two in the AG with only selected edges. Finally, the access graph considering only the selected edges will determine the placement of the variables in the memory. Figure 2.2 (d) shows the memory layout of such a solution with a final cost of one which is equal to the weight of the non selected edge (w,z). This cost of one represents the one address arithmetic instruction needed to update the address register pointing to the memory location of variable z to point to that of variable w. However, in the case of variable coalescing, variables z and w can be coalesced as they have non-overlapping live ranges. Figure 2.2 (f) shows the memory layout for the coalescing case with variables z and w mapped to the same memory location with a final cost of zero meaning that no explicit address register instruction is needed.

The simple offset assignment problem with variable coalescing studied in this chapter can now be defined as follows.

CSOA Problem Definition: *Given an access sequence of variables in a program, a DSP architecture of one address register (AR), and an interference graph, find the offset assignment of the variables in the memory so that the number of explicit address arithmetic instructions is minimized.*

2.2 Related Work

The problem of simple offset assignment was first discussed by Bartley [13]. Then Liao et al. [47] showed that the SOA problem is NP-complete and that it is equivalent to the Maximum Weight Path Cover (MWPC) problem. They proposed a heuristic for the SOA problem. Their heuristic is as follows. Given an access sequence of the variables, the access graph has a node for each variable with an edge of weight w between nodes a and b meaning that variables a and b appear consecutively w times in the access sequence. In their greedy heuristic, edges are selected in decreasing order of their weights provided that choosing an edge does not introduce a cycle and it does not result in a node of degree more than two. Finally, the access graph considering only the selected edges will determine the placement of the variables in the memory. One possible result of applying Liao's heuristic to the access sequence in Figure 2.3 (a) is shown in Figure 2.3 (c), where the bold edges are the selected edges and the final offset assignment is [e b a c d]. The cost of a solution is the sum of the weights of all unselected edges (i.e., non-bold edges). For the example in Figure 2.3 (a), the cost is 1 which represents the non-bold edge that refers to the one address arithmetic operation needed to go from a to e in the access sequence since variables a and e are mapped to non-consecutive memory locations.

Leupers and Marwedel [44] extended Liao's work by proposing a tie-break heuristic for the SOA problem. Liao et al. [47] did not state what happens if two edges have equal weight. Leupers and Marwedel used the following tie-break function: if two edges have the same weight, they pick the edge with the smaller value of the tie-break function $T_2(a,b)$ defined for an edge (a,b) as in Equation 2.5.

Atri et al. [8] solved the SOA problem using an incremental approach. They tried to overcome some of the problems with Liao's algorithm, mainly in the case of equal weight edges as well as the greedy approach of always selecting the maximum weight edges. Starting with an initial

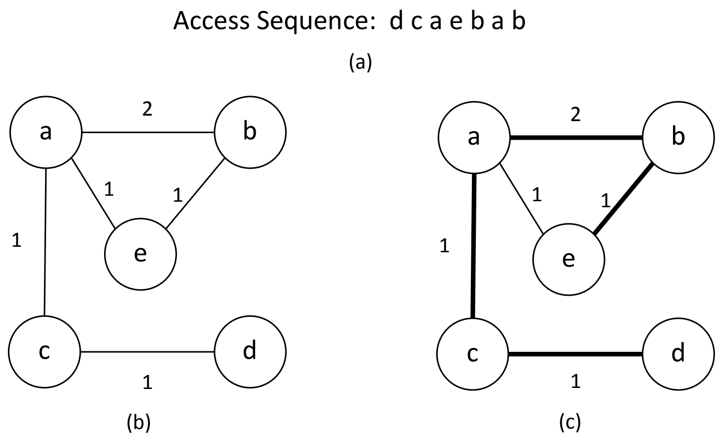


FIGURE 2.3. (a) Access sequence. (b) Access graph. (c) Liao's solution.

offset assignment (which could be the result of any SOA heuristic), their incremental-SOA tries to explore more points in the solution space by considering the effect of selecting currently unselected edges.

Leupers [41] compared several algorithms for simple offset assignment. Ottoni et al. [57, 58] studied the simple offset assignment problem with variable coalescing (CSOA). Their algorithm uses liveness information to construct the interference graph. In the interference graph, the nodes represent variables and an edge between two variables means that they interfere and thus they cannot be coalesced. The authors used the SOA heuristic proposed by Liao et al. [47] enhanced with the tie-break in [44], with the difference that at each step the algorithm chooses between (i) coalescing two variables; and (ii) selecting the edge with the maximum weight as in Liao's algorithm. Their algorithm finds the pair of nodes that can be coalesced with maximum *csave*, where *csave* represents the actual saving from coalescing this pair of nodes. At the same time, it finds the edge with the maximum weight *w* that can be selected using Liao's algorithm. If there are candidates for both coalescing and selection, then it will use coalescing if *csave* is larger than *w*, otherwise use selection.

In [81], the authors studied the cases of SOA with variable coalescing at the same time as [57]. Their coalescing algorithm first separates values into atomic units called webs by applying variable renaming. Their proposed heuristic starts by applying pre-iteration coalescing rules. Then the algorithm picks the two variables (i.e., nodes) with maximum saving for coalescing provided that they respect the validity conditions. If the saving is positive, then the two nodes are coalesced. Liao's SOA will then be applied to the new access graph. This process will continue as long as there are two variables that can be coalesced.

2.3 Our Heuristic

Our algorithm presented in Figure 2.5 integrates both selection and coalescing options in a way to minimize the total cost, which is represented by the number of address arithmetic instructions, as well as to decrease the memory requirement for storing the variables in memory. The algorithm takes as an input the interference graph (IG) and the access sequence and outputs the mapping of the variables to memory locations possibly with coalescing. From the access sequence, it constructs the access graph (AG), which captures the frequency of consecutive occurrence of any two variables in the access sequence. Then it sorts the edges whose end-point vertices interfere in decreasing order of their weights as a guide for selection. Since one of the purposes of our heuristic is to decrease the memory requirement for storing the variables, an edge (a,b) such that $(a,b) \notin$ IG will not be considered for selection. The vertices of such an edge will be left as candidates for coalescing which means that fewer edges will be considered for selection and thus more variables will probably be coalesced. Note that the selection of an edge may prevent future variable coalescing opportunities. So only those edges whose endpoints interfere will be considered as candidates for selection in each iteration of the algorithm.

In each iteration, all pairs of variables that meet the three conditions for variable coalescing (mentioned earlier) are candidates for coalescing. We define the following values:

$$Gain(a,b) = \frac{Actual\_Gain(a,b)}{Possible\_Loss(a,b)} \tag{2.1}$$

$$\begin{aligned} Actual\_Gain(a,b) = \; & W(a,b) \\ & + \sum_{\substack{x \in Adj(a) \cap Adj(b) \\ (b,x) \in Selected\_Edges \\ (a,x) \notin Selected\_Edges}} W(a,x) + \sum_{\substack{y \in Adj(a) \cap Adj(b) \\ (b,y) \notin Selected\_Edges \\ (a,y) \in Selected\_Edges}} W(b,y) \end{aligned} \tag{2.2}$$

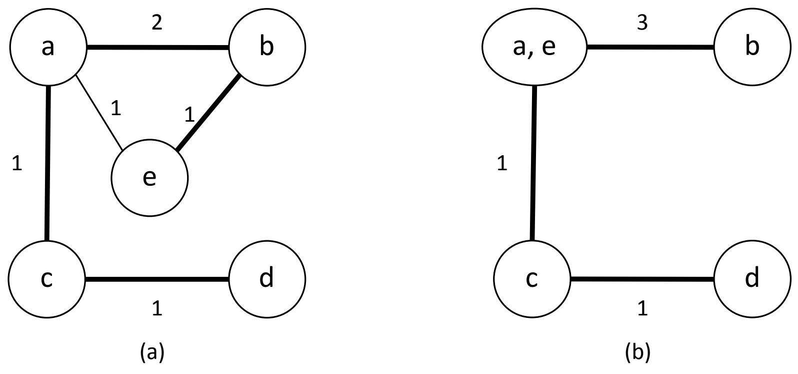


FIGURE 2.4. (a) Access graph. (b) Access graph after coalescing *a* and *e*.

$$Possible\_Loss(a,b) = 1 + \sum_{\substack{(a,x)\notin IG,(b,x)\in IG\\(b,x)\notin Selected\_Edges}} (a,x) + \sum_{\substack{(b,y)\notin IG,(a,y)\in IG\\(a,y)\notin Selected\_Edges}} (b,y) \tag{2.3}$$

A *Gain* value for each of those candidate pairs is calculated that captures the benefit of coalescing as well as the possible loss of future coalescing opportunities. The value $Gain(a,b)$ is defined in Equation 2.1 as the actual saving that results from coalescing variables *a* and *b* divided by the possible loss of future coalescing opportunities due to coalescing *a* and *b*. When variables *a* and *b* are coalesced, all edges incident at *a* and *b* of the form (a,x) and (b,x) will be merged, and if edge (a,b) exists, it will be deleted. When edges (a,x) and (b,x) are merged into edge (ab,x), if at least one of the edges was already selected, then (ab,x) will be marked as selected.

The value $Actual\_Gain(a,b)$, Equation 2.2, is basically the sum of the weights of the edges incident at *a* or *b* that were not selected and will become selected if variables *a* and *b* are coalesced as a result of being merged with a selected edge, plus the weight of the edge (a,b) if it exists. Consider the example in Figure 2.4 (a). The $Actual\_Gain(a,e)$ from coalescing variables *a* and *e* is equal to the weight of the unselected edge (e,b), as this edge will be merged with the selected edge (a,b), plus the weight of the edge (a,e) as this edge will no longer exist after coalescing *a* and *e*. Thus $Actual\_Gain(a,e)$ = 1 + 1 = 2.

The value $Possible\_Loss(a,b)$ is defined in Equation 2.3 as the sum of the edges (a,x) such that $(a,x) \notin$ IG, (b,x) is not selected, and $(b,x) \in$ IG plus the sum of the edges (b,y) such that $(b,y)\notin$

IG, (a,y) is not selected, and $(a,y) \in$ IG. In simple words, *Possible_Loss*(a,b) accounts for the number of edges incident at *a* or *b* whose corresponding vertices were interference-free and now interfere as a result of coalescing *a* and *b*.

As depicted in Equation 2.3, *Possible_Loss*(a,b) considers only vertices that are neighbors to *a* or *b*. Although other definitions of the loss can be used, we found that our definition captures the possible effect of coalescing on future solutions that can be constructed. Even though coalescing involves vertices and not edges, using the number of edges as the essence for the loss in Equation 2.3 leads to better results. The rationale behind this is that an edge whose corresponding vertices interfere will probably end up as a selected edge and thus it may prevent some future coalescing opportunities and this may degrade the quality of the final solution.

It is worth noting that although our heuristic integrates both selection and coalescing, it gives priority to coalescing, which can be clearly deduced from the definition of loss. We believe this is one of the main reasons for our improvements in terms of the cost as well as the memory requirement for storing the variables. The reason behind dividing *Actual_Gain*(a,b) with *Possible_Loss*(a,b) is the idea that coalescing two variables with a larger *Possible_Loss* value may prevent more future coalescing opportunities and thus may prevent reaching a solution of smaller cost compared to coalescing two variables with a smaller *Possible_Loss* value.

2.3.1 Coalescing and Selection Criteria

Among all the pairs that are candidates for coalescing, our algorithm picks the pair with the maximum *Gain*. If the algorithm is able to find a pair for coalescing as well as an edge for selection in some iteration, it will coalesce if the *Actual_Gain* from coalescing is greater than or equal to the weight of the edge considered for selection; otherwise, it will select the edge. One way our heuristic attempts to maximize the number of variables mapped to each memory location is to also allow the coalescing of pairs of variables with zero *Gain* value (if possible) after no more variables with positive *Gain* can be coalesced.

Coalescing variables without a good guide may prevent possible improvements over the standard SOA solution. Consider the example in Figure 2.6. Figure 2.6(b) shows Liao's greedy solu-

```
CSOA-ALGORITHM

    Input: Access sequence AS, Interference graph IG
    Output: Offset assignment
    Build the access graph (AG) from the access sequence.
    L = list of edges (x,y) such that (x,y) $\in$ IG in decreasing order of their
        weights using $T_1$ then $T_2$ for tie break.
    Coalesce = false.
    Select = false.
    Do
        Find a pair of nodes (a,b) for coalescing that satisfies:
            1. (a, b) $\notin$ IG.
            2. AG will still be acyclic after a and b are coalesced considering
               selected edges.
            3. No node will end up with degree $> 2$ considering selected edges.
            4. (a,b) has max Gain where Gain is calculated as in Equation 2.1.
        where $T_0, T_1$, and $T_2$ are the three tie break functions used in that order.
        If such a pair of nodes is found, then Coalesce = true.
        Among the edges that belong to L pick the first edge (c,d) such that:
            1. Selecting (c,d) will not result in a cyclic AG considering selected edges.
            2. Selecting (c,d) will not result in a node with degree $> 2$
               considering only selected edges.
        If such an edge is found, then Select = true;
        If (Coalesce && Select)
            If (Actual_Gain(a, b) $\geq$ Weight(c, d))
                Update access graph AG with (a, b) coalesced.
                Update interference graph IG with (a, b) coalesced.
                Update list L
            Else
                Select edge (c,d)
                Remove (c,d) from L.
        Else
            If (Coalesce)
                Update access graph AG with (a,b) coalesced
                Update interference graph IG with (a,b) coalesced
                Update list L
            Else if (Select)
                Select edge (c,d)
                Remove (c,d) from L.
    While (Coalesce || Select)
    Return offset assignment
```

FIGURE 2.5. Our heuristic for simple offset assignment with variable coalescing.

tion, [47]. The cost of this offset assignment is 4. Figure 2.6(c) shows a possible solution using the CSOA algorithm in [58] whose cost is also 4. Although there is a potential for improvement through variable coalescing, the algorithm in [58] fails to capture this possible improvement over Liao's solution. This is because their algorithm first chooses to coalesce vertices *b* and *e* since they have the maximum *csave* (*Actual_Gain*). However, this choice will prevent any future coalescing opportunities of positive *csave* provided that their heuristic picks edges (a, eb) and (d, eb) for selection which is a random choice in this case.

Our algorithm alleviates this shortcoming by calculating the $Possible\_Loss(b,e)$ = 5 and thus $Gain(b,e)$ = 3/5. As a result our algorithm first picks *a* and *b* for coalescing since $Gain(a,b)$ = 2/3; edge (b,e) will not be considered for selection since *b* and *e* do not interfere. The cost of the final solution using our heuristic is zero, as shown in Figure 2.6(d). Another possible solution by Ottoni's heuristic for the example in Figure 2.6 is presented in Figure 2.7 which is also not the optimal solution as it encounters a cost of 2.

2.3.2 Tie Break

Tie break is the process of deciding between two candidates that have the same gain value. For selection, we used two tie-break functions T_1 and T_2 defined below,

$$T_1(a,b) = degree(a) + degree(b) \tag{2.4}$$

$$T_2(a,b) = \sum_{x \in Adj(a)} W(a,x) + \sum_{y \in Adj(b)} W(b,y), \tag{2.5}$$

where $T_1(a,b)$ is the sum of the degree of *a* and degree of *b* in the access graph. $T_2(a,b)$ is the Leupers tie-break function defined as the sum of the weights of the edges that are incident at *a* plus the sum of the weights of the edges that are incident at *b*. If two edges that are candidates for selection have the same weight then we try to tie break using the function T_1; if T_1 cannot break the tie, we use T_2. An edge with smaller T_1 or T_2 will win the tie.

If two pairs of variables (a,b) and (c,d) that are candidates for coalescing are such that $Gain(a,b) = Gain(c,d)$, then we first try to break the tie using T_0 which is the *Actual_Gain* such that we choose the pair with the bigger *Actual_Gain*. If both candidate pairs have the same actual gain, then we tie break using T_1 followed by T_2, if needed.

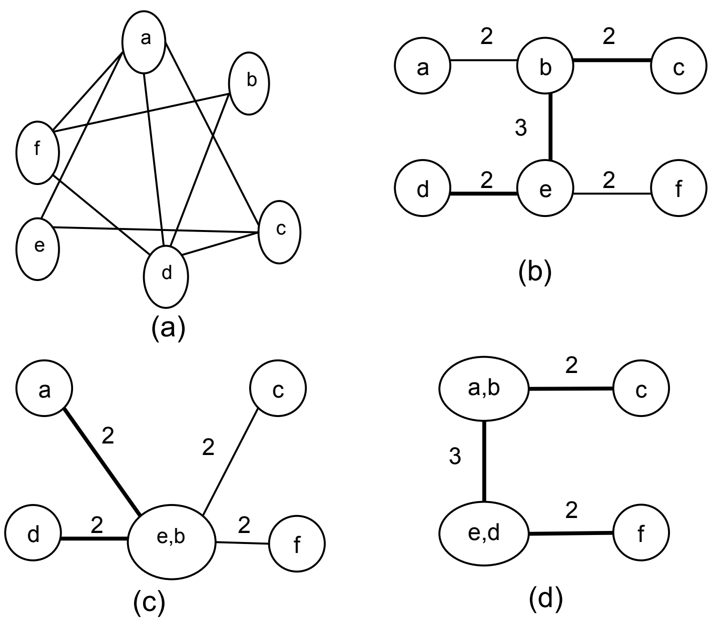


FIGURE 2.6. (a) Interference graph. (b) Liao's SOA greedy solution with cost = 4. (c) A possible solution from the Ottoni's CSOA with cost 4 where it fails to capture the potential improvements from coalescing. (d) The optimal solution using our algorithm with cost = 0.

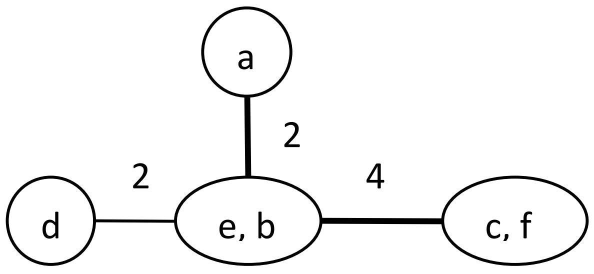


FIGURE 2.7. One possible final solution for the example in Figure 2.6 using Ottoni's CSOA.

2.4 Simulated Annealing

Since the offset assignment problem is NP complete, the heuristic presented in Section 2.3 will very likely produce a suboptimal solution. In order to further improve the results, we used a simulated annealing approach.

Simulated annealing (SA) [35] is a global stochastic method that is used to generate approximate solutions to very large combinatorial problems. The technique originates from the theory of statistical mechanics and is based on the analogy between the annealing process of solids and the solution procedure for large combinatorial optimization problems. The annealing algorithm begins with an initial feasible configuration, and then a neighbor configuration is created by perturbing the current solution. If the cost of the neighboring solution is less than that of the current solution, the neighboring solution is accepted; otherwise, it is accepted or rejected with some probability. The probability of accepting inferior solutions is a function of a parameter, called the temperature T, and the change in cost between the neighboring solution and the current solution. The temperature is decreased during the optimization process, and the probability of accepting an inferior solution decreases with the reduction of the temperature value.

The set of parameters controlling the initial temperature, stopping criterion, temperature decrement between successive stages, and number of iterations for each temperature is called the *cooling schedule* [35]. Typically, at the beginning of the algorithm, the temperature T is large and an inferior solution has a high probability of being accepted. During this period, the algorithm acts as a random search to find a promising region in the solution space. As the optimization progresses, the temperature decreases and there is a lower probability of accepting an inferior solution. The algorithm then behaves like a downhill algorithm for finding the local optimum of the current region.

2.4.1 Initial Solution

The initial configuration is usually chosen to be a random memory offset solution. However, since simulated annealing requires a significant amount of time in order to converge to a good solution especially for the large benchmarks used in our experiments, we decided to use the final solution from our heuristic as the initial solution for SA. Then we ran SA for a short period of time with a low probability of accepting a bad solution. The solution is basically a linear memory offset assignment as shown in Figure 2.8.

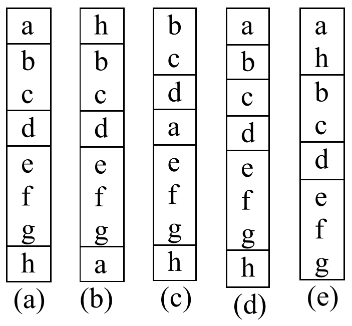


FIGURE 2.8. (a) Original memory contents. (b)-(e) Memory after applying different operations.

2.4.2 Neighborhood Transformation

The main operation of the simulated annealing is the neighborhood function. Starting from a current solution, the neighborhood function applies some operations to move into a new solution. We illustrate the neighborhood transformation using the example in Figure 2.8. For instance, a neighborhood solution is created by randomly selecting two memory locations and then swapping the variables placed in such locations.

The neighbor function can perform one of the following operations to the original memory content shown in Figure 2.8 (a).

- Exchange the contents of two memory locations, Figure 2.8 (b),
- Move the content of one memory location, Figure 2.8 (c),
- Uncoalesce a coalesced node into two or more nodes, Figure 2.8 (d), or
- Coalesce two memory locations, Figure 2.8 (e).

2.4.3 Cost Function and Cooling Schedule

Given an offset assignment, the cost is the actual cost from applying our simple offset assignment heuristic with variable coalescing. The cost is the actual number of explicit address arithmetic instructions generated based on the current offset assignment solution. The cost can also be the memory requirement to store all the variables but we resort to the former cost criteria in our simulated annealing algorithm.

The cooling schedule is the set of parameters controlling the initial temperature, the stopping criterion, the temperature decrement between successive stages, and the number of iterations for each temperature. The cooling schedule was empirically determined as follows.

1. $T_{init} = 400$.

2. The temperature reduction multiplier, α , is set to 0.8.

3. The number of iterations, M, is set to 5 while the iteration multiplier, β, is set to be 1.05.

The algorithm stops when the current temperature, T, is below 0.001.

2.5 CSOA: Example

For the sake of clarity, consider the example in Figure 2.9 where Figure 2.9(a) shows the interference graph (IG) and Figure 2.9(b) shows the original access graph (AG). Figures 2.9(c)–(h) show how the access graph is updated when our heuristic is applied to this example. Although not shown, whenever two nodes are coalesced, the interference graph (IG) will be updated to reflect the coalescing of the nodes as well as to update the interference edges accordingly. Table 2.1 shows the step-by-step execution of our algorithm and the criteria used for choosing the candidates for selection and for coalescing. Note that in Table 2.1 we do not show the coalescing candidates with zero *Gain*. Figure 2.9(i) shows the final solution with zero cost. If we run the CSOA algorithm in [58] on the same example presented in Figure 2.9, the cost of a possible final solution, shown in Figure 2.10, is 4.

2.6 Results

We implemented our techniques in the *OffsetStone* toolset [5, 41] and we tested them on the *MediaBench* benchmarks [39]. In Table 2.4, we compare our CSOA heuristic with four different techniques used to solve the simple offset assignment problem without variable coalescing (SOA), namely Liao et al. [47], Leupers' tie-break [44], incremental with Leupers' tie-break INC-TB [8, 41], and Genetic algorithm GA [43].

We measured the number of explicit address arithmetic instructions needed by each method. We presented the cost resulting from the SOA-OFU in Table 2.3 as a reference point. The cost in

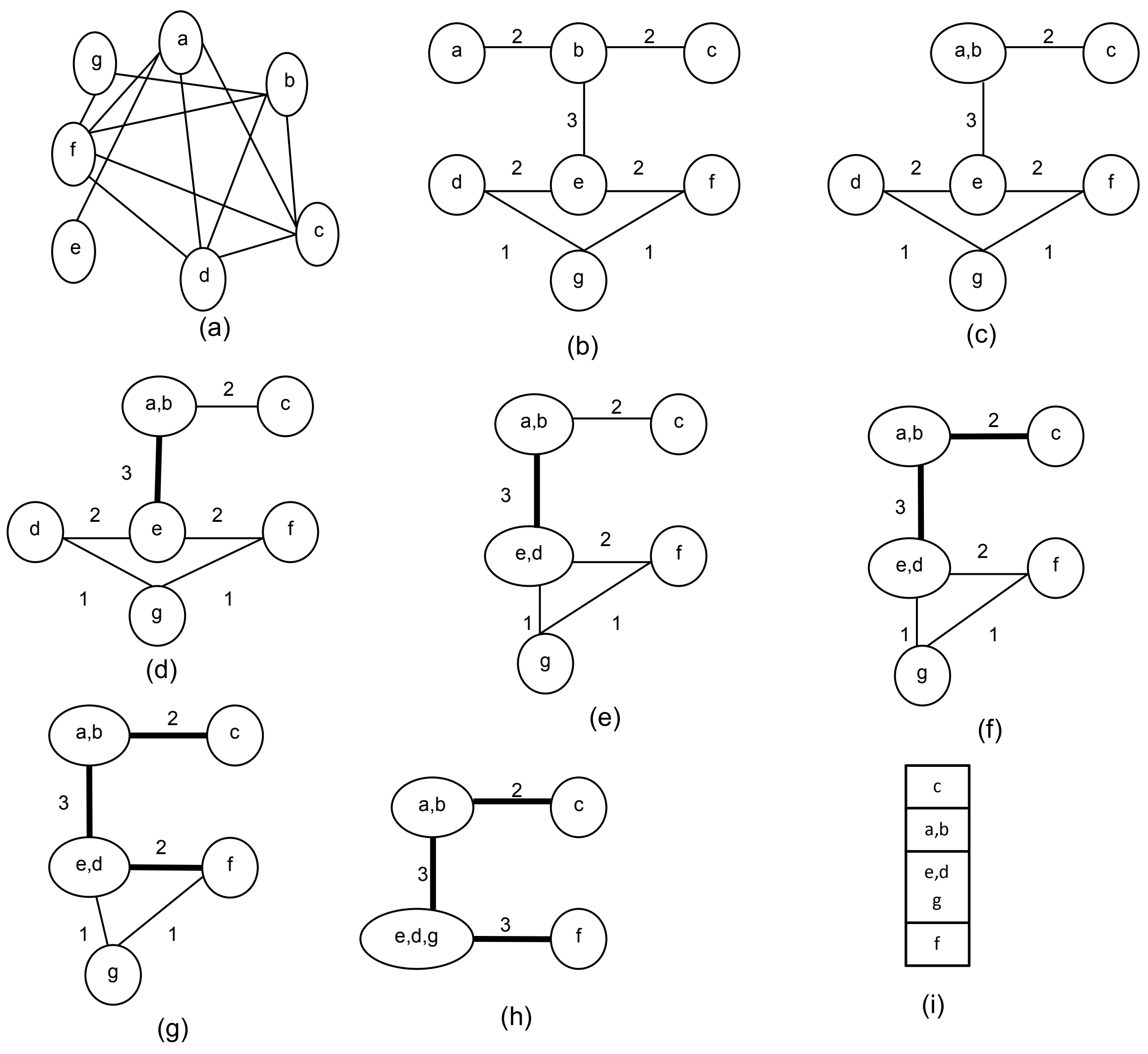


FIGURE 2.9. (a) The interference graph. (b) The original access graph. (c)-(h) The access graphs after each iteration of our algorithm. (i) The final offset assignment, which incurs zero cost.

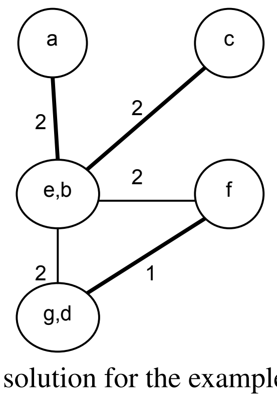


FIGURE 2.10. One possible final solution for the example in Figure 2.9 using Ottoni's CSOA.

TABLE 2.1. A step by step run of our algorithm on the example in Figure 2.9

Iteration	Coalesce Candidate				Selection		Decision
	Vertices	ActualGain	PossibleLoss	Gain	edge	Weight	
1	a,b b,e d,e g,d f,e	2 3 2 1 2	2 4 3 1 3	1 3/4 2/3 1 2/3	(b,c) (g,f)	2 1	Coalesce(a,b) Tie-break T_0
2	d,e g,d f,e	2 1 2	2 1 2	1 1 1	(ab,e) (ab,c) (g,f)	3 2 1	Select (ab,e)
3	d,e g,d f,e c,e	2 1 2 2	2 1 2 3	1 1 1 2/3	(ab,c) (g,f)	2 1	Coalesce (d,e) Tie-break T_0
4	ed,g	1	1	1	(ab,c) (ed,f) (g,f)	2 2 1	Select (ab,c) Tie-break T_1
5	ed,g	1	1	1	(ed,f) (g,f)	2 1	Select (ed,f)
6	ed,g	2	1	2	(g,f)	1	Coalesce (ed,g)

Table 2.3 is the actual number of explicit address arithmetic instructions resulted from generating the code with the offset assignment achieved using the SOA-OFU where no variable coalescing is allowed. OFU is a naive offset assignment algorithm based on order of first use where variables are assigned to offsets in the order of their appearance in the access sequence. Smaller numbers in the tables of results are better as they mean fewer address arithmetic instructions are needed.

The following is a brief description of each of the benchmarks used.

1. Adaptive Differential Pulse Code Modulation (ADPCM).
2. EPIC (Efficient Pyramid Image Coder) is an experimental image data compression utility.
3. PEGWIT is a program for performing public key encryption and authentication.
4. PGP uses ”message digests” to form signatures.
5. JPEG stands for Joint Photographic Experts Group, is a commonly used method of compression for photographic images.

6. MPEG2 is a standard for the generic coding of moving pictures and associated audio information.

7. GSM is a European standard for voice transcoding.

8. RASTA is for speech recognition.

9. G721: CCITT voice compression.

Our CSOA heuristic drastically reduces the cost of simple offset assignment when compared to heuristics that do not perform variable coalescing since variable coalescing increases the proximity between variables in memory, thus it reduces the number of update instructions.

In Table 2.5, we compare our results with those of two heuristics that perform SOA with variable coalescing, mainly Ottoni's CSOA [58] and Zhuang's CSOA [81]. Clearly our CSOA outperforms the two other heuristics. This improvement is due to the guide used in our choice between candidates for coalescing where we not only consider the actual saving but also an estimate of the possible loss in future coalescing opportunities. Also the idea of just considering edges whose endpoints interfere for selection increases the opportunity for coalescing nodes with maximum *Gain* as defined in Equation 2.1. The ability to coalesce depends on the selected edges and vice-versa. So an algorithm that can choose the right candidates for selection and coalescing, at the right iteration, and decide between them, should consider the influence of such a decision on future solutions. This is accounted for in our algorithm by defining the possible loss as a guide for the possible effect of coalescing on future solutions. The three tie-break functions T_0, T_1, and T_2 play a role in achieving the clear improvements to the final solution.

Our simulated annealing (SA) algorithm further improved the results by searching the feasible region for better solutions starting from the final solution of our heuristic. Results in Table 2.4 Column 7 shows that the SA further improved the results in all the cases in a short CPU time.

In Table 2.6, we show the reduction in memory slots needed to store the variables using our CSOA heuristic compared to that of those in [58] and [81]. We measured the percentage of memory slots needed compared to heuristics that do not perform coalescing where in such a case a memory

TABLE 2.2. Percentage of temporary variables.

Benchmarks	Temporaries (%)
adpcm	59.6
epic	48.1
g721	80.7
gsm	86.6
jpeg	65.2
mpeg2	65.6
pegwit	72.1
pgp	67.5
rasta	43.6

TABLE 2.3. SOA-OFU cost.

Benchmarks	SOA-OFU cost
adpcm	207
epic	6235
g721	718
gsm	1511
jpeg	10338
mpeg2	7981
pegwit	2249
pgp	7235
rasta	6626

TABLE 2.4. Comparison between different techniques for solving the SOA problem.

Benchmarks	Liao [47]	TB [44]	INC-TB [8] [44]	GA [43]	Our CSOA	SA
adpcm	138	132	132	132	58	54
epic	4508	4364	4352	4352	2119	2025
g721	526	506	506	506	138	122
gsm	1091	1052	1052	1052	159	147
jpeg	7112	6895	6875	6875	2202	2066
mpeg2	5706	5555	5547	5539	1780	1708
pegwit	1536	1399	1392	1392	607	554
pgp	5122	4862	4855	4855	1526	1403
rasta	4353	4287	4287	4287	864	847

slot is needed for each variable. Results show that our algorithm drastically reduces the memory requirement by maximizing the number of variables that are assigned to the same memory location, and it outperforms both other CSOA heuristics in all the cases. The reason behind this reduction is that we defined the *Gain* from coalescing in terms of possible loss in coalescing opportunities as well as due to the fact that we did not consider the edges (a,b) such that $(a,b) \notin$ IG as candidates for selection and this will result in more coalescing opportunities.

However, the main reason for our improvement over Ottoni's CSOA is that our heuristic allows zero *Gain* coalescing between nodes in the final AG. That is, we coalesce pairs of vertices (a,b) (if possible) such that $Gain(a,b) = 0$. This zero *Gain* coalescing will not reduce the cost in terms of the number of address arithmetic instructions but it will contribute to maximizing the number of variables mapped to a memory location. This explains the huge difference between the improvements in Table 2.5 and Table 2.6. Although a heuristic designed just to decrease the memory requirement for storing the variables may get better results than those in Table 2.6, it will be detrimental to the quality of the final solution in terms of the number of address arithmetic instructions. So our heuristic not only decreases the cost (which is defined as the reduction in the number of address arithmetic instructions), but also decreases the number of memory locations needed to store the variables.

TABLE 2.5. Results for different CSOA algorithms.

Benchmarks	CSOA-Ottoni	CSOA-Zhuang	Our CSOA
adpcm	62	66	58
epic	2264	2488	2119
g721	145	159	138
gsm	202	221	159
jpeg	2264	2750	2202
mpeg2	1955	2139	1780
pegwit	585	682	607
pgp	1628	1903	1526
rasta	921	1637	864

TABLE 2.6. The percentage of the memory slots needed using different CSOA heuristics with respect to the number of variables.

Benchmarks	Memory slots(%) [58]	Memory slots(%) [81]	Memory slots(%) our CSOA
adpcm	27.3	28.3	21.7
epic	27	26.6	18.5
g721	25	22.7	17.3
gsm	21.5	19.8	9.1
jpeg	34.5	25.7	18.8
mpeg2	31.8	21.9	17.1
pegwit	35.3	26.8	22.1
pgp	31.5	24.7	18.4
rasta	26.1	21.4	14

2.7 Chapter Summary

The offset assignment problem has received a lot of attention from researchers due to its great impact on code size reduction for DSPs. Reducing the code size is beneficial in the case of DSPs since the code is directly transformed into silicon area. The main idea of the ongoing research in this field is to decrease the number of address arithmetic instructions and thus the code size. The problem is studied as simple offset assignment (SOA) when there is one address register in the system. In this chapter, we presented a heuristic to solve the simple offset assignment with variable coalescing that chooses between selection and coalescing in each iteration by calculating the *Actual_Gain* and *Possible_Loss* for each pair of coalescing candidates. Results on real life benchmarks show that our algorithm not only decreases the number of address arithmetic instructions, but also drastically decreases the memory requirement for storing the variables by maximizing the number of variables that are mapped to the same memory slot. Simulated annealing further improved the final solution from our heuristic.

Chapter 3
General Offset Assignment with Variable Coalescing

In Chapter 2, we studied the problem of simple offset assignment with variable coalescing (CSOA). CSOA is the offset assignment problem when the system has only one available address register (AR). Embedded systems usually have more than one address register and thus the CSOA cannot be very helpful to such systems. The problem of offset assignment with a system of k address registers is referred to as general offset assignment (GOA). CSOA is essential in solving the CGOA problem as the solution approach to such problem is dividing it into multiple CSOA problems. An optimized solution to the CSOA problem will be propagated to a better CGOA solution and hence the importance of the technique presented in the previous chapter.

3.1 Problem Definition

The general offset assignment problem (GOA) refers to the case where there is more than one address register. In the literature, GOA solutions are based on partitioning variables among the available address registers. An important aspect in the GOA problem is how to partition the variables into L partitions ($L \leq k$), where k is the number of available address registers, so that the cost is minimized. There is no clear way to decide which variables should be mapped to which address registers.

To clarify the solution to the GOA problem, consider the example in Figure 3.1 with two available address registers $AR0$ and $AR1$. Figure 3.1(a) shows the original access sequence. Assume that variables a, b, c and d are mapped to $AR0$ and variables e and f are mapped to $AR1$. This variable partitioning is the optimal solution for the access sequence in Figure 3.1(a). Then two access sequences are extracted from the original access sequence. The first one represents the sequence of the variables mapped to AR_0, and the second represents the sequence of the variables mapped to AR_1. Figure 3.1(b) shows that access sequence for the variables mapped to address register AR_0 with the corresponding access graph in Figure 3.1(c). Applying SOA to the access graph in Figure

3.1(c) results in an offset assignment, Figure 3.1(d), of cost = 1 which represents the one explicit address arithmetic instruction needed to update AR_0 pointing to variable d at a certain program point to point to the address of variable b. Similarly, Figures 3.1 (e), (f), and (g) show the access sequence, access graph and offset assignment for the variables mapped to address register AR_1 with a cost of zero. The GOA solution, Figure 3.1(h), is the concatenation of the SOA solutions with the final cost of 3 which represents the sum of the costs of the SOA solutions plus an initialization cost of 2 for the two used address registers in the generated code. Below is the actual assembly code for the sample example in Figure 3.1 showing only the address registers accesses.

```
LDAR AR0,&c
LDAR AR1,&e
*(AR0)+
*(AR0)+
*(AR0)
SBAR AR0,3
*(AR1)+
*(AR0)+
*(AR0)-
*(AR1)-
*(AR1)+
*(AR0)
*(AR1)
*(AR0)+
*(AR0)+
*(AR0)
```

The general offset assignment with variable coalescing problem studied in this chapter can now be defined as follow.

CGOA Problem Definition: *Given an access sequence AS, an interference graph IG, and k address registers, find the best partitions of the variables among l address registers ($l \leq k$) so that the CSOA cost of each partition plus the ARs initialization cost is minimum.*

The main assumption of this problem is that a variable can be accessed by one address register throughout the program run. This is the assumption made by all the researchers that studied the general offset assignment problem and therefore we abide by it in this chapter. This assumption will be relaxed in Chapter 4.

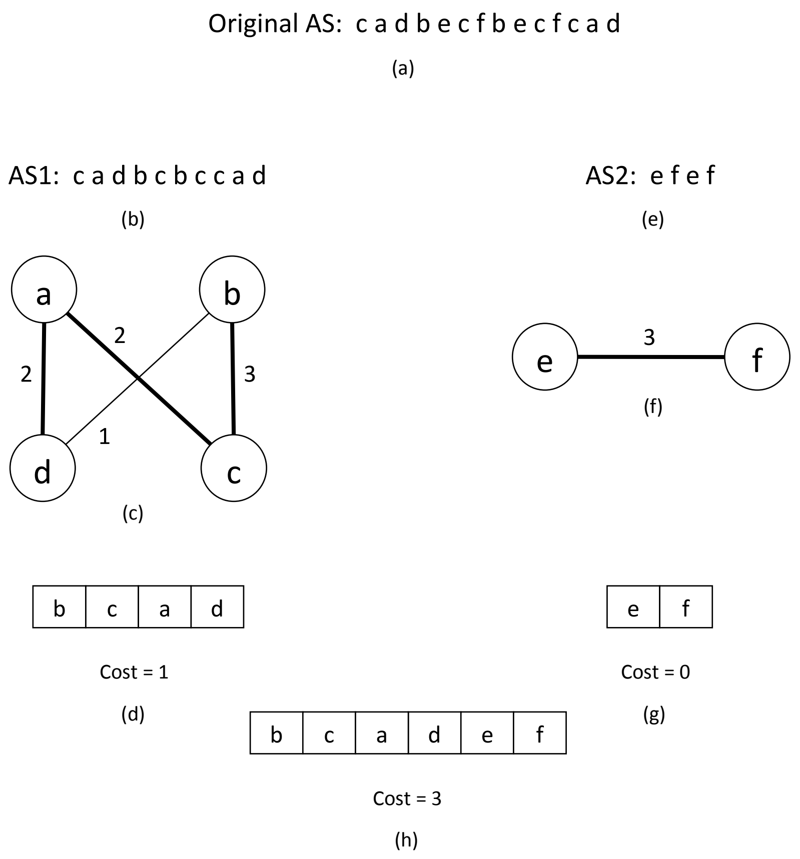


FIGURE 3.1. (a) Original access sequence. (b) Access sequence for AR_0. (c) Access graph for AR_0. (d) Offset assignment for AR_0. (e) Access sequence for AR_1. (f) Access graph for AR_1.(g) Offset assignment for AR_1. (h) Offset assignment for the GOA problem.

3.2 Related Work

Several researchers proposed heuristics to solve the GOA problem. The basic idea of those heuristics is to partition variables and then map each partition into an address register. The number of partitions cannot exceed the number of available address registers. SOA is then applied to each partition separately. The GOA solution is constructed by concatenating the SOA solutions. This idea was first introduced in [47] without mentioning how to form the partitions. Leupers et al. [44] proposed to form the partitions as follows. First sort the edges in the AG in decreasing order. Following this order, a disjoint edge will be mapped to each address register, if possible. Then the rest of the variables, if any, will be mapped to partitions such that a variable x is mapped to partition p with the minimum new cost if x is assigned to p.

Zhuang et al. [81] studied the GOA problem with variable coalescing. Their heuristic starts by applying variable coloring using the register coloring technique in [53]. If $2k$ colors are enough to color the AG, where k is the number of address registers, then the cost of the solution is the initialization cost. Otherwise, the heuristic progresses in a similar fashion to [44] using the cost from their proposed CSOA heuristic as a criteria to assign variables to partitions. Ottoni et al. [58] then proposed a CGOA heuristic which first sorts variables in decreasing order of their number of interferences. Each variable is assigned to the partition with the least number of interferences. The size of the partition is used as a tie break when a variable has the same minimum number of interferences with more than one partition. The priority is given to the partition with the fewest variables. Recently Huynh et al. [31] defined the memory layout permutation problem (MLP) and showed through exhaustive search that solving such a problem improves the offset assignment solutions but they did not present a solution to this problem.

Several others [45, 67, 70, 78, 79, 71, 26, 60, 19, 46, 29, 62] have addressed problems related to offset assignment.

3.3 The CGOA Heuristic

In this section, an effective heuristic is presented to solve the GOA problem in the presence of variable coalescing (CGOA), Figure 3.2. A set of variables such that their corresponding access graph (AG) is a line graph will have an SOA (CSOA) cost of zero since the AG in this case is acyclic and each vertex in the graph has two incident edges. To exploit this fact, our CGOA algorithm first tries to partition as many of the variables in the access sequence as possible into L sets of variables such that the access graph of each set is a line graph. Then our CGOA assigns the remaining variables into the partitions in a way to decrease the final cost.

Definition 3.1: *Define $First(v)$ to be the place in the access sequence where variable v appears the first time.*

Definition 3.2: *Define $Last(v)$ to be the place in the access sequence where variable v appears the last time*.

A set S of variables such that any two variables, $u \in S$, $v \in S$ and such that the ranges [*First(u), Last(u)*] and [*First(v), Last(v)*] do not overlap, will constitute a line access graph. First, our CGOA heuristic will compute $First(v)$ and $Last(v)$ for all variables v in the access sequence. Then it sorts the variables in increasing order of $(First + Last)$ in list $L1$ using the degree of the node in the interference graph as a tie break. Although sorting the variables in increasing order of *Last* is also a possible criteria, we found that sorting them in the increasing order of $(First + Last)$ is a better criteria to map as many variables to the available ARs as possible in this part of the heuristic.

In the first loop of our heuristic, the first unassigned variable in $L1$ will be assigned to the first address register AR_0. The heuristic will then assign to AR_0 as many variables as possible following the order in $L1$ such that no two variables assigned to AR_0 have overlapping ranges. The same procedure will be repeated for all the available address registers that we have provided that at least one variable is not assigned yet. At the end of this part of the heuristic, the CSOA cost for the sets of variables mapped to each address register is zero as the access sequence in each partition represents a line access graph and thus the CGOA cost is the address registers' initialization cost.

Since probably not all the variables will meet the requirements to be assigned to a certain address register in the first part of the heuristic, the second part will take care of assigning the rest of the variables to the address registers. For each unassigned variable, we calculate its frequency, in the original access sequence, where the frequency of a variable v is defined as the number of times

```
CGOA-ALGORITHM
------------------------------------------------------------
    Input: Access sequence AS, Interference graph IG
    Output: Offset assignment
    For each variable $v$, calculate $First(v)$ and $Last(v)$.
    $L1$= list of variables $v_i$ in increasing order of $Start(v_i)+Last(v_i)$
    While ($L1 \neq$ NULL) and ($L \leq k$)
        Pick the first variable $v$ in $L1$.
        Assign $v$ to address register $AR_L$.
        $Last = Last(v)$
        Remove $v$ from $L1$.
        f = true
        While (($L1 \neq$ NULL) and (f=true)
            Pick the first variable $u$ in $L1$ such that $Start(u) > Last$.
            if such variable $u$ is found
                $Last = Last(u)$.
                Add $u$ to $AR_L$
                Remove $u$ from $L1$.
            elseif no such variable is found
                f = false
                $L$++
        End while
    End while
    Calculate the frequency in the original access sequence for all the variables in $L1$.
    Sort variables in $L1$ in decreasing order of their frequencies.
    While ($L1 \neq$ NULL)
        Pick the first variable $u$ in $L1$.
        Calculate $Penalty_{AR_j}(u)$ for all address registers.
        Assign $u$ to the $AR$ with the smallest penalty using the
         number of interferences as the tie break.
    End While
    CGOA solution = CSOA($AS_1$)+ ... +CSOA($AS_L$)
```

FIGURE 3.2. Our general offset assignment heuristic with variable coalescing.

this variable appears in the access sequence. Then we sort the variables in decreasing order of their frequency in list $L1$. Variables with small frequency have small degree nodes in the AG. For each unassigned variable v, the *Penalty* of assigning v to AR_j is defined in Equation 3.1.

Definition 3.3: *Define the Penalty of assigning v to AR_j as the number of variables u in set $S_{AR_j}(v)$, cardinal of set S, where $S_{AR_j}(v)$ contains the variables u mapped to AR_j such that $(u,v) \in$ original access graph (AG) and such that u and v interfere.*

Since $(u,v) \in$ original access graph (AG), (u,v) will automatically be an edge in the access graph representing the variables mapped to address register AR_j if v is assigned to AR_j. *Penalty* is a measure of how far will the new access graph representing AR_j be from a line graph if the variable v is assigned to the address register AR_j considering only newly added edges whose end points interfere. Variable v will be assigned to the address register corresponding to the smallest penalty. If variable v has the same *Penalty* value corresponding to more than one AR, it will be assigned to the one with fewer number of variables that interfere with v. So the main idea in this step is to try to keep the access graph for each address register as close to a line graph as possible since line access graphs have zero CSOA costs.

$$Penalty_{AR_j}(v) = Card(S_{AR_j}(v)) \tag{3.1}$$

where

$$S_{AR_j}(v) = \{u | u \in AR_j,\ (u,v) \in IG, (u,v) \in AG\} \tag{3.2}$$

After the second while loop, we will end up with L access sequences for each of the L address registers used. Note that our heuristic may end up not using all the available address registers, and this will reduce the address registers initialization cost compared to a heuristic that uses more address registers. This is true since our CGOA tries to map to a certain address register as many variables as possible before considering other address registers.

For each address register used, the corresponding access sequence will be built from the original access sequence by considering only the variables that are mapped to this address register. And then the access graph will be constructed for each access sequence. The heuristic will then apply the CSOA algorithm in Chapter 2 to each set of variables that belongs to the same address register.

The total CSOA cost of all the partitions will be added up plus the cost of initializing the address registers used as the CGOA cost as shown in Equation 3.3 where I is the initialization cost. We assume an initialization cost of 1 in our experiments. The CGOA solution is the concatenation of the CSOA solutions.

$$CGOA\_Cost = \sum_{i=1}^{L} CSOA\_Cost_{AR_i} + L * I. \tag{3.3}$$

3.4 CGOA: Example

To clarify our CGOA algorithm, consider the example in Figure 3.3 with two available address registers $AR0$ and $AR1$. Figure 3.3(a) shows the access sequence with the access graph and the interference graph in Figure 3.3(b) and Figure 3.3(c), respectively. Our CGOA algorithm will start by calculating $First$ and $Last$ values for each variable in the access sequence, Table 3.1. Then the variables will be sorted in increasing order of $(First + Last)$ as $a\ b\ e\ c\ d$. Following the criteria described earlier in the first part of our CGOA algorithm, variables a and e will be mapped to AR_0 and variables b and d will be mapped to AR_1 since $[First(a), Last(a)] \cap [First(e), Last(e)] = \emptyset$ and $[First(b), Last(b)] \cap [First(d), Last(d)] = \emptyset$.

At the end of the first part of the heuristic, only variable c is not yet mapped to any address register. Since c is adjacent to e in the original access sequence and $(e,c) \in$ IG, $Penalty_{AR_0}(c) = 1$. Also c is adjacent to b and d in the access sequence with $(c,d) \in$ IG and $(c,b) \in$ IG, so $Penalty_{AR_1}(c) = 2$. Therefore, variable c will be mapped to AR_0. Figure 3.3(d)-(f) and Figure 3.3(g)-(i) show the access sequences for variables mapped to AR_0 and AR_1, respectively, with their corresponding access graphs and CSOA solutions. Notice that both the access graphs resulting from our heuristic are line graphs. The offset assignment for the CGOA solution is the concatenation of the offset assignments for the two CSOA solutions. The cost of our CGOA solution is the sum of the costs of the individual CSOA solutions plus the cost of 2 for initializing the two address registers used. Thus the cost of our CGOA is two since the two CSOA solutions resulted in zero cost offset assignments.

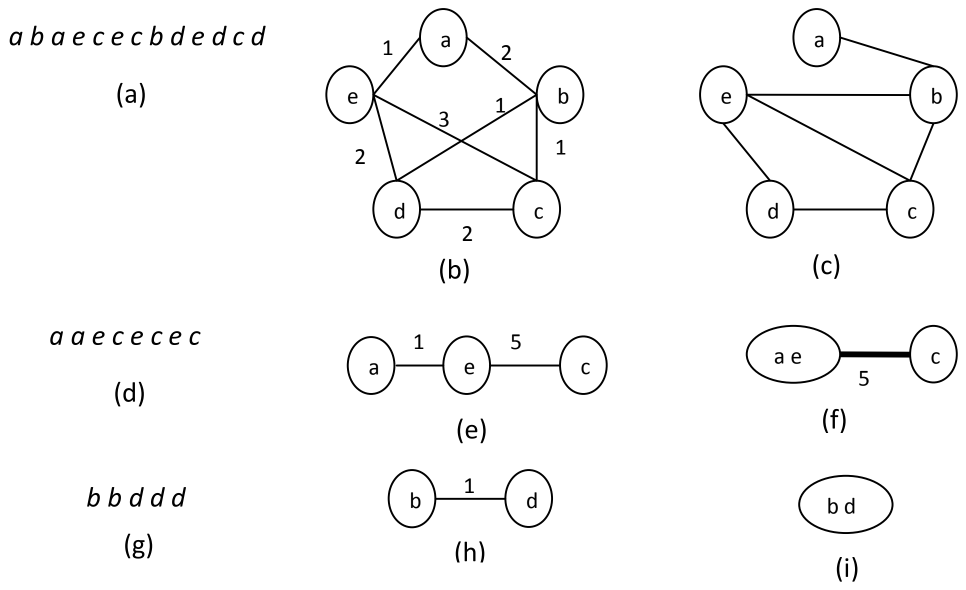


FIGURE 3.3. (a) Access sequence. (b) Original access graph. (c)Interference graph. (d)-(f) Access sequence for variables mapped to AR0, access graph, and CSOA solution of zero cost. (g)-(i) Access sequence for variables mapped to AR1, access graph, and CSOA solution of zero cost.

TABLE 3.1. *First* and *Last* values for the CGOA example in Figure 3.3.

Variable	*First*	*Last*	*First* + *Last*
a	1	3	4
b	2	8	10
c	5	12	17
d	9	13	22
e	4	10	14

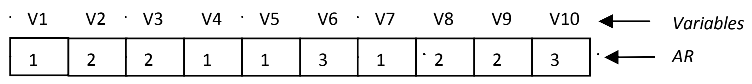


FIGURE 3.4. Solution representation for the SA

3.5 Simulated Annealing

Since the offset assignment problem is NP-complete, the heuristic presented in Section 3.3 will very likely produce a suboptimal solution. So in order to further improve the results, we used a simulated annealing approach, Figure 3.5. Simulated Annealing (SA) [35] is a global stochastic method that is used to generate approximate solutions to very large combinatorial problems. The annealing algorithm begins with an initial feasible configuration, the solution from our CGOA heuristic in our case, and then a neighbor configuration is created by perturbing the current solution.

An SA solution is represented in Figure 3.4 as a vector of n elements, where n is the number of variables in the access sequence (AS). Each variable in the AS has a fixed position in this vector with the AR assigned to this variable. A neighboring solution is selected by randomly selecting the position of variable v_i from the current configuration and changing its corresponding address register AR_i to a randomly chosen AR_j with $j < k$. The cost of a solution is calculated in the same procedure as the cost of the *CGOA* presented in Section 3.3. The cooling schedule was empirically determined as follows: 1) initial temperature = 600, 2) the temperature reduction multiplier, α , is set to 0.89, and 3) the number of iterations, M, is set to 4 while the iteration multiplier, β, is set to be 1.05. The algorithm stops when the current temperature, T, is below 0.001.

3.6 Results

We implemented our techniques in the *OffsetStone* toolset [5, 41] and we tested our heuristics on the *MediaBench* benchmarks [39]. We measured the number of explicit address arithmetic instructions needed by each method. We presented the cost resulted from the SOA-OFU in Table 3.3 as a reference point. The cost in Table 3.3 is the actual number of explicit address arithmetic instructions resulted from generating the code with the offset assignment achieved using the SOA-OFU where no variable coalescing is allowed. OFU is a naive offset assignment algorithm based on order first use where variables are assigned to offsets in the order of their appearance in the access

```
**Annealing CGOA**
$S_0$ = Initial solution.
$\alpha$ = Cooling rate.
$\beta$ = Iteration multiplier.
$T_0$ = Initial temperature.
MaxTime = Total allowed time for the annealing process.
$M_0$ = Time until next parameter update.
BestS = $S_0$
T = $T_0$
Call CGOA();
$S_0$ = Output Solution of CGOA();
CurrentS = S0
CurrentCost = CGOA_Cost(CurrentS)
BestCost = CGOA_Cost(BestS)
Time = 0
do{
    M = $M_0$
    do{
        NewS = Neighbor(CurrentS);
        NewCost=CGOA_Cost(NewS)
        $\delta$ Cost = NewCost - CurrentCost
        If ($\delta$ Cost $<$ 0)
            CurrentS=NewS
            CurrentCost=CGOA_Cost(CurrentS);
            If (NewCost $<$ BestCost) then
              BestS=NewS
              BestCost = CGOA_Cost(BestS)
        elseif (Random $< e^{-\frac{\delta Cost}{T}}$) then
            CurrentS=NewS
        CurrentCost = CGOA_Cost(CurrentS);
        M = M - 1
    } while (M $\geq$ 0)
Time = Time + M0;
T = $\alpha$ * T;
M0 = $\beta$ * M0;
while (Time $>$ MaxTime and T $>$ 0.001);
Return(BestS);
```

FIGURE 3.5. Annealing CGOA algorithm.

TABLE 3.2. Percentage of temporary variables.

Benchmarks	Temporaries (%)
adpcm	59.6
epic	48.1
g721	80.7
gsm	86.6
jpeg	65.2
mpeg2	65.6
pegwit	72.1
pgp	67.5
rasta	43.6

sequence. Smaller numbers in the tables of results are better as they mean fewer address arithmetic instructions are needed.

We compared our results to Ottoni's CGOA [58]. We did not compare our results to the CGOA in [81] since it underperforms Ottoni's CGOA as clearly shown in [58]. Recall that Ottoni's CGOA works as follows. First it sorts the variables in decreasing order of their number of interferences. Each variable is then assigned to the partition with the least number of interferences. The size of the partition is used as a tie break when a variable has the same minimum number of interferences with more than one partition. The priority is given to the partition with the fewest variables. One of the biggest drawbacks of this CGOA heuristic is that it uses more address registers than necessary. Since variables are assigned to partitions with the least corresponding number of interferences and since the size of the partition is used as a tie break where the partition with the fewest number of variables is given the highest priority, more address registers are used than necessary and thus the cost of the final solution will increase due to the address registers initialization cost.

To clarify this point, consider an example of a line access graph of *n* nodes corresponding to the *n* variables in the program. Assume that *n* address registers are available. Ottoni's CGOA heuristic will assign a variable to each address register and thus the final CGOA cost is *n* corresponding to the initialization cost of the *n* address registers. However, using only one address register which is basically applying a simple offset assignment heuristic on our line access graph will result in a final cost of 1 which is the initialization cost of the one address register used.

TABLE 3.3. SOA-OFU cost.

Benchmarks	SOA-OFU cost
adpcm	207
epic	6235
g721	718
gsm	1511
jpeg	10338
mpeg2	7981
pegwit	2249
pgp	7235
rasta	6626

This major drawback in their CGOA heuristic is more exposed when more address registers are used. This will have a great impact on our benchmarks since usually the embedded applications are divided into many basic blocks and thus the CGOA is applied separately to each basic block. Many of those basic blocks consist of a number of variables that is close to the number of available address registers. As a result Ottoni's CGOA will end up using more address registers for such basic blocks than needed. The initialization costs of the address registers used for each basic block will be added to the cost of the offset assignment solution and thus their CGOA cost will get higher with more available address registers in the system mostly due to the large initialization cost.

However, in our CGOA heuristic, fewer address registers will be used in most cases since it tries to map as many variables to a single address register as possible. To show the effectiveness of this idea, consider the results of Ottoni's CGOA and our CGOA in Tables 3.4, 3.5 and 3.6 for 2, 4, and 8 address registers, respectively. Results show that our solution outperformed that of Ottoni by a large margin. This margin increases as the number of available address registers in the system increases.

To alleviate the effect of using more address registers than necessary on the final cost of the problem, Ottoni [58] presents the CGOA results as the minimum between their CSOA and CGOA costs. This means that they need to run CSOA first to get the cost and then run CGOA. This explains the difference between the results presented in the paper [58] and the actual CGOA results presented in Tables 3.4, 3.5, and 3.6. In this way, the cost of the solution will most probably be the CSOA cost for basic blocks of small to medium number of variables. To show that the

TABLE 3.4. CGOA results for 2 ARs.

Benchmarks	Our CGOA	CGOA-Ottoni
adpcm	17	22
epic	1366	1463
g721	79	61
gsm	103	118
jpeg	1464	1659
mpeg2	1220	1412
pegwit	243	279
pgp	923	1006
rasta	1095	1381

TABLE 3.5. CGOA results for 4 ARs.

Benchmarks	Our CGOA	CGOA-Ottoni
adpcm	13	15
epic	686	769
g721	89	84
gsm	168	287
jpeg	1240	1694
mpeg2	1012	1187
pegwit	193	278
pgp	830	989
rasta	1250	1318

TABLE 3.6. CGOA results for 8 ARs.

Benchmarks	Our CGOA	CGOA-Ottoni
adpcm	21	35
epic	383	505
g721	127	195
gsm	256	619
jpeg	1477	2766
mpeg2	995	1515
pegwit	282	556
pgp	914	1622
rasta	1055	1199

TABLE 3.7. COA results for 2 ARs.

Benchmarks	COA1	COA2	COA-Ottoni
adpcm	14	15	15
epic	1334	1353	1422
g721	50	64	49
gsm	43	44	46
jpeg	1199	1251	1293
mpeg2	966	1046	1214
pegwit	211	213	243
pgp	803	825	825
rasta	586	634	690

improvement from our CGOA heuristic is not just from the reduction in the initialization cost, we tuned our CGOA heuristic to a COA heuristic, as in [58], where the final cost is not the final cost of our CGOA but rather the minimum between our CSOA and CGOA costs, $COA\_Cost = min(CSOA\_Cost, CGOA\_Cost)$.

We tested our general offset assignment heuristic with the CSOA heuristic in Chapter 2 applied to the final partitions as COA1 and with Ottoni's CSOA in [58] applied to the final partitions as COA2. Both COA1 and COA2 heuristics outperformed Ottoni's COA [58] in most cases. We used COA2 just to show the net performance of our partitioning technique compared to the one in [58]. Tables 3.7, 3.8, and 3.9 show the results for 2 ARs, 4 ARs, and 8 ARs, respectively, where the first columns show the benchmarks and the next three columns show the results for the three COA heuristics. This improvement is most probably due to the another drawback in Ottoni's COA heuristic which is that they only consider the interferences between variables as a criterion for partitioning their variables into the available address registers whereas in our case we also consider the structure of the access graph in our variable partitioning method.

Figures 3.6–3.8 show the CGOA stack savings normalized with respect to the number of variables in the program when two, four and eight address registers are used, respectively. Our CGOA reduced the memory slots needed to store the variables by 10% on average compared to that of using the CGOA algorithm in [58] for 2, 4 and 8 ARs. The stack size reduction decreases with more available address registers since partitions with fewer number of variables have less variables coalescing opportunities. Reduction in memory slots needed is essential in DSP architectures as

TABLE 3.8. COA results for 4 ARs.

Benchmarks	COA1	COA2	COA-Ottoni
adpcm	9	9	9
epic	629	648	655
g721	51	62	51
gsm	56	56	67
jpeg	837	878	951
mpeg2	726	790	806
pegwit	123	137	128
pgp	542	578	565
rasta	570	596	610

TABLE 3.9. COA results for 8 ARs.

Benchmarks	COA1	COA2	COA-Ottoni
adpcm	19	19	21
epic	324	330	343
g721	88	91	94
gsm	83	85	108
jpeg	920	961	961
mpeg2	670	695	735
pegwit	193	202	212
pgp	514	572	594
rasta	556	602	623

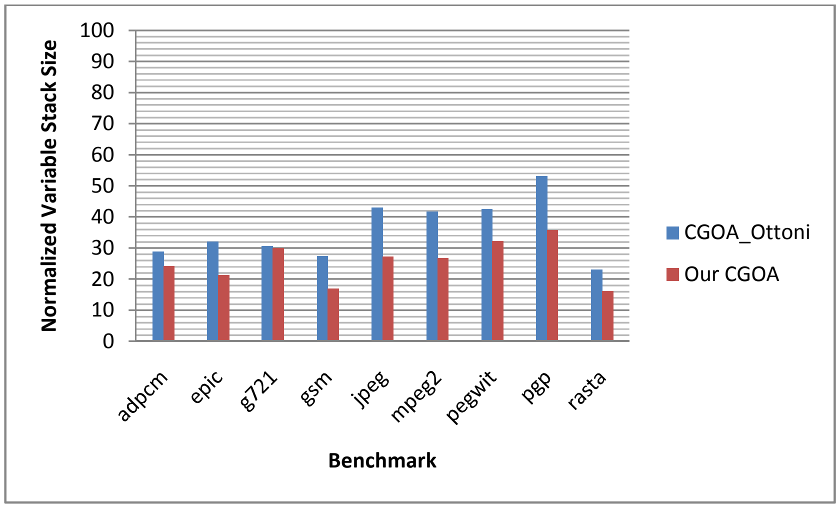


FIGURE 3.6. The normalized stack size reduction for 2 ARs using Ottoni's CGOA and our CGOA with respect to the number of variables.

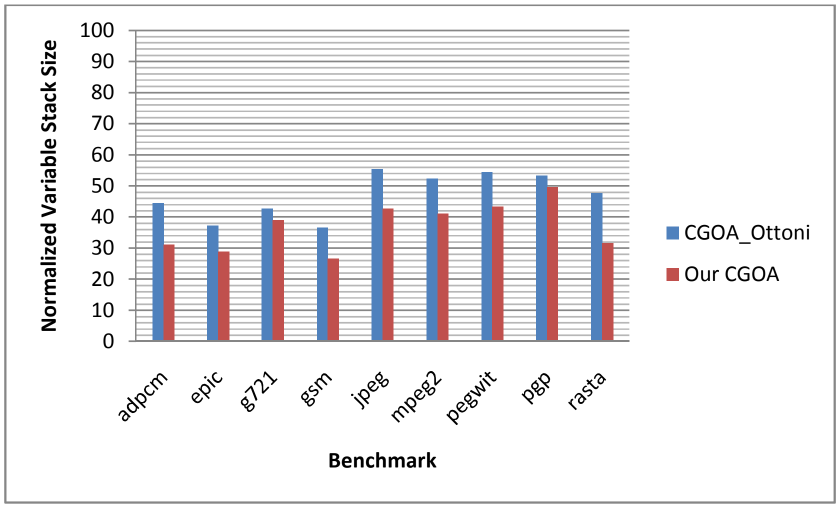


FIGURE 3.7. The normalized stack size reduction for 4 ARs using Ottoni's CGOA and our CGOA with respect to the number of variables.

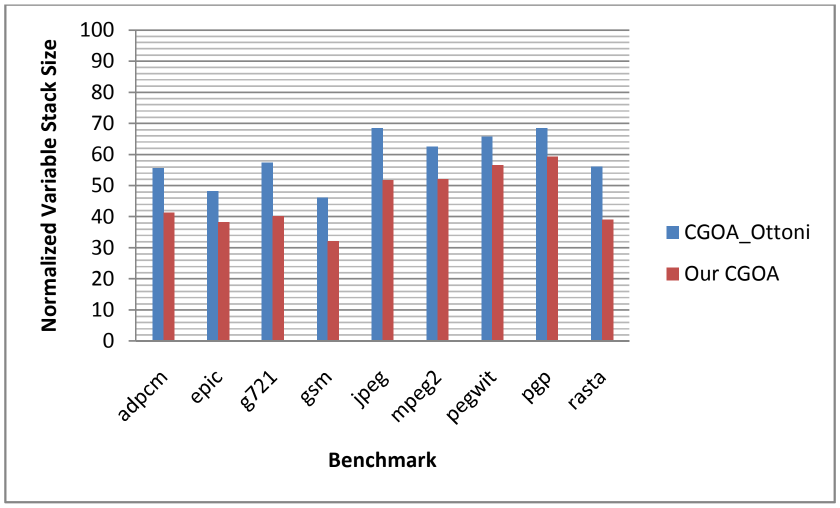


FIGURE 3.8. The normalized stack size reduction for 8 ARs using Ottoni's CGOA and our CGOA with respect to the number of variables.

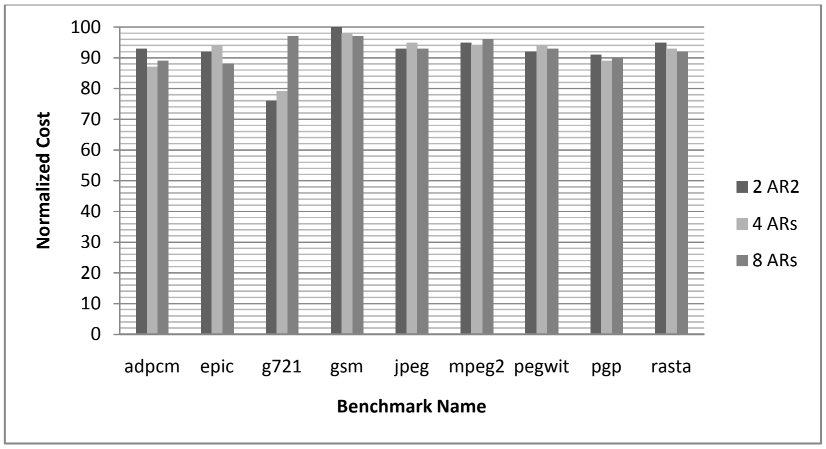


FIGURE 3.9. Normalized cost for SA with respect to our CGOA 2, 4, and 8 ARs

this means less memory is needed to store the variables in the application. This reduction from our techniques results from coalescing more variables which means that more variables share the same memory slot. Although not studied in this paper, one can expect a similar reduction in the power consumption due to the smaller code size and fewer execution cycles.

Finally, we tested all the benchmarks using our SA. Due to the big benchmarks that we used, the simulated algorithm takes a long time to converge into a high quality solution. So to speed up SA, we started it with our CGOA solution as the initial solution and we ran it for a maximum of 15 minutes. Figure 3.9 shows the normalized cost of our SA with respect to our CGOA which shows a cost reduction of 0% to 12% compared to our CGOA and thus it shows that there is still room for improvement.

The cost reduction in our techniques is essential for DSPs since the code in such systems resides in the ROM and thus it directly translates into silicon area. Our techniques reduced the code size by reducing the number of explicit address arithmetic instructions as well as the variable stack size through variable coalescing. The techniques presented in this paper outperformed the best known solutions in the literature [58]. This is very significant as address arithmetic instructions are sometimes up to 50% of the code size in such DSP systems. Those improvements are basically possible due to the large number of temporary variables in DSP applications where a variable is considered temporary if it is alive in only one basic block, Table 3.2.

3.7 Chapter Summary

In this chapter, we presented a heuristic to solve the general offset assignment problem with variable coalescing where more than one variable can be mapped to the same memory location. Results on different benchmarks show the effectiveness of our heuristic compared to other heuristics. Results were further improved using simulated annealing.

Chapter 4
The Offset Assignment Problem with Variable Permutation

In Chapters 2 and 3, we presented two effective heuristics for the simple and the general offset assignment problem with variable coalescing. However, we assumed that the access sequence is fixed. In this chapter, we will assume that the access sequence is not fixed but rather variables can be permuted in the case of permutative operations. For instance $a = b + c$ can be rewritten as $a = c + b$ in our approach provided that this improves the final solution. We will present an optimal integer linear programming (ILP) formulation for the simple offset assignment and the general offset assignment with variable permutation.

One of the main assumptions endorsed in the literature and in Chapter 3 for the general offset assignment problem is that a variable can be accessed by only one address register throughout the program run. In this chapter, we will formulate the CGOA problem assuming different instances of the same variable can be accessed by different address registers. The optimal solution to our new approach to the general offset assignment is at least as good as that of the traditional CGOA as the latter is a special case of the former.

4.1 CSOA ILP Formulation

For the simple offset assignment problem with variable coalescing, we formulate the ILP based on the access graph and thus the solution is the best cover of the access graph so that the sum of the weights of the uncovered edges is minimized. The idea is to find the best offset assignment that is the best placement of the variables in the memory so that the cost is minimized. Recall that two variables with non-overlapping live ranges can share the same memory location. This variable coalescing is intended to increase the proximity between the variables in the memory and thus decrease the number of explicit address arithmetic instructions in the code generated for a DSP architecture with an address generation unit.

First define the binary variable X_i^l that takes the value 1 if the variable i is at position l in the memory offset assignment.

$$X_i^l = \begin{cases} 1, \text{ if variable } i \text{ is in position } l \text{ in the OA} \\ 0, \text{ otherwise} \end{cases} \tag{4.1}$$

A certain variable i can be mapped to one and only one location l in the memory, Equation (4.2). For the simple offset assignment with variable coalescing, more than one variable can be mapped to the same memory location. Two variables can be mapped to the same memory location if they do not interfere, that is, there is no edge in the interference graph between those two variables. Thus in Equation (4.3), two variables that interfere are not allowed to share the same memory location.

$$\sum_l X_i^l = 1 \qquad \forall\, i \tag{4.2}$$

$$X_i^l + X_j^l \leq 1 \;\; \forall\, (i,j) \in \textit{Interference graph} \tag{4.3}$$

There is no need for an explicit address arithmetic instruction to update the address register pointing to variable location i to point to variable location j if those two variables are in adjacent locations, $X_i^l + X_j^{l+1} = 2$, or if they are coalesced, that is, they share the same memory location, $X_i^l + X_j^l = 2$. Define the binary variable Y_{ij} as a variable that takes a value of 1 if variables i and j are within the auto-modify range,

$$Y_{ij} = \begin{cases} 1, \text{ if } X_i^l + X_j^{l+1} = 2 \text{ or } X_i^l + X_j^l = 2 \\ 0, \text{ otherwise} \end{cases} \tag{4.4}$$

To force variable Y_{ij} to be 1 only when variables (X_i^l&& X_j^{l+1}) or (X_i^l && X_j^l) are equal to 1, we have to include the constraints in Equations (4.5)-(4.7). In that set of constraints, for any two locations l and l' in the memory such that those two locations are within the auto-modify range, the value of the binary variable Y_{ij} will be 1 which means that there is no need for an address arithmetic instruction. Since in our case we assumed an auto-modify range of [-1,1] then variable Y_{ij} will take the value of 1 if locations l and l' are adjacent or are the same location that is l = l'. Note that those constraints can be easily modified to accommodate any auto-modify range. We need the constraints in Equations (4.5)-(4.7) for all values of $|l\text{-}l'| \leq 1$ since we defined the variable

Y as Y_{ij} rather than Y^l_{ij} that is variable Y has no index for the memory position.

$$\forall\ i,\ j,\ l \text{ and } l' \text{ such that } |l\text{-}l'| \leq 1 :$$

$$Y_{ij} \leq X^l_i \tag{4.5}$$

$$Y_{ij} \leq X^{l'}_j + X^l_j \tag{4.6}$$

$$Y_{ij} \geq X^l_i + X^l_j + X^{l'}_j - 1 \tag{4.7}$$

We define the overall cost of our CSOA solution as the sum of the weights of the edges in the access graph that are not selected as those edges represent the explicit address arithmetic instructions needed. Our objective function is used as the sum of the weights of the selected edges in the access graph and thus we need to maximize this function as more selected edges results in smaller number of address arithmetic instructions in the generated code and thus smaller code size. The objective function is as defined in Equation (4.8).

$$Maximize : \sum_i \sum_i w_{ij}(Y_{ij} + Y_{ji}) \tag{4.8}$$

4.2 ILP Formulation with Variable Permutation

Most of the previous research on the offset assignment problem was concerned about finding good heuristics for the SOA and the GOA problems assuming that the access sequence (AS) order is fixed. A solution to the offset assignment problem greatly depends on the AS. A slight change in the access sequence may change the structure of the access graph (AG). As the structure of the access graph is the core for the quality and the cost of the offset assignment solution, trying to find a new feasible AG to further improve the OA solution becomes an important issue. The basic idea is to try to change the positions of the variable instances in the AS in a way so that the final cost is decreased. For instance, statement $a = b + c$ is equivalent to the statement $a = c + b$ as the addition operation is permutative. An example of a non-permutative operation is the division operation. Although a minus operation is not commutative, but it is permutative as $a - b$ is equivalent to $-b + a$. However, we allow such permutation between variables around a minus operation as long as the first variable in the resultant statement is not in the form of - x. An example of an allowable permutation in the presence of a minus operation is $a + b - c \rightarrow a - c + b$ but not $-c + a + b$.

Permuting the operators of a permutative operation changes the position of appearances of the variables in the AS and hence we end up with a new AS and consequently a new offset assignment problem. Variables and statements can be moved around as long as the new code is equivalent to the original code, that is, for a set of input values, both the codes will result in the same output values.

In this section, we formulate the simple offset assignment problem with the inclusion of variable permutation. The solution looks for the best permutation of variables in the right hand side of the statements to decrease the cost.

For instance the possible permutations for the statement $a = b + c + d$ and the corresponding access sequences are:

1. $a = b + d + c$ AS: $b\ d\ c\ a$
2. $a = c + d + b$ AS: $c\ d\ b\ a$
3. $a = c + b + d$ AS: $c\ b\ d\ a$
4. $a = d + b + c$ AS: $d\ b\ c\ a$
5. $a = d + c + b$ AS: $d\ c\ b\ a$

Many permutations were possible since all of the operations in that statement are addition operations which is a permutative operation. Now consider the following statement with a subtraction operation which is a non permutative operation $a = b - c + d$. The only possible permutations allowed in our formulation for that statement are: $a = d + b - c$, $a = d - c + b$ and $a = b + d - c$.

To understand the effectiveness of variable permutation in decreasing the offset assignment cost, consider the example in Figure 4.1. Figure 4.1 (a) shows the original code with the corresponding access sequence in Figure 4.1 (b). Applying Liao's SOA on the access graph in Figure 4.1 (c) will result in an offset assignment as shown in Figure 4.1 (d) with a cost of 3 which represents the two address arithmetic instructions needed to move the address register between variables a and c as well as one address arithmetic instruction needed to update the address register pointing to variable c in the memory to point to variable d.

Now consider an equivalent version of the code with variable permutation in Figure 4.1 (e). The codes in Figures 4.1 (a) and (e) are equivalent. The only difference between those two versions is the position of variables in the access sequence and thus they will result in two completely different access graphs as shown in Figures 4.1 (c) and (g). Again applying Liao's offset assignment to the access graph in Figure 4.1 (g) will result in a zero-cost offset assignment solution, Figure 4.1 (h), compared to a cost of three resulting by applying the SOA heuristic to the AG corresponding to the original code.

This simple example clearly shows that permuting the variables can be an effective technique to further decrease the offset assignment cost which is the number of explicit address arithmetic instructions. Thus applying variable permutation will further improve the code generated for embedded applications on a DSP architecture by reducing the code size.

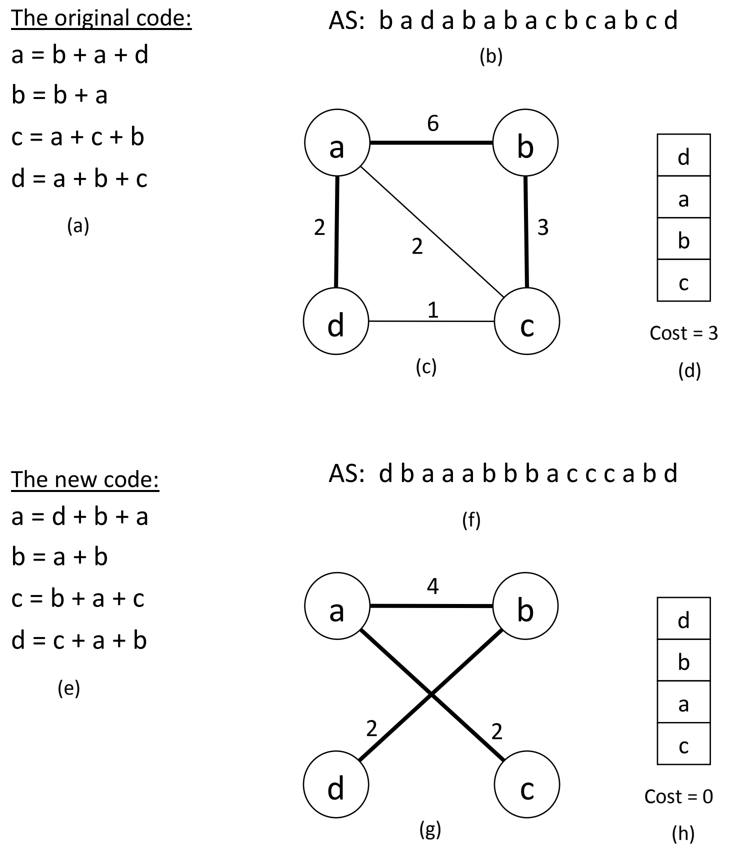


FIGURE 4.1. (a) The original code. The corresponding AS (b), access graph (c), and offset assignment (d). (e) The code after permutation. The corresponding AS (f), access graph (g), and offset assignment (h).

Recall that the cost function for simple offset assignment with variable coalescing was in the form of $w_{ij} \cdot Y_{ij}$. In that case the weights of the edges are known since the access sequence is fixed. However, in the case of variable permutation, the access sequence is not fixed and consequently the access graph and thus the weights of the edges in the access graph are not fixed. To include this into our ILP formulation, define the binary variable W_{ijs} as below. For instance, consider the statement $S1 : a = b + c$. W_{ac1} = 1, but if variables b and c are permuted, then W_{ac1} = 0.

$$W_{ijs} = \begin{cases} 1, \text{ if variables } i \text{ and } j \text{ are next to each other in the statement's } S \text{ AS} \\ 0, \text{ otherwise} \end{cases} \tag{4.9}$$

To include variable permutation for permutative operations in our formulation, a few things should be taken into consideration. Consider for instance the statement $s : a = b + c + d$. The right hand side of s has three variables with permutative operations. Only two variables can have two neighbors considering only statement s. Usually for n variables with permutative operations in a certain statement, n-1 variables will end up with two neighbors locally in that statement. Another important aspect is that only one variable i in the RHS can be the neighbor of the variable in the LHS of the statement s. One more constraint needed is that the variable in the LHS of statement $s-1$ that precedes statement s in the code must have an edge to only one of the variables j in the RHS of s. Notice that, for a statement s with more than one variable in the RHS, $i \neq j$. The following equations are sufficient to take care of the number of neighbors of each variable in a certain statement. For each statement s with a as the variable in the LHS and with b as the LHS variable of statement $s-1$:

$$\sum_i \sum_j W_{ijs} = n - 1 \qquad \forall s \tag{4.10}$$

$$\sum_{i \in RHS(s)} W_{ais} = 1 \qquad a = LHS(s-1) \tag{4.11}$$

$$\sum_{i \in RHS(s)} W_{gis-1} = 1 \qquad g = LHS(s) \tag{4.12}$$

$$\sum_{i \in RHS(s)} W_{ijs} = 2 - W_{ajs} - W_{gjs-1} \qquad \forall j, s \tag{4.13}$$

The objective function in this case is the same as in the CSOA. However, since W and Y are both variables, the objective function in Equation (4.14) is not linear and thus we resort to linearization.

$$\sum_s \sum_i \sum_i W_{ijs}(Y_{ij} + Y_{ji}) + \sum_s \sum_{j \in s} W_{ajs}(Y_{ij} + Y_{ji}) \qquad a \in LHS(s-1) \tag{4.14}$$

To do so, define the binary variable R_{ijs} as follows: the value of 1 if Y_{ij} + Y_{ji} = 1 & W_{ijs} = 1,

$$R_{ijs} = \begin{cases} 1, \text{ if } Y_{ij} + Y_{ji} = 1 \ \& \ W_{ijs} = 1 \\ 0, \text{ otherwise} \end{cases} \tag{4.15}$$

$$(R_{ijs} \leq Y_{ij} + Y_{ji}) \qquad \forall i, j, s \tag{4.16}$$

$$(R_{ijs} \leq W_{ijs}) \qquad \forall i, j, s \tag{4.17}$$

$$(R_{ijs} \geq Y_{ij} + Y_{ji} + W_{ijs} - 1) \qquad \forall i, j, s \tag{4.18}$$

The objective function can now be expressed linearly as shown below where the first part represents the number of explicit address arithmetic instructions saved between variables of the same statement whereas the second part takes care of the variables in statement s and the variable a in the left hand side of statement $s-1$ which is the statement that precedes statement s in the code:

$$Maximize \sum_{s}\sum_{i}\sum_{i} R_{ijs} + \sum_{s}\sum_{j \in s} R_{ajs} \qquad a \in LHS(s-1) \tag{4.19}$$

Note that in this subsection, we considered variable permutation without variable coalescing as the permutation can change the interference graph as will be shown in the next section. The ILP formulation for the case of variable coalescing and permutation is presented in the next section.

4.3 General Offset Assignment with Variable Coalescing

4.3.1 Problem Definition

The general offset assignment problem (GOA) refers to the case when there are more than one address register. Traditionally, GOA solutions are based on variable partitioning among the available address registers. In all of the heuristics (to the best of our knowledge), a variable can be accessed by only one address register.

To clarify the traditional solution to the GOA problem, consider the example in Figure 4.2 with two available address registers AR_0 and AR_1. Figure 4.2(a) shows the original access sequence. Assume that variables a, b, c and d are mapped to AR_0 and variables e and f are mapped to $AR1$. This variable partitioning is the optimal solution for this access sequence. Then two access sequences are extracted from the original access sequence. The first one represents the sequence of variables mapped to AR_0 and the second represents the sequence of variables mapped to address register AR_1. Figure 4.2(b)-(c) shows that access sequence for the variables mapped to address

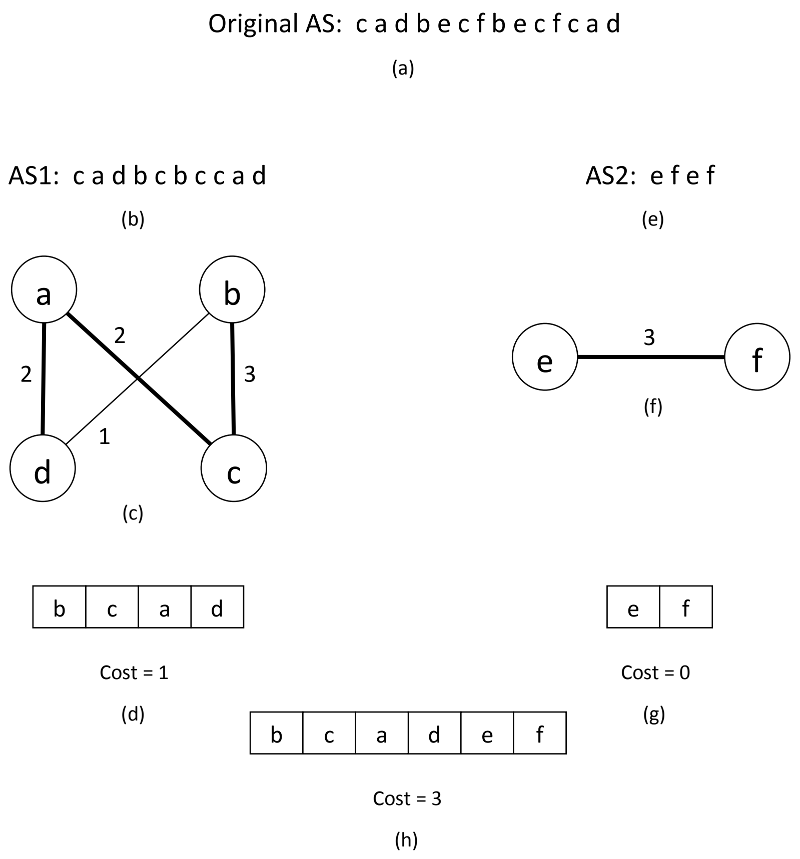


FIGURE 4.2. (a) Original access sequence. (b) Access sequence for AR_0. (c) Access graph for AR_0. (d) Offset assignment for AR_0. (b) Access sequence for AR_1. (c) Access graph for AR_1. (d) Offset assignment for AR_1. (e) Offset assignment for the GOA problem.

register AR_0 with the corresponding access graph. Applying SOA to the access graph in Figure 4.2 (b) results in an offset assignment, Figure 4.2 (d), of cost = 1 which represents the one explicit address arithmetic instruction needed to update AR_0 pointing to variable b at a certain program point to point to the address of variable d. Similarly, Figures 4.2(e),(f) and (g) show the access sequence, access graph and offset assignment for the variables mapped to address register AR_1 with a cost of zero. The GOA solution, Figure 4.2 (h), is the concatenation of the SOA solutions with the final cost of 3 which represents the sum of the costs of the SOA solutions plus an initialization cost of 2 for the two used address registers in the generated code.

In the literature, GOA is based on the assumption that all the instances of a certain variable are accessed by only one address register and thus the solution is usually based on dividing the variables into partitions and then mapping each partition into an address register. However, this assumption is used to simplify the problem but it may degrade the quality of the final solution as the optimal solution to this problem may not be the solution with minimum number of explicit address arithmetic instructions for the problem in hand. An alternative solution to the general offset assignment problem is to allow a variable to be accessed by more than one address register. As a result, our CGOA solution is based on the idea of partitioning the variable access sequence rather than the variables.

The new statement of the general offset assignment problem can now be defined as follows.

Problem Definition: *Given an access sequence AS, partition AS into l ($l \leq k$) partitions where different instances of the same variable can be mapped into different partitions so that the number of explicit address arithmetic instructions is minimized.*

Recall that the solution of the traditional general offset assignment problem is based on the concatenation of the solutions of the simple offset assignment applied to each partition. This is possible because there are no variables in common between the access sequences of different address registers since the solution is based on variable partitioning. However, the idea of partitioning the variables' instances in the access sequence rather than the variables may result in access sequences with common variables.

Consider for instance the example AS in Figure 4.3 (a) with two available *ARs* which is the same example used in Figure 4.2. Partitioning the access sequence into two partitions in Figures 4.3 (b) and (d) results in two access sequences with variables a and d in common. Applying SOA to the corresponding access graphs in Figures 4.3 (c) and (e) is not feasible as it will result in an offset assignment that needs to duplicate the common variables between the two access variables in the memory. To take care of this, the solution of our new proposed general offset assignment problem is reached as follows.

1. Divide the original access sequence in the best possible way into l partitions, Figures 4.3(b),(d).
2. Build the access graph corresponding to each access sequence, Figures 4.3 (c),(e).
3. Merge the resultant access graphs into one access graph, Figures 4.3 (f).
4. Apply SOA to the resultant access graph.

5. The GOA solution is the offset assignment of the solution in Step 4, Figure 4.3 (g).

6. The GOA cost is the SOA cost in Step 4 plus the initialization cost of each address register used.

The merge operator in Step 3 is performed as follows. Assume the original access sequence is partitioned into two access sequences with the corresponding access graphs $AG_1(V_1, E_1)$ and $AG_2(V_2, E_2)$. Assume AG is the access graph resulting from merging AG_1 and AG_2. Then $AG(V,E) = \{V, E| \ V = V_1 \cup V_2 \ and \ E = E_1 \cup E_2\}$. Merging the access graphs in Figures 4.3(c) and (e) results in the access graph in Figure 4.3(f). Applying SOA to the AG in Figure 4.3(f) will result in a zero cost offset assignment and thus the GOA cost is equal to 2 which is the initialization cost of the two address registers used. Notice that this cost is less than the cost of 3 when the traditional GOA is applied to the same problem in Figure 4.2. Note that the optimal cost of this new CGOA is at least as good as the traditional CGOA as the latter is a special case of the former.

4.3.2 CGOA ILP Formulation with Variable Permutation

Trying to formulate the new nontraditional CGOA problem defined in Section 4.3.1 based on building the access graphs like in the CSOA case is quite expensive especially with the inclusion of variable permutation. Thus we formulate the general offset assignment problem with variable coalescing based on the cost formulated in [43] that does not work on the access graph but rather directly on the access sequence.

The inclusion of variable permutation for permutative operations means that the position of a variable in an access sequence may not be static. A variable's position can change as permutation is applied. Based on the permutativity of the operations in a certain statement, a variable can be positioned at different places. Consider for instance the following statement: $a = b + c + d$. The position of the variable a in the left hand side is static since the position of the statement is assumed static in our case. Assume that the position of variable a in the access sequence is p. The positions of variables b, c and d can be in the range of $[p-3, p[$ since addition is permutative and thus the statement can be written as:

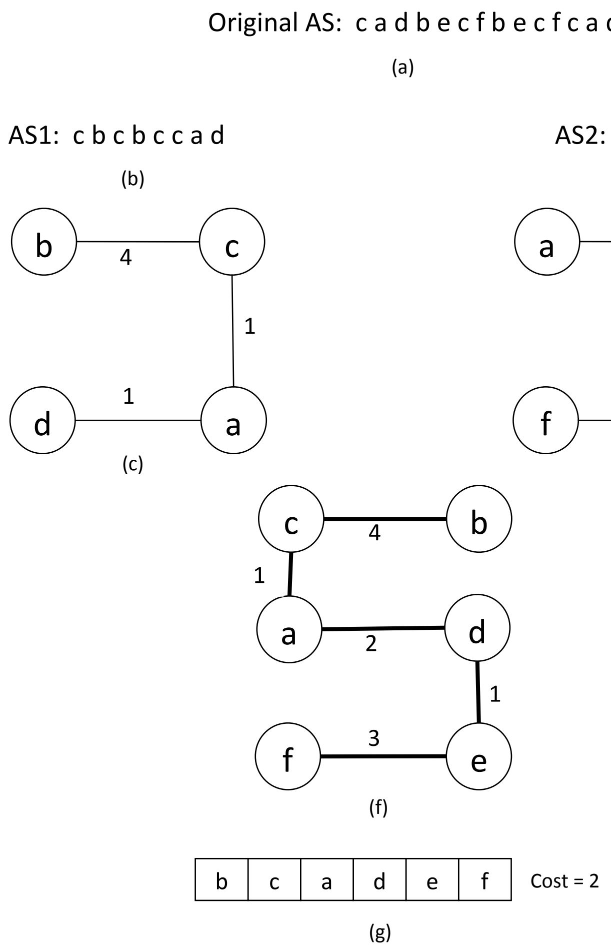


FIGURE 4.3. (a) Original access sequence. (b) Access sequence for AR_0. (c) Access graph for AR_0. (d) Access sequence for AR_1. (e) Access graph for AR_1. (f) The resultant access graph from the merge operator. (e) Offset assignment for the GOA problem.

1. $a = b + d + c$
2. $a = c + b + d$
3. $a = c + d + b$
4. $a = d + c + b$
5. $a = d + b + c$

Define the binary variable $P_{ixx'}$ as a binary variable that keeps track of the position of the variable i initially positioned at x in the access sequence before any permutation is applied. The legal positions of a certain variable, say in the range $[p, p+n]$, are extracted by the compiler based on the permutativity of the operation and thus the $P_{ixx'}$ can be 1 only for $x' \in [p, p+n]$. Equation (4.21) ensures that $P_{ixx'} = 0$ for all $x' \notin [p, p+n]$.

$$P_{ixx'} = \begin{cases} 1, \text{ if variable } i \text{ intially positioned at } x \text{ is repositioned to } x' \text{ in the AS.} \\ 0, \text{ otherwise} \end{cases} \tag{4.20}$$

$\forall\ x' \notin$ *[p,p+n]* where *[p,p+n]* is the legal range of positions of variable i :

$$P_{ixx'} = 0 \tag{4.21}$$

Since only one variable i can be at a certain position x' in the access sequence and a variable i initially positioned at x in the access sequence can be repositioned to at most one position x', the following constraints are needed:

$\forall\ x' \in [1, m]$ *where m is the size of the access sequence*:

$$\sum_{\textit{i positioned at x such that i is legal to be in x'}} P_{ixx'} = 1 \tag{4.22}$$

And $\forall$ *i positioned at x that is legal to be positioned at x'* $\in$ *[p,p+n]* :

$$\sum_{x'} P_{ixx'} = 1 \tag{4.23}$$

Define the binary variable $R_{kix'}$ which keeps track of the variable that address register k points to at program point x'. Equation (4.25) takes care of an important aspect in our formulation which is that a variable register has to point to one variable at a program point if this address register is used. If the address register is not used then it should not point to any variables at any point in the program.

$$R_{kix'} = \begin{cases} 1, \text{ if address register } k \text{ points to variable } i \text{ at program point } x' \\ 0, \text{ otherwise} \end{cases} \tag{4.24}$$

$$\sum_i R_{kix'} = AR_k \qquad \forall\ k,\ x' \tag{4.25}$$

We define the binary variable $I_{x'k}$ to take a value 1 if there is a need for an explicit address arithmetic instruction by the address register k at program point x' as follows.

$$I_{x'k} = \begin{cases} 1, \text{ if condition 1} \\ 0, \text{ otherwise} \end{cases} \tag{4.26}$$

where condition 1 = if variable at position x', $P_{ixx'}$, is covered by address register k and the variable at position $x'-1$, $P_{ixx'-1}$, is covered by address register k and the variables are within an auto-modify range (Y_{ij} = 1).

The constraint in Equation (4.27) basically keeps track of the explicit address arithmetic instructions needed at each program point. $I_{x'k}$ will take a value of one at program point x' if the variables pointed to by address register k at points x' and $x'-1$ are not within the auto-modify range and thus an explicit load is needed. Note that $I_{x'k}$ will take the smallest possible value since the objective function discussed later on is to minimize the sum of all Is. Also $I_{x'k}$ cannot take a negative value as it is defined as a binary variable which means that the only possible values that I can take are either 0 or 1.

$$I_{x'k} \geq R_{kix'} + P_{ixx'} + R_{kjx'-1} + P_{jyx'-1} - Y_{ij} - 3 \qquad \forall\ ix,\ jy,\ x',\ k \tag{4.27}$$

The binary variable Y_{ij} in the expression below is the same binary variable defined for the case of CSOA and is redefined below as

$$Y_{ij} = \begin{cases} 1, \text{ if } X_i^l + X_j^{l+1} = 2 \text{ or } X_i^l. + X_j^l = 2 \\ 0, \text{ otherwise} \end{cases} \tag{4.28}$$

$$\forall\ i,\ j,\ l \text{ and } l' \text{ such that } |l\text{-}l'| \leq 1 : \tag{4.29}$$

$$Y_{ij} \leq X_i^l \tag{4.30}$$

$$Y_{ij} \leq X_j^{l'} + X_j^l \tag{4.31}$$

$$Y_{ij} \geq X_i^l + X_j^l + X_j^{l'} - 1 \tag{4.32}$$

A very important idea when formulating the offset assignment problem with variable coalescing and variable permutation is that the permutation may alter the interference graph. Recall that in the interference graph there is a node for each variable and an edge between two variables meaning that their live ranges overlap and thus coalescing is not possible. Consider the two equivalent versions of the piece of code below.

Version 1:	Version 2:
S1: $c = a + c$	S1: $c = a + c$
S2: $a = b + d + c$	S2: $a = c + b + d$
S3: $b = a + d$	S3: $b = a + d$

For Version 1, assume that the variable b in statement S2 is the first appearance of b in the program and the appearance of variable c in S2 is the last in the program. The live ranges of variables b and c overlap and thus they cannot be coalesced. Now consider the same code in Version 2 after applying permutation to the variables in the RHS of statement S2 of Version 1. This permutation makes variables b and c interference free as their live ranges do not overlap any more. Notice that this can happen only when applying permutation to variables i and j in a certain statement and such that it is the first appearance of variable i and the last appearance of the variable j. To take this into consideration, we divide the edges in the interference graph (IG) into three types:

1. *Type 1*: Edges in the IG that cannot be deleted by applying variable permutation.
2. *Type 2*: Edges in the IG that are possible to be deleted by variable permutation.
3. *Type 3*: Pairs (i, j) that should be added as edges to the IG if those two variables were interference free and now interfere after applying permutation.

Recall that in the case of variable coalescing with no variable permutation in Section 4.1, the constraint to take care of variables interference is as follows,

$$X_i^l + X_j^l \leq 1 \;\; \forall \textit{ (i,j)} \in \textit{Interference graph.} \tag{4.33}$$

This simple formulation is no more sufficient in the case of variable permutation as the IG is not fixed any more. That is, some of the edges in the IG may get deleted as discussed earlier and some edges can be added to the IG.

For the *Type 1* edges in the IG, the constraint in Equation (4.33) is sufficient and thus the constraint is now expressed as follows,

$$X_i^l + X_j^l \leq 1 \;\; \forall \textit{ Type1 edges (i,j)} \in \textit{Interference graph.} \tag{4.34}$$

The formulation gets more complicated to take care of the other types of edges in the IG. The formulation is only for the statements that have a variable i that appears the last time and a variable j that appears the first time as in only this case can applying permutation alter the IG. The current position x' in the AS of the variable i of legal range $[p, p+n]$ originally positioned at x can be expressed as: $p1 = \sum_{x' \in [p,p+n]} x' \cdot P_{ixx'}$ where as the current position y' in the AS of the variable j of legal range $[p', p'+n']$ originally positioned at y can be expressed as: $p2 = \sum_{y' \in [p',p'+n']} y' \cdot P_{jyy'}$. Now for each pair (i, j) that respects the condition mentioned earlier, if $p1 > p2$, then i and j interfere; otherwise they are interference free.

For all *Type 2* and *Type 3* pairs of variables (i, j), the constraint is now expressed in Equation (4.35) where the expression $(2 - \frac{p1-p2}{p1})$ is < 2 if $p1 > p2$ and > 2 if $p1 < p2$,

$$X_i^l + X_j^l \leq 2 - \frac{p1 - p2}{p1} \;\; \forall \textit{ Type 2 or Type 3 (i,j).} \tag{4.35}$$

Sometimes using more address registers for a certain CGOA problem can result in a higher cost due to the initialization cost for using an address register which is basically an extra instruction in the generated code. So at this point, we need to keep track of how many address registers are used and thus add an initialization cost of 1 for each address register. Define the binary variable AR_k as:

$$AR_k = \begin{cases} 1, \text{ if address register } k \text{ is used} \\ 0, \text{ otherwise} \end{cases} \tag{4.36}$$

To ensure that variable AR_k is 1 when the address register k is used, the constraint in Equation (4.37) is needed. AR_k will take a value of 1 if at no program point there was a variable instance accessed by address register k. If $R_{kix'}$ = 0 at all program instances, then address register k is not used and thus $AR_k = 0$ since AR_k is a binary variable that can take values 0 or 1 and the sum $\sum_k AR_k$ is minimized in the objective function, Equation (4.38). And on the other side, if $R_{kix'} = 1$ at any program point meaning that address register k is used, then AR_k in the constraint below will be greater than 1 but since AR_k is binary variable, AR_k will take the value 1.

$$AR_k \geq R_{kix'} \qquad \forall\ k,\ i,\ x' \tag{4.37}$$

We define the overall cost of our CGOA solution in Equation (4.38) as the number of explicit address arithmetic instructions needed by each address register which is basically the sum of all the I variables at all the program points for all the address registers used plus the initialization cost of each address register used.

$$Minimize \sum_k \sum_{x'} I_{kx'} + \sum_k AR_k \tag{4.38}$$

Another important aspect of the offset assignment problem with variable coalescing is the resulting ability to decrease the memory needed to store the variables. Coalescing increases the proximity between the variables and thus more variables tend to share the same memory location. As the average number of variables mapped to each memory location is increased, the memory stack size needed to store the program variables is decreased. Thus another objective function for the problem can also be stated as to decrease the memory stack size. To formulate this idea into a linear form, define the binary variable m_l that takes a value of 1 if at least a variable is mapped to the memory location l as

$$m_l \geq X_i^l \qquad \forall\ l,\ i. \tag{4.39}$$

The objective now is to minimize the number of memory locations used and thus *minimize* $\sum_l m_l$.

4.4 Results

We implemented our techniques in the *OffsetStone* toolset [5, 41] and we tested our heuristics on the *MediaBench* benchmarks [39]. We measured the percentage of the number of address arithmetic instructions inserted by each method with respect to the number of address arithmetic instructions inserted by the simple and general offset assignment heuristics with variable coalescing presented by Ottoni [58]. Those two heuristics are well known effective heuristics that solve the offset assignment problem in the case of variable coalescing. Variable permutation is not used in Ottoni's case as the access sequence is assumed to be fixed.

The methodology of our experimental evaluation is as follows. Given a benchmark, extract the original access sequence assuming that the statements' variable order is fixed, that is, before any variable permutation is applied. Based on the permutativity of the operations available in the statements, we find the range of positions at which a certain variable can be placed in the access sequences. Two variables with a non permutative operation are looked at as a single variable in our techniques. For instance, consider the statement $a = b + \frac{c}{d}$. Variables c and d are around a non-permutative operation and thus those variables cannot be permuted. So we can look at $\frac{c}{d}$ as a single variable C and thus the statement can be looked at as $a = b + C$ and thus now variables b and C can be permuted due to the permutativity of the addition operation and thus it can be written as $a = C + d$ which is a simpler presentation of $a = \frac{c}{d} + b$. The output will be the range of positions that a variable can take in the access sequence. Those ranges are to be used by our ILP formulation.

We implemented four different versions of our integer linear formulations using CPLEX mainly:

- *CSOA*: Integer linear programming of the simple offset assignment problem with variable coalescing.
- *CSOA_Perm*: Integer linear programming of the simple offset assignment problem with variable coalescing with the inclusion of variable permutation for permutative operations.
- *CGOA*: Integer linear programming of the general offset assignment problem with variable coalescing.
- *CGOA_Perm*: Integer linear programming of the general offset assignment problem with variable coalescing with the inclusion of variable permutation.

For the first part of our experimental results, we show the effectiveness of our ILP formulation for the simple offset assignment with variable coalescing. For this simple offset assignment, variable permutations are applied to find the best possible access sequence that results in the best

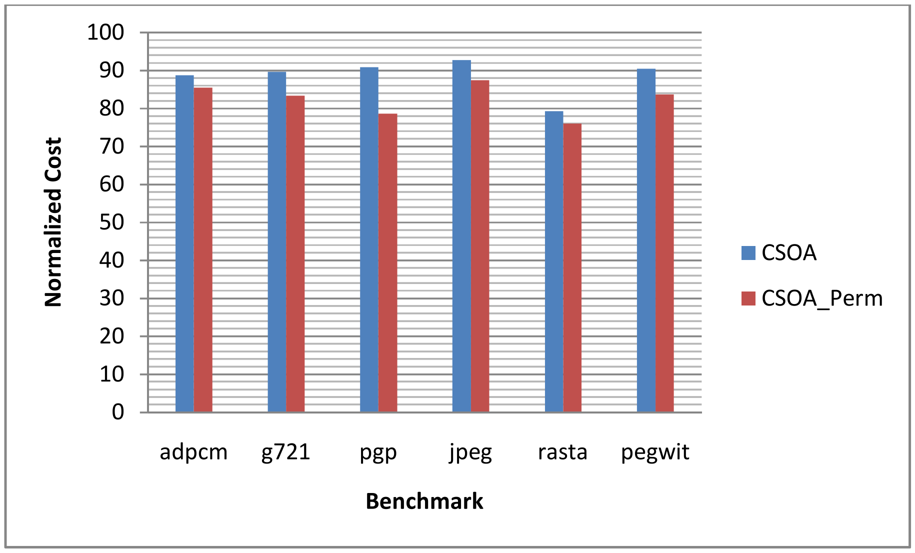


FIGURE 4.4. The normalized cost for the CSOA problem with permutation.

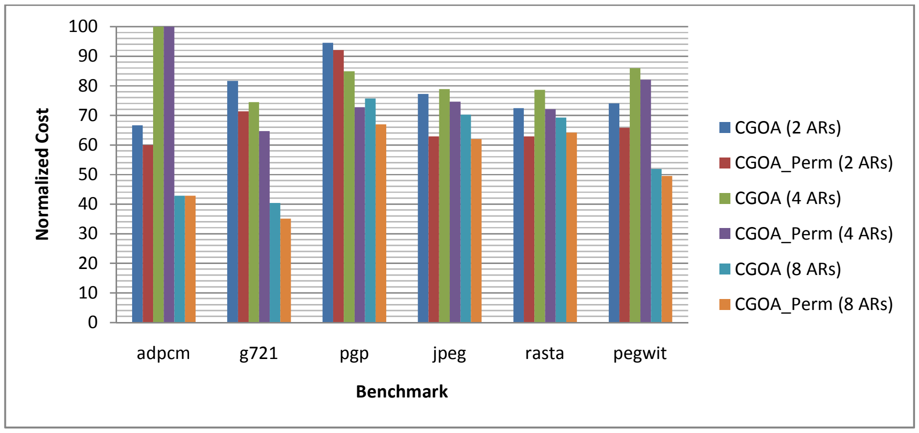


FIGURE 4.5. The normalized cost for the CGOA problem with permutation.

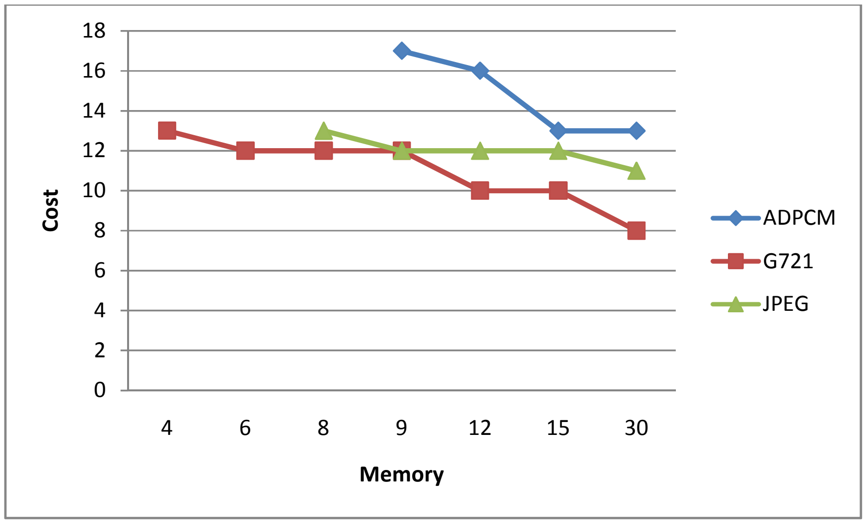


FIGURE 4.6. Design space exploration of memory versus cost.

cost. The cost in this part is the number of explicit address arithmetic instructions needed in the code generated for a DSP architecture with an address generation unit. The auto-modify range is assumed to be [-1,1] in our experimental results but our techniques can handle any auto-modify range. The benchmarks used are made of basic blocks. We apply our formulations to each basic block separately. We limit our experiments on basic blocks of number of variables less than or equal to 30 since as the number of variables increase, the run time for our ILP increases exponentially.

For the CSOA, we tested our ILP formulations against the CSOA heuristic in [58]. Results in Figure 4.4 are the cost (number of explicit address arithmetic instructions) normalized against the cost from the CSOA heuristic. Results show that our CSOA ILP formulation improved over the CSOA heuristic by 12% on average whereas the inclusion of variable permutation improved over the heuristic by 18% on average.

Then we tested our CGOA ILP techniques and presented the results as the cost normalized against the CGOA heuristic in [58]. We show the results for 2, 4, and 8 address registers (AR). Our CGOA optimal ILP formulation improved over the heuristic by 22%, 17%, and 42% on average for 2, 4, and 8 ARs, respectively, whereas the CGOA formulation with variable permutation improved over the heuristic by 31%, 23%, and 46% on average for 2, 4, and 8 ARs, respectively, Figure 4.5.

Results show that with more address registers, the improvements from our ILP solution over-performs the CGOA heuristic by a big margin. The reason is that the heuristic may use more address registers than needed and thus the initialization cost of the ARs used for each basic block will be added up to the cost of the final solution. This is not the case for the ILP optimal solution since the solution for l available address registers for each basic block is the minimum cost for all the solutions for any number of address registers $\leq l$ as our ILP will find the optimal solution that may be a solution using k address registers where $k \leq l$. Note that some of the improvement is due to the new approach to the CGOA presented in Section 4 where we partitioned the variable instances rather than variables.

For this part of the experiments, we tested our ILP *CGOA* with the objective to decrease the memory requirement for the program variables (stack size). Coalescing more variables reduces the memory stack size. Our optimal results improved on average over the Ottoni's offset assignment heuristics by 25%, 28%, 33%, and 41% for 1, 2, 4, and 8 address registers, respectively. Reducing the variable stack size is very important as the available memory is usually limited in such DSP systems.

In this part of our experiments, we perform design space exploration to find the best cost for an available number of address registers. This will be helpful in the embedded system design. We do so by setting the objective function to minimize the stack size, in Section 4.3, as a constraint in the form $\sum_l m_l <$ memory size m. We start with m equals to the number of variables so that is there is a memory location for each variable in the program which is the case of no variable coalescing. If it is feasible to find a solution to this problem then we try half the size of the memory $m/2$ and if the solution with memory size of $m/2$ is not feasible, then we try $\left\lceil \frac{m+m/2}{2} \right\rceil$. We always try half of the range up or down between the memory size explored at this point and the previous memory size explored. This will be very helpful for the design space exploration.

Figure 4.6 shows the design space exploration for the basic blocks of 30 variables extracted from *ADPCM*, *G*721, and *JPEG* benchmarks with 4 available address registers. This will be helpful for the designer to find the best memory versus cost parameters. Notice that with less memory, the ILP is forced to coalesce more variables and in many cases heavily variable coalescing increases the cost as that may prevent some select opportunities in the offset assignment solution methodology.

The cost reduction in our techniques is essential for DSPs since the code in such systems resides in the ROM and thus it directly translates into silicon area. The number of address arithmetic instructions is sometimes up to 50% of the code size in some DSP applications. Our ILP techniques reduced the code size by reducing the number of explicit address arithmetic instructions as well as the variable stack size through variable coalescing. The techniques presented in this chapter outperformed the best known solutions in the literature [58]. Although ILP formulations are more expensive than heuristics in terms of solution time, the cost reduction is very essential and significant for such DSP systems and this makes long solution times bearable.

4.5 Chapter Summary

Reducing the code size of an embedded application in a tightly constrained DSP architecture is crucial. Thus, an optimal solution is favorable compared to heuristics even though it needs more computation time. The problem of offset assignment has received a lot of attention from researchers due to its great impact on code size reduction for DSPs. Reducing the code size is beneficial in the case of DSPs since the code is directly transformed into silicon area. In this chapter, optimal ILP formulations for the CSOA and CGOA with variable permutation are presented. Also a new proposed approach to the general offset assignment was presented. Results on different benchmarks show the big improvement possible from the optimal solution as well as from the variable permutation compared to other heuristics in the literature.

Chapter 5
Address Register Allocation for Arrays in Loops

In the previous three chapters, we studied the problem of offset assignment with variable coalescing as a technique to reduce the code size and memory requirement for scalar-based embedded applications. In this chapter, we are concerned with array-intensive embedded applications. The offset assignment techniques are not applicable to the case of arrays. The main difference with arrays is that the array element locations are fixed and cannot be rearranged and hence the necessity of a completely different approach to reduce the code size and memory requirement for array-intensive applications.

Many DSP algorithms have an iterative pattern of references to array elements within loops. In this chapter, we study the address register allocation for array references in array-intensive DSP applications. Given an array-intensive DSP application, the problem is to assign the array references to the available address registers (ARs) so that auto-increment/auto-decrement mode is maximally utilized. Proper assignment of array references to ARs will reduce the number of explicit address arithmetic instructions and thus the code size. DSP applications are known to have up to 50% address arithmetic instructions (20% to 30% in most cases) [75]. Thus there is significant potential for code size reduction, which is essential for digital signal processors. Code size reduction often leads to execution cycle reduction and energy reduction. Due to the large benefit from solving the address register allocation problem, finding an optimal or near-optimal solution is favorable to finding solutions using heuristics, regardless of the overhead in the computation time of finding such solutions.

5.1 Problem Definition and Related Work

Array references in many DSP applications usually have small constant strides and thus they observe high locality. As a result, an AGU can be highly utilized to maximally exploit auto-increment and decrement, which obviates the need for explicit address arithmetic instructions. For instance, if two consecutive array elements say $X[i]$ and $X[i+1]$ are mapped to the same address register, then auto-increment can be used to update the address register pointing to $X[i]$ to point to $X[i+1]$. The address register problem can be defined as follows.

Problem Definition: *Given an array reference sequence, access sequence, and a set of available address registers (ARs), map each reference to an address register such that the number of address arithmetic instructions is minimized.*

Since the number of ARs is usually small compared to the number of array references, the address register allocation can be a difficult problem. Consider the following array reference example to illustrate the problem [42].

```
for (i = 2;  i < M; i++){
    X[i+1]      //r1
    X[i]        //r2
    X[i+2]      //r3
    X[i-1]      //r4
    X[i+1]      //r5
    X[i]        //r6
    X[i-2]      //r7
    }
```

Assume that we have two available address registers $AR0$ and $AR1$. If references $r1$, $r2$, and $r3$ are assigned to $AR0$ and the rest of the references are assigned to $AR1$, then the corresponding assembly code contains 4 explicit address arithmetic instructions (shown under Case(a) below). In contrast, with two ARs, if $r1$, $r2$, $r3$, $r5$ and $r6$ are assigned to address register $AR0$, and the rest to $AR1$, the assembly code has 3 explicit address arithmetic instructions (shown under Case(b) below). Below are the corresponding codes where the instructions in bold represent the explicit address arithmetic instructions. Note that in the code below we only show the address arithmetic instructions and operations. This simple example shows that careful assignment of the array references to the available address registers reduces the address arithmetic instructions.

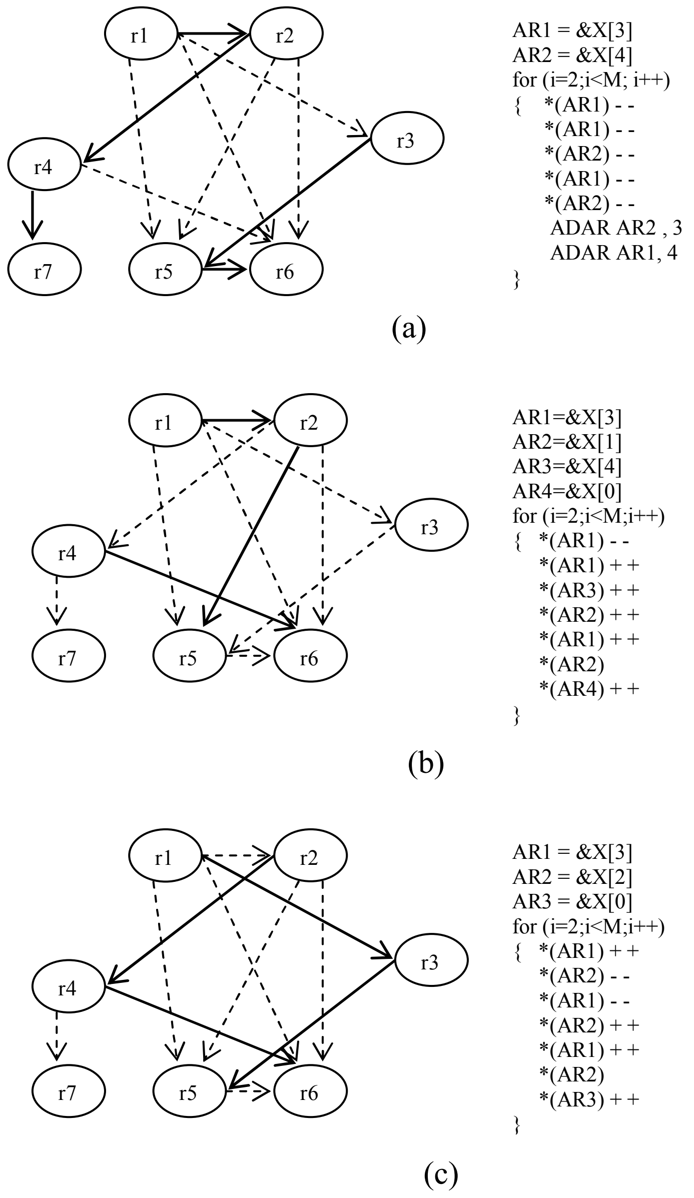
```
 Case (a)                 Case (b)

LDAR AR0, &X[3]          LDAR AR0, &X[3]
LDAR AR1, &X[1]          LDAR AR1, &X[1]
```

```
for (i = 2;  i < M; i++)          for (i = 2; i < M; i++)
       *(AR0)- -                          *(AR0)- -
       ADAR AR0,2                         ADAR AR0,2
       *(AR0)                             *(AR0)- -
       ADAR AR1,2                         *(AR1)- -
       *(AR1)- -                          *(AR0)- -
       SBAR AR1,2                         ADAR AR0,2
       ADAR AR1,2                         ADAR AR1,2
```

Several researchers have studied the address register allocation problem. Gebotys [26] described a technique based on the minimum network flow circulation problem [74] to minimize the cost of merging paths together. Araujo and Malik [7] introduced the indexing graph (IG) to represent the address register allocation problem. Each array reference is represented by a vertex in the IG and an edge between two vertices means that the index distance between the corresponding array references is within the auto-modify range, for simplicity [-1,1] in our case. Thus an edge represents the possible transition from one array reference to another without the need of an address instruction. They mapped the problem to determining the disjoint path/cycle cover of the IG which minimizes the total number of paths and cycles and which has the smallest number of paths, and showed that this problem is similar to the problem of finding the minimum disjoint cycle cover of a graph (MDCC).

Since MDCC is NP-complete, the authors proposed the *simple-IG* which is an acyclic graph (see Figure 5.1 for an example) that is derived by dropping all the back edges (from the corresponding IG), where a *back edge* is an edge between two array references across loop iterations. An example of a *back edge* in our sample code is the edge between the array reference $r5$ of the current iteration (i.e., $X[i+1]$) and the array reference $r4$ of the next iteration (i.e., $X[(i+1)-1]$), since these references are within the auto-modify range. Using the simple-IG reduces the problem to the minimum disjoint path cover (MDPC) problem. This problem was studied before by Boesch and Gimpel [16] based on the Hopcroft-Karp [30] algorithm for a maximum bipartite matching. The main idea was to transform the simple-IG into a bipartite graph by splitting each vertex v into two vertices $v1$ and $v2$ where vertex $v1$ (resp. $v2$) is the source (resp. destination) of all outgoing (resp. incoming) edges from (resp. into) v. However, the simple-IG covering problem does not eliminate the need for explicit address arithmetic instructions in the loop body (for back edges).



```
AR1 = &X[3]
AR2 = &X[4]
for (i=2;i<M; i++)
{   *(AR1) - -
    *(AR1) - -
    *(AR2) - -
    *(AR1) - -
    *(AR2) - -
     ADAR AR2 , 3
     ADAR AR1, 4
}
```

```
AR1=&X[3]
AR2=&X[1]
AR3=&X[4]
AR4=&X[0]
for (i=2;i<M;i++)
{  *(AR1) - -
   *(AR1) + +
   *(AR3) + +
   *(AR2) + +
   *(AR1) + +
   *(AR2)
   *(AR4) + +
}
```

```
AR1 = &X[3]
AR2 = &X[2]
AR3 = &X[0]
for (i=2;i<M;i++)
{  *(AR1) + +
   *(AR2) - -
   *(AR1) - -
   *(AR2) + +
   *(AR1) + +
   *(AR2)
   *(AR3) + +
}
```

FIGURE 5.1. (a) Match-based algorithm solution. (b) Path-based algorithm solution. (c) ILP optimal solution. (IG + corresponding assembly code)

Leupers et al. [42] proposed a path-based algorithm to find an upper bound on the number of address registers. They introduced an extended distance graph where a node for each array reference in the next loop iteration is added to the indexing graph. They allocated the address registers in a greedy fashion based on a longest path heuristic. Their heuristic outputs an upper bound on the number of address registers needed to incur a zero cost solution. They enhanced their algorithm with a merge operator to merge paths if the path-based algorithm results in a number of ARs that exceeds the number of available ARs [14]. Ottoni et al. [60, 59] presented an efficient way for array reference allocation for loops in embedded systems. Recently, Chen et al. [18] studied the effect of transformations to reduce the cost.

When the matching-based algorithm in [7] is applied to the piece of code presented earlier in this section, we end up with two address registers. In this solution, two address arithmetic instructions are needed at the end of each iteration as shown in Figure 5.1-(a). The path-based algorithm [42] produces a zero cost solution with four ARs as shown in Figure 5.1-(b). However, the optimal solution is a zero-cost solution with only three address registers as seen in Figure 5.1-(c). Note that even though our solution proposed in this chapter works on the IG, for simplicity we show the result on the simple-IG in Figure 5.1-(c). Solid edges in all sub-figures of Figure 5.1 refer to selected edges whereas dashed edges are not selected. Similarly, a path or a cycle cover represents the references covered by the same address register. For instance, references $r1$, $r3$, and $r5$ in Figure 5.1-(c) are addressed using address register $AR1$ due to the path cover (*r1, r3, r5*) and references $r2$, $r4$, and $r6$ are covered by $AR2$ whereas reference $r7$ is covered by $AR3$.

5.2 ILP Formulation of the Address Register Allocation Problem

In this section, we will optimally solve two versions of the address register allocation problem namely, (i) minimum cost for a given number of address registers, and (ii) minimum number of ARs for a zero cost cover. We will optimally solve these two problems using integer linear programming (ILP). Due to the great benefits resulting from optimally solving these two problems, the run time overhead by the ILP is bearable.

5.2.1 Minimum Cost for a Given Number of ARs

The first ILP formulation finds the minimum number of explicit address arithmetic instructions for a given number of ARs. This problem is defined as follows.

Problem Definition: *Given an indexing graph (IG) and the number of address registers (ARs), find the best mapping of the array references to the ARs such that the total number of explicit address arithmetic instructions is minimized.*

This problem is similar to the problem of finding the minimum disjoint cycle cover of a graph (MDCC) which is an NP-hard problem. The MDCC of a graph G is the minimum number of disjoint cycles that cover all the vertices.

Definition 5.1: *A back edge (ri, rj) is an edge across iterations, where reference ri is a reference in the current iteration, reference rj is in the next iteration, and $i > j$.*

Definition 5.2: *A forward edge (ri, rj) is an edge between reference ri in the loop body to a later reference rj in the same iteration where $i < j$.*

Note that our formulation works on the IG whereas the solutions in [42, 14, 7] work on the simple-IG, that is, the IG without any *back edges*; therefore, these works [42, 14, 7] solve a simpler (and clearly different) version of the problem with a solution that is usually worse than the solution to the original problem (on the IG).

The objective is to be able to assign the IG vertices to those ARs such that the number of address arithmetic instructions is minimized. This is an important problem since the number of available address registers is usually limited. For this problem we need to extend the IG to also include edges (u,v) such that $|W(u,v)| >$ the auto-modify range (which is 1 in our case). We call such edges *cost-inducing* edges.

Definition 5.3: *A cost-inducing edge is an edge between two array references that are not within the auto-modify range.*

To develop the ILP formulation, we first define the following binary variables.

- AR_k is a binary variable for address register k such that:

$$AR_k = \begin{cases} 1, \text{ if address register } k \text{ is used} \\ 0, \text{ otherwise} \end{cases} \tag{5.1}$$

- Y_{ijk} is a binary variable that denotes if the edge $(i,j) \in$ IG is covered by address register k and is defined as follows:

$$Y_{ijk} = \begin{cases} 1, \text{ if } (i,j) \text{ is covered by } AR_k \\ 0, \text{ otherwise} \end{cases} \tag{5.2}$$

- X_{ik} is a binary variable that denotes if vertex i in the IG is covered by address register k and is defined as follows:

$$X_{ik} = \begin{cases} 1, \text{ if vertex } i \text{ is covered by } AR_k \\ 0, \text{ otherwise} \end{cases} \tag{5.3}$$

- Z_{ijk} is a binary varliable for each *cost-inducing* edge (i, j) in the extended IG and is defined as follows:

$$Z_{ijk} = \begin{cases} 1, \text{ if } (i,j) \text{ is covered by } AR_k \\ 0, \text{ otherwise} \end{cases} \tag{5.4}$$

The objective of this problem is to minimize the number of *cost-inducing* edges used in the IG cover and the number of ARs used. Our objective function decreases the number of *cost-inducing* edges since a *cost-inducing* edge is an edge between two references that are not within the auto-modify range and thus an explicit address arithmetic instruction is always needed if that edge is mapped to a certain address register. The minimization in our objective function for the address registers will minimize the number of AR initialization instructions. The objective function is defined as follows.

$$Minimize : \alpha \sum_i \sum_j \sum_k Z_{ijk} + \sum_{l=0}^{k} AR_l, \tag{5.5}$$

where the actual cost is just the number of costly edges represented by the first part of the objective function divided by α. The second part of the objective function makes sure that we do not use more address registers than needed to get this cost. But since the main objective is to minimize the number of *cost-inducing* edges, the first part of the objective function is given more weight through the constant α, where α is a big number.

Here are the list of constraints needed in our ILP formulation.

Constraint 1: Each vertex in an IG is covered by only one address register:

$$\sum_k X_{ik} = 1 \qquad \forall\, i. \tag{5.6}$$

Constraint 2: In this constraint we ensure that if an address register k is not used (AR_k=0), then no vertex or edge is covered by this address register.

$$X_{ik} - AR_k \leq 0 \qquad \forall\, i,k \tag{5.7}$$

$$Y_{ijk} - AR_k \leq 0 \qquad \forall\, i,j,k \tag{5.8}$$

$$Z_{ijk} - AR_k \leq 0 \qquad \forall\, i,j,k \tag{5.9}$$

Constraint 3: If an edge (i, j) is covered by address register k then both vertices i and j are covered by the same address register k. This is formulated as follows.

$$X_{ik} + X_{jk} \geq 2Y_{ijk} \qquad \forall i,j,k \tag{5.10}$$

$$X_{ik} + X_{jk} \geq 2Z_{ijk} \qquad \forall i,j,k \tag{5.11}$$

Constraint 4: This constraint ensures that each vertex in the IG has one covered ingoing edge and one covered outgoing edge.

$$\sum_i \sum_k Y_{ijk} + \sum_i \sum_k Z_{ijk} = 1 \qquad \forall j \tag{5.12}$$

$$\sum_j \sum_k Y_{ijk} + \sum_j \sum_k Z_{ijk} = 1 \qquad \forall i \tag{5.13}$$

Constraint 5: Constraint 5 makes sure that a cycle cover can contain only one back edge (or self-edge). We used the equality condition here rather than the inequality ($\leq$) to ensure that each cover is a cycle with a back edge that can be a *cost-inducing* or a regular edge (a non-*cost-inducing* edge). But since the objective function is minimizing the number of *cost-inducing* edges, this constraint maximizes the inclusion of regular back edges.

$$\sum_j \sum_{i \leq j} Y_{jik} + \sum_j \sum_{i \leq j} Z_{jik} = AR_k \qquad \forall k \tag{5.14}$$

Constraint 6: This constraint ensures that all the vertices and the edges that are covered by the same AR are connected in a legal way that maintains a feasible solution. This makes sure that the number of vertices in an IG that are covered by the address register AR_k is equal to the number of edges covered by AR_k.

$$\sum_i X_{ik} = \sum_i \sum_j Y_{ijk} + \sum_i \sum_j Z_{ijk} \qquad \forall k \tag{5.15}$$

5.2.2 Minimum Number of ARs for a Zero Cost Cover

The second ILP formulation finds the minimum number of ARs for a zero-cost solution. Such a solution is desired since the speed penalty of each address computation is multiplied by the number of loop iterations which is usually large. Since the objective is a cover with zero cost, only cycles are allowed where a node by itself is considered as a self-cycle. A path cover in an IG is not allowed in this case since an explicit address arithmetic instruction may be needed to update the address register pointing to the array reference represented in the tail of the path to point to the array reference represented in the head of the path in the next iteration. So our problem now can be defined as follows:

Problem Definition: *Given an IG, find the minimum number of cycles that cover the IG such that each cycle contains only one back edge.*

The formulation of such a problem is a simpler version of the previous formulation as it works on the IG and not the extended IG where the extended IG is the IG with the costly edges. As a result all the Z binary variables will be dropped from our formulation. Since the number of ARs is not known in this problem, an upper bound on the number of ARs will be used in the objective function based on the upper bound solution presented in [42].

Objective function : The objective of this problem is to minimize the number of address registers needed to cover the indexing graph such that the cost is zero. The objective function that takes care of this is defined as follows:

$$Minimize : \sum_{k=0}^{u} AR_k, \tag{5.16}$$

where u is the upper bound on the number of address registers needed, computed using the path-based algorithm [42].

Following is the list of the constraints that are equivalent to those in Section 5.2.1.

Constraint 1:

$$\sum_{k} X_{ik} = 1 \qquad \forall\, i \tag{5.17}$$

Constraint 2:

$$X_{ik} - AR_k \leq 0 \qquad \forall\, i,k \tag{5.18}$$

$$Y_{ijk} - AR_k \leq 0 \qquad \forall\, i,j,k \tag{5.19}$$

Constraint 3:

$$X_{ik} + X_{jk} \geq 2Y_{ijk} \qquad \forall i,j,k \tag{5.20}$$

Constraint 4:

$$\sum_i \sum_k Y_{ijk} = 1 \qquad \forall j \tag{5.21}$$

$$\sum_j \sum_k Y_{ijk} = 1 \qquad \forall i \tag{5.22}$$

Constraint 5:

$$\sum_j \sum_{i \leq j} Y_{jik} = AR_k \qquad \forall k \tag{5.23}$$

Constraint 6:

$$\sum_i X_{ik} = \sum_i \sum_j Y_{ijk} \qquad \forall k \tag{5.24}$$

5.2.3 Code Restructuring

In the previous formulation, we assumed that the order of the array references is fixed. In this section, we study the effect of code restructuring (i.e., code transformations) on reducing the number of address arithmetic instructions as well as the number of address registers used. The restructuring technique studied in this section consists of reordering the array references inside a statement as well as reordering the statements inside the loop body. The statement reordering technique is valid if it does not violate the dependences inside the loop body or the operation's correctness.

In order to be able to formulate the above problem as an ILP, we need to update the indexing graph in a way to reflect the potential legal reordering possibilities as well as the benefits of such reordering. In order to take care of this, we introduce the *reordering* edge $e_{i,j}$ between two array reference i and j of the same array, say X_k, indicating that there is no legality violation that prevents

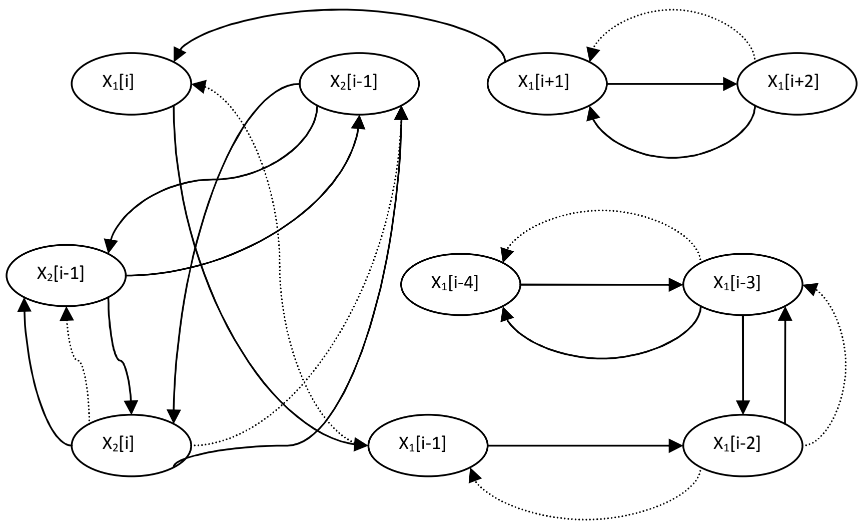


FIGURE 5.2. An example of an indexing graph with reordering forward edges (dash edges)

the array reference j to be scheduled after the array reference i and that j and i are within the auto-modify range. Figure 5.2 shows the updated indexing graph for the example code below where the reordering edges are shown as dashed edges (*backward reordering* edges are not shown for simplicity).

```
          for ($i\ =\ 4;\ i\ <\ M;\ i$++){
$S_1$ :          $X_1[i] = X_2[i-1] + X_1[i+1] + X_1[i+2]$
$S_2$ :          $X_2[i-1] = X_1[i-4] + X_1[i-3]$
$S_3$ :          $X_2[i] = X_1[i-1] + X_1[i-2]$}
```

To clarify the edges in Figure 5.2, consider the three edges between the two references $X_1[i+1]$ and $X_1[i+2]$.

1. The edge $(X_1[i+1], X_1[i+2])$ is a forward edge.
2. The solid edge $(X_1[i+2], X_1[i+1])$ is a back edge.
3. The dashed edge $(X_1[i+2], X_1[i+1])$ is a reordering edge which means that it is legal for reference $X_1[i+2]$ to be accessed before reference $X_1[i+1]$.

Below is the code after applying code restructuring. Applying the formulation in the previous subsection will show that the cost goes down from two explicit address arithmetic instructions with 3 ARs before code restructuring to one with 2 ARs after code restructuring with the assembly code for those two codes are respectively shown below. The code only shows the address arithmetic instructions and operations. The instructions in bold are the explicit address arithmetic instructions.

```
for ($i = 4$; $i < M$; $i$++){
    $S_1 : X_1[i] = X_2[i-1] + X_1[i+2] + X_1[i+1]$
    $S_3 : X_2[i] = X_1[i-1] + X_1[i-2]$
    $S_2 : X_2[i-1] = X_1[i-3] + X_1[i-4]$}
```

```
Before code restructuring:              After code restructuring:
LDAR AR0, &X2[3]                          LDAR AR0, &X2[3]
LDAR AR1, &X1[5]                          LDAR AR1, &X1[6]
LDAR AR2, &X1[0]
for (i = 4;  i < M; i++)                for (i = 4; i < M; i++)
        *(AR0)                                  *(AR0)++
        *(AR1)++                                *(AR1)- -
        SBAR AR1,2                              *(AR1)- -
        *(AR1)- -                               *(AR1)- -
        *(AR2)- -                               *(AR1)- -
        *(AR2)- -                               *(AR1)- -
        *(AR0)++                                *(AR0)- -
        ADAR AR1,3                              *(AR1)- -
        *(AR2)- -                               ADAR AR1,7
        *(AR0)                                  *(AR0)++
```

To include the reordering techniques in our formulation, additional constraints are needed to make sure that the final solution is a feasible one. First, we define the binary variable P_{ij} for every pair of statements S_i and S_j in the loop body as a variable that keeps track of the position of statement S_i with respect to that of statement S_j.

$$P_{ij} = \begin{cases} 1, \text{ if } S_i \text{ occurs before } S_j \text{ in the loop body} \\ 0, \text{ otherwise} \end{cases} \tag{5.25}$$

Constraint 1: In this constraint we make sure that if statement S_j precedes statement S_i in our final solution, then none of the forward edges (including the reordering forward edge) from S_i to S_j can be selected since those edges are no longer feasible forward edges. Also this constraint takes care of the requirement that if any forward edge Y_{ij} is selected, where i and j are array references in statements S_i and S_j, respectively, then statement S_i precedes statement S_j in the loop body.

$$Y_{ij} - P_{ij} \leq 0 \qquad \forall i, j \text{ such that } i \in S_i \text{ and } j \in S_j \tag{5.26}$$

<u>*Constraint 2:*</u> If statement S_i precedes S_j then at least one of the reordering forward edges (if any) from S_i to S_j is selected.

$$\sum_{i \in S_i, j \in S_j, (i,j) \text{ is reordering forward}} Y_{ij} - P_{ij} \geq 0 \tag{5.27}$$

<u>*Constraint 3:*</u> If there is a loop-independent dependence between statements S_i and S_j (i.e., a dependence from S_i in a loop iteration to S_j in the same loop iteration), then statement S_i must precede statement S_j after applying the reordering techniques. For every pair of such statements we should have the following constraint:

$$P_{ij} = 1. \tag{5.28}$$

<u>*Constraint 4:*</u> This constraint takes care of the feasibility of the solution in terms of making sure that the proper ordering of the statements in the loop body is legal. It basically states that if statement S_i precedes statement S_j and statement S_j precedes statement S_k, then statement S_i precedes statement S_k in the loop body. The constraints in Equations (5.29) and (5.30) are very important as they make sure that the proper positioning of the statements is feasible. Note that in our formulation we do not have a variable that keeps track of the position of each statement in the loop body but rather we have variable P_{ij} that reflects the position of the statement S_i with respect to the statement S_j.

Equation (5.30) takes care of an important aspect in our formulation which is to make sure that if a reordering edge is selected going from a statement S_{k2} to statement S_{k1}, then no forward edge from S_{k1} to S_{k2} can be selected. Such forward edges are no longer valid as they violate the definition of a forward edge which is an edge from an array reference to a later array reference in the same loop body.

$$P_{ij} + P_{jk} \leq 2P_{ik} \qquad \forall \text{ statements } S_i, S_j, S_k \tag{5.29}$$

$$P_{ij} + P_{ji} = 1 \qquad \forall \text{ statements } S_i, S_j \tag{5.30}$$

5.3 Genetic Algorithm

The ILP formulations presented in Section 5.2 guarantee optimal solutions but with high execution time for large applications. So in order to get near-optimal solutions for large embedded applica-

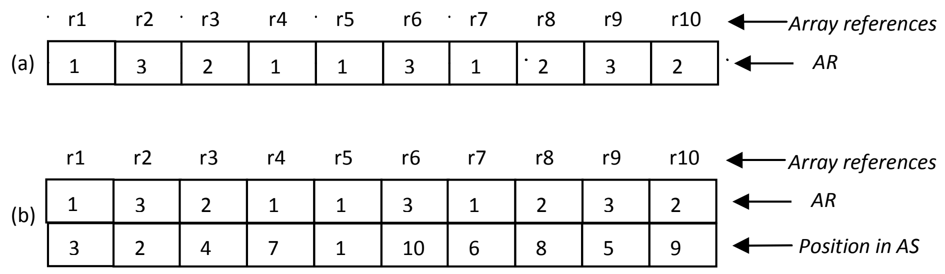


FIGURE 5.3. (a) Chromosome representation for the GA. (b) Chromosome representation for the GA with code restructuring.

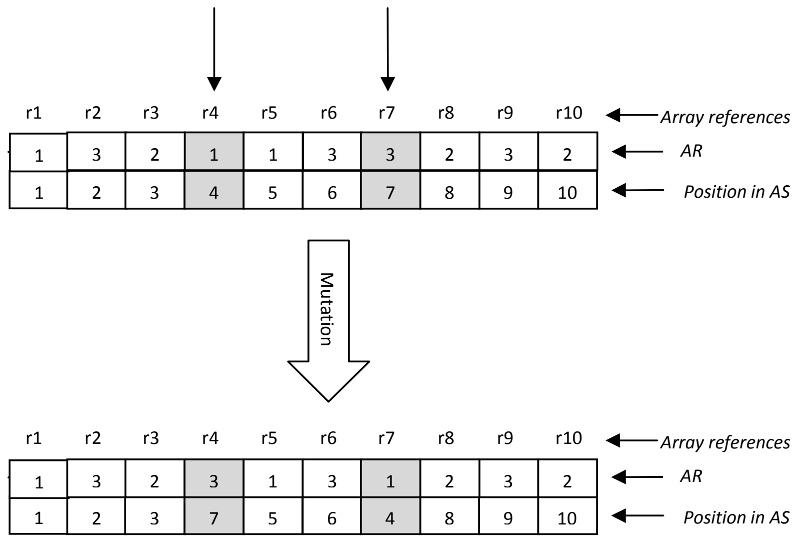


FIGURE 5.4. The mutation operation for the GA with code restructuring.

tions in a reasonable amount of time, we used a *genetic algorithm* (referred to GA) [52] (see Figure 5.6). GA is a well known technique that can result in good solutions for NP-complete problems. GAs work with a family of solutions, known as the current population, from which we obtain the next generation of solutions. When the algorithm is designed properly, we obtain progressively better solutions from one generation to the next. The main advantage of using GAs is in the fact that it only needs an objective function with no specific knowledge about the problem space. The challenge, however, remains in finding an appropriate problem representation that results in an efficient and successful implementation of the algorithm.

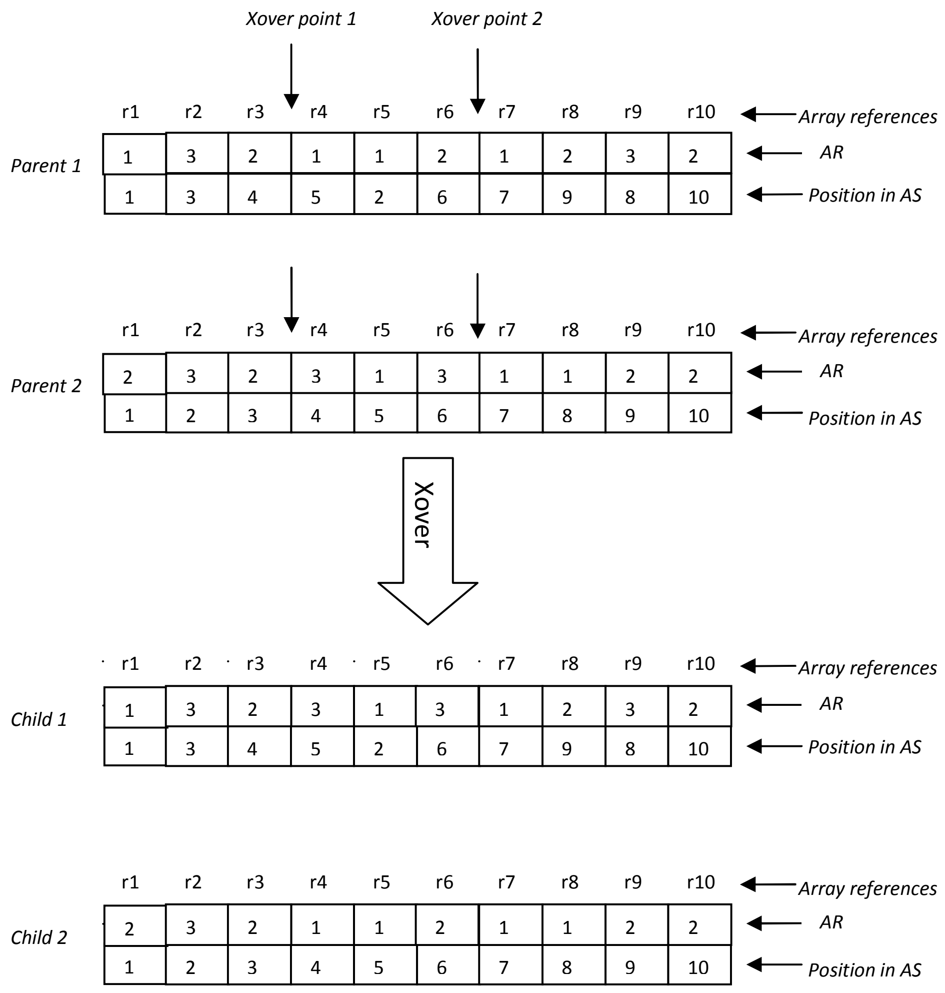

Xover point 1
Xover point 2
Parent 1
Parent 2
Child 1
Child 2
r1 r2 r3 r4 r5 r6 r7 r8 r9 r10
Array references
AR
Position in AS
Parent 1 AR: 1 3 2 1 1 2 1 2 3 2
Parent 1 Position in AS: 1 3 4 5 2 6 7 9 8 10
Parent 2 AR: 2 3 2 3 1 3 1 1 2 2
Parent 2 Position in AS: 1 2 3 4 5 6 7 8 9 10
Xover
Child 1 AR: 1 3 2 3 1 3 1 2 3 2
Child 1 Position in AS: 1 3 4 5 2 6 7 9 8 10
Child 2 AR: 2 3 2 1 1 2 1 1 2 2
Child 2 Position in AS: 1 2 3 4 5 6 7 8 9 10

FIGURE 5.5. The crossover operation for the GA with code restructuring.

In order to solve the ARA problem, we propose the chromosomal representation shown in Figure 5.3(a). This representation is based on a vector where every gene corresponds to an array reference with its corresponding address register. The corresponding AR for each array reference is not static and it may change as the GA operators are applied. Part of the initial population is constructed randomly and the rest is based on a random perturbation of the available chromosomes using the crossover operator. Within each generation, individuals are selected for reproduction using the genetic operators *mutation* and *crossover* that are applied iteratively with their corresponding probabilities.

Mutation is an important operator that introduces incremental changes in the offspring by randomly changing allele values of some genes. The algorithm randomly chooses an array reference, r_i and changes its address register allocation to a randomly chosen AR. *Crossover* is the main genetic operator as it provides a mechanism for the offspring to inherit the characteristics of the parents. We use a *two-point crossover* that randomly chooses two chromosomes that are split into three segments of contiguous genes. The offsprings are created by taking alternative segments from the two parents. The fitness of an individual is crucial for the transmission of its gene information to the next generation. The fitness of an individual solution in our formulation is the reciprocal of the number of explicit address arithmetic instructions needed by such a solution.

For the ARA with code restructuring, we propose the chromosome representation in Figure 5.3(b). In the case of code restructuring, the position of an array reference in the access sequence (AS) is needed. The mutation and crossover operations for the ARA with code restructuring are as follows:

- *Mutation*: The mutation operation exchanges the contents of two genes in the chromosome. The array register and the position in the access sequence (AS) of those two array reference will be exchanged as shown in Figure 5.4.

- *Crossover*: The crossover operation is a two-point crossover that randomly chooses two parent chromosomes and two crossover points and then exchanges the contents of the genes that encode the address registers (AR) between those two points to create the two child chro-

```
Genetic_ARA()
{
  M = Population size.
  N0 = Population size/2.
  Ng = Number of generations.
  Read the Access sequence.
  Construct the Indexing Graph (IG).
  Read the number of AR.
  Get the population size and the number of generations (Ng).
  Generate an initial population, current_pop
  for i = 1 to M
    evaluate(current_pop)
  Keep the best()
  for i = 0 to Ng do
    for j = 0 to N0 do
      Select two chromosomes from current_pop for mating.
      Apply crossover with probability $P_{xover}$.
    for k = 0 to N0 do
      Select a chromosome from current_pop.
      Apply mutation with probability $P_m$.
  Evaluate the population fitness.
  new_pop = select(current_pop, offspring).
  current_pop ← new pop.
}
```

FIGURE 5.6. Our genetic algorithm for the address register allocation problem.

mosomes. The crossover operation does not exchange the values in the genes that encode the position in the AS as shown in Figure 5.5; those genes can be changed by the mutation operation.

Note that not any repositioning of references in the access sequence is legal. For instance a *write* in an instruction code statement cannot be positioned before the *read* in the same statement. The *GA* will follow the criteria described in Section 5.2 to determine if a certain solution is feasible or not. An illegal solution will be penalized in the sense that its fitness is defined to be close to zero. Such solutions will most probably not be chosen for the next generation.

5.4 Results

We implemented our ARA techniques and the techniques in [42, 14, 7]. We compared our results to those achieved by the techniques in [14] as they have the best results. We executed our ILP formulations using CPLEX [1]. The techniques were tested using real-life benchmarks from

DSPstone [82] as well as statistical analysis using generated test files. We tested those test files on the problem of finding the minimum number of address registers for a zero number of address arithmetic instructions as well as for the the minimum cost for a given number of address registers. We performed our analysis under different array access sequence length. Our optimal ILP formulations decreased the number of ARs needed by 4% to 11% whereas they decreased the cost, number of explicit address arithmetic instructions, by 8% to 15%. A reduction in the number of address registers needed is also important as the number of ARs in a DSP system is usually limited. Note that even a small cost reduction makes a difference since the loop body is sometimes executed thousands of times equal to the loop index.

The ILP formulation for the code restructuring techniques was also evaluated on our test files. Based on the dependence analysis of the statements as well as the feasibility analysis, the IGs were updated to include all the possible reordering edges such that the offset between the end nodes of each edge is within the auto-modify range, [-1,1] in our case. On average, the number of address arithmetic instructions is decreased by 12.1% whereas the minimum number of address registers needed for a zero cost went down by 11.4%. Note that as the length of the access sequence increases, the improvements in the results usually increase as more array references and statements in the loop body translate to more reordering opportunities. All of the results were achieved in a matter of seconds ($\leq$ 45 seconds in all cases).

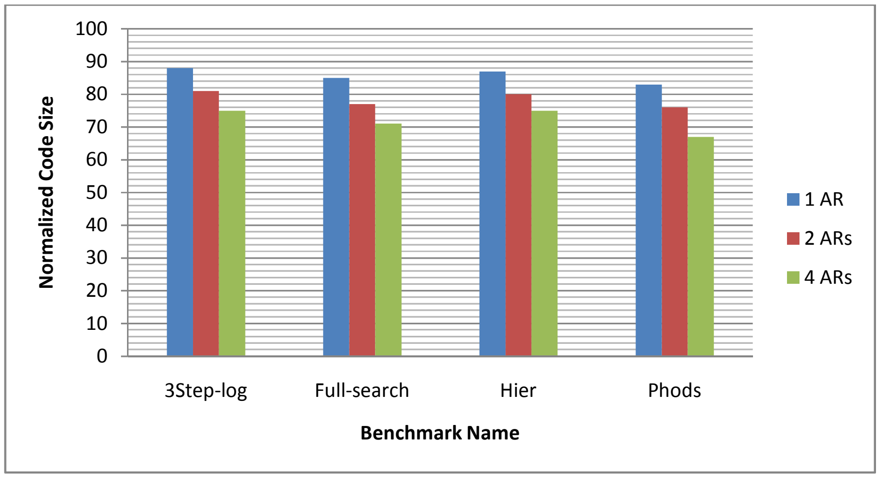


FIGURE 5.7. Normalized code size with respect to [6] for 1, 2, and 4 ARs

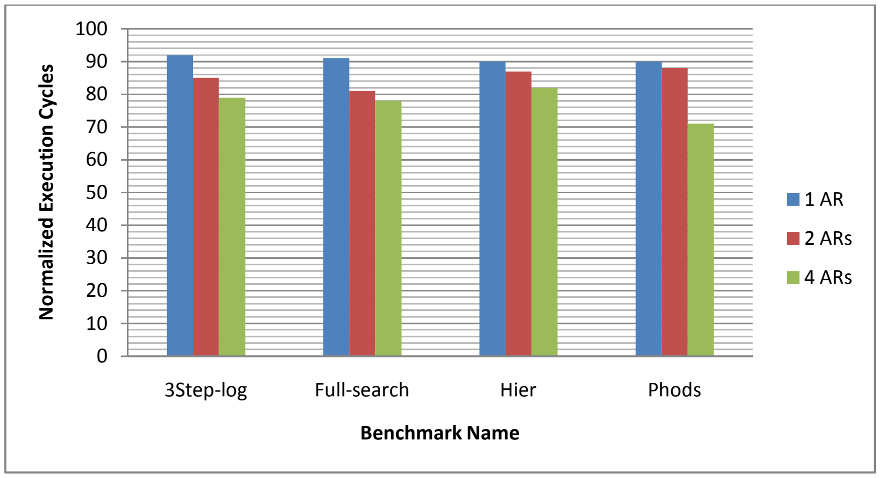


FIGURE 5.8. Normalized execution cycles with respect to [6] for 1, 2, and 4 ARs

Next we implemented our GA with and without code restructuring and tested it on four array-intensive real-life benchmarks, namely, $3Step-log$, $Full-search$, $Hier$, and $Phods$ [80]. Various GA parameters are important in achieving good results. Given a sufficient population size and number of generations, a suboptimal but good ARA solution can be found; however, the GA execution time is directly proportional to both parameters. We have experimentally determined that for the problems we attempted, a population size of 150 and a generation number of 400 were sufficient to achieve good solutions. We have also determined experimentally the crossover probability P_{xover} to be 0.65, and the mutation probability, P_m to be 0.35.

The results in Figure 5.7 and Figure 5.8 show respectively the normalized code size and execution cycles for our GA with code restructuring compared to the techniques used in [14] for 1, 2 and 4 ARs. The average reductions in code size with 1, 2, and 4 ARS are 14.25%, 21%, and 28%, respectively. The average execution cycle reductions with 1, 2, and 4 ARS are 9.25%, 14.75%, and 22.5%, respectively. Our results also show a reduction in code size of 8% to 10% more than the quantified results in [18] when code restructuring techniques are used. The execution time of our GA was between a few seconds in some cases to a few minutes in bigger applications.

To show the effectiveness of the code restructuring techniques, we tested the four benchmarks, $3Step-log$, $Full-search$, $Hier$ and $Phods$ using the GA without code restructuring. Figure 5.9 shows the execution cycles for the benchmarks from the codes generated based on the GA with the code restructuring address register allocation solution normalized with respect to the execution

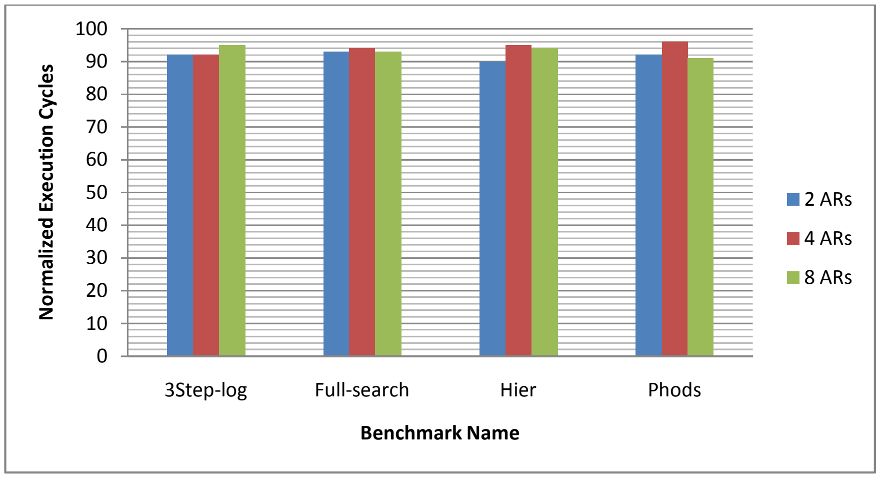


FIGURE 5.9. Normalized GA with code restructuring execution cycles with respect to GA without code restructuring for 2, 4, and 8 ARs

cycles of the codes generated from the GA without code restructuring. The code restructuring techniques improved the results up to 10%, 8% and 7% for 2 ARs, 4 ARs and 8 ARs, respectively.

The results clearly show the significant potential for improvement as well as the effectiveness of the code restructuring techniques to reduce the code size and the execution cycles. This reduction in the number of explicit address arithmetic instructions and the number of execution cycles is crucial for digital signal processors (DSPs), which are designed under very tight constraints on area, memory, power, etc. In such DSP systems, the code usually resides on the ROM and thus the code size directly translates into silicon area. This implies that optimal or near-optimal solutions are much desired in such systems even though the execution time of techniques that generate the executed code is much higher than that of heuristics. Although not studied in this chapter, one can expect a similar reduction in the power consumption due to the smaller code size and fewer execution cycles.

5.5 Chapter Summary

Many DSP algorithms have an iterative pattern of references to array elements within loops. A careful assignment of array references to address registers reduces the number of explicit address register instructions. Code size reduction is essential for such processors due to the tight constraints such as area and cost. Code size reduction also results in fewer execution clock cycles and thus less energy consumption. Due to tight constraints in DSP processor systems, optimal or near-optimal solutions are highly desirable; the potentially high execution time of the techniques that produce these solutions often outweights the great benefits of such (near-) optimal solutions. In this chapter, the address register allocation problem for array-intensive DSP applications is studied. To the best of our knowledge, we are the first to develop integer linear programming (ILP) formulations of the problems (with and without code restructuring) and also the first to develop a genetic algorithsm (GA) solution to the problem with code restructuring to further improve the results. Results on several benchmarks show the effectiveness of our approaches.

Chapter 6
Task Scheduling and Memory Partitioning for MPSoC: Single Application

The current trend in modern complex embedded system design is to deploy a multiprocessor system-on-chip (MPSoC), thanks to recent advances in architecture, VLSI and electronic design. Generally speaking, an MPSoC consists of multiple heterogeneous processing elements (PEs), memory hierarchies, and I/O components interconnected by complex communication architectures. Such architectures provide the flexibility of simple design, high performance, and optimized energy consumption. An MPSoC provides an attractive solution to the problems brought forth by increasing complexity and size of embedded systems applications. Execution time predictability is a critical issue for real-time embedded applications; this makes the use of data caches not suitable as a cache is hardware-controlled and hence it is hard to model the exact behavior and to predict the execution time of programs. To alleviate such problems, many modern MPSoC systems use software-controlled memories known as *scratchpad memories* (SPMs).

An SPM is fully software-controlled and hence the execution time of an application on such memories can be predicted with accuracy. Unfortunately, scratchpad memories are expensive and hence they are usually of limited size and as a result not all the application data variables can be stored in the on-chip scratchpads. Many multi-processor system-on-chip models use a memory hierarchy with slow off-chip memory (DRAM) and fast on-chip scratchpad memories. Such a hierarchy means that proper allocation of variables to the on-chip memory is an essential part in reducing the off-chip accesses. The computation time of a program on a processor depends on how much SPM is allocated to that processor as accessing an element from the off-chip memory is usually in the order of 100 times slower than accessing elements stored locally in the on-chip memory.

An embedded application can usually be divided into multiple tasks, and different tasks can be scheduled on different processors. The computation time for each task depends on the amount of

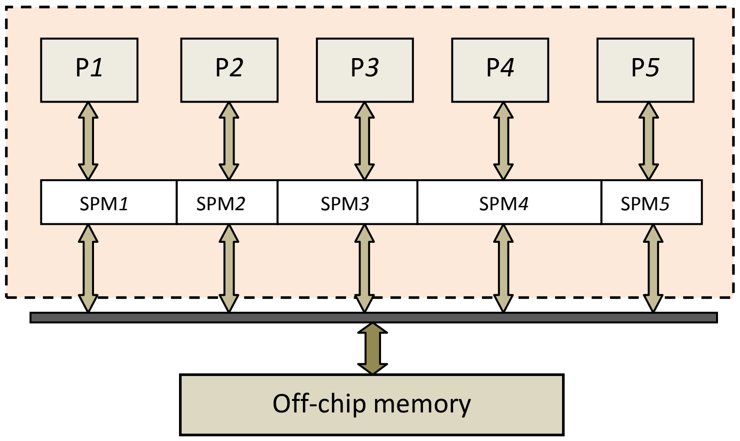


FIGURE 6.1. An architectural model example with five processors, SPM budget, off-chip memory and interconnection buses.

SPM allocated to the processor executing this task. The problem of task scheduling and memory allocation on MPSoCs is an NP-complete problem [38]. Traditionally, these two steps are done separately where tasks are usually scheduled and the SPM budget is then partitioned among the processors. Such a decoupled technique may not result in better schedules in terms of minimizing the computation time of the whole application. The appropriate configuration of a processor's scratch pad memory depends on the tasks scheduled on that processor. Therefore, the integration of those two steps is critical to improve the performance. In this chapter, we present a heuristic that performs task scheduling and SPM memory partitioning in an integrated fashion where the private on-chip memory budget allocated to a processor is decided dynamically as tasks are mapped to this processor.

6.1 Task Scheduling and Memory Partitioning

6.1.1 Architecture Overview and Problem Definition

Dividing an application into a set of tasks where one or more independent tasks can be executed in parallel on the available processors is extremely useful for MPSoCs. Parallelism leads to potential for speeding up the execution time; this is a major issue in embedded processors. A typical MP-SoC is shown in Figure 6.1 which consists of multiple processors, an SPM budget divided among the processors, and a global off-chip memory that can be accessed by all the processors. Our problem formulation is based on a task dependence graph (*TDG*).

Definition 6.1: *A TDG is a directed acyclic graph with weighted edges where each vertex represents a task in the embedded application. An edge between two tasks, say T_i and T_j in the* TDG, *represents a scheduling order that needs to be enforced due to the fact that T_j needs data to be transferred from T_i after T_i is already executed.*

A certain processor cannot start executing task T_j unless all the necessary data communication is performed. The weight of an edge is the communication cost. Each task can be mapped to any of the available processors. Since the processors in our architectural model can be heterogeneous, the execution time of each task depends on the processor to which this task is mapped as well as the SPM memory allocated to that processor. Generally speaking, a larger SPM results in less computation time since off-chip access is more expensive in terms of the clock cycles compared to fast on-chip SPM. A large portion of the execution cycles of a task goes to accessing the data variables. Accessing a data variable from an SPM is usually in the order of 100 times faster than accessing it from the off-chip memory. Since the available SPM memory is usually limited due to the MPSoC's design constraints, a good utilization of SPM can be critical in narrowing the gap with the processor's speed. The problem can now be stated as follows.

Problem Definition: *Given an embedded application consisting of t tasks, an MPSoC architectural model and an SPM budget: (i) find a schedule of those tasks on the available processors, (ii) partition the SPM memory among the processors, and (iii) assign data variables of a certain task T scheduled on processor P to the private SPM budget assigned to P. The objective is to minimize the execution time in cycles of the embedded application on the MPSoC architectural model.*

6.1.2 Motivation

Most works so far have treated task scheduling and memory partitioning as two decoupled steps that are performed independently. Given a set of tasks and an MPSoC model with a certain amount of available scratch pad memory budget, tasks are usually scheduled on the processors and then memory is partitioned among used processors. In this aspect, those two steps are performed independently. However, the configuration of a processor's scratch pad memory is highly dependent on the tasks scheduled on this processor. Thus, task scheduling and memory partitioning are interdependent on each other and they should be integrated in one step in order to get high quality schedules.

The computation time of a task depends on the processor to which it is mapped as well as on the SPM memory available for that task. Therefore, task scheduling should take into consideration the varying computation time of a task based on the processor and on the SPM budget. Considering static computation time, meaning that the computation time is fixed from the scheduler point of view, may limit the quality of the schedule.

Consider the example in Figure 6.2 (a) of a task graph with 6 tasks, *T1, T2, T3, T4, T5,* and *T6*. Task *T4* depends on tasks *T1, T2* and *T3,* and task *T6* depends on tasks *T4* and *T5*. Any time there is an edge between two tasks *Ti* and *Tj* means that a communication cost should be accounted for provided that those two tasks are allocated to two different processors. Although our technique takes such costs into account, we omit them in Figure 6.2 for simplicity. Define Min_{ij}, Avg_{ij}, and Max_{ij} as the computation time for task T_i on processor P_j assuming all of the available SPM budget is assigned to P_j, $1/n$ of the available SPM budget is assigned to P_j where n is the number of processors, and no SPM is assigned to P_j, respectively. Those values will be used later on by our heuristic. In this example, we assume two homogeneous processors. The (Min, Avg, Max) values are shown in Table 6.1. Figure 6.2 (b) shows the schedule assuming no available scratch pad memories. First tasks *T1* and *T2* will be mapped to the two available processor $P1$ and $P2$. At this time only task *T3* is ready to be scheduled. The scheduling algorithm will map *T3* to $P2$ as it is free before $P1$ since the computation time of *T2* is less than that of *T1*. In a similar fashion, the scheduling algorithm will assign tasks *T4* and *T6* to processor $P1$ whereas task *T5* will be mapped to processor $P2$. The cost of such a schedule is equal to 29.

Figure 6.2 (c) shows the results following the common practice of partitioning the available SPM memory equally between the two processors. With such a criterion, the available SPM budget will be equally divided between processors $P1$ and $P2$ regardless of what tasks are mapped to what processors. Equally partitioned SPM reduces the computation time of the whole application to 25.

TABLE 6.1. Min, Avg, and Max values

Tasks	Min	Avg	Max
T1	7	9	15
T2	8	9	10
T3	3	5	6
T4	4	5	6
T5	2	2.5	3
T6	5	6	7
T7	5	6	7

To further reduce this application's computation time, the available SPM can be divided between the two processors in any ratio.

From the task schedule, we can see that task $T4$ can start only after $P2$ is done executing task $T3$. The issue now is to try to reduce the dead time between tasks $T1$ and $T4$ imposed by the computation time for tasks $T2$ and $T3$. To minimize this dead time, techniques usually allocate more SPM budget to processor $P2$ to reduce the computation time of tasks $T2$ and $T3$. Notice that if all the SPM memory is allocated to processor $P2$ then the computation time for $T1$ will jump to 15 and as the results the minimum start time of $T4$ will increase from 14 to 15. To avoid this increase, some SPM memory should be allocated to $P1$ to keep the execution time as balanced to the end time of $T3$ as possible. Intuitively speaking, the approximated minimum end time of $T3$ will be 12 and thus the total computation time for our example application will be close to 23. With the same memory partitioning, the computation time can be reduced to 22 assuming that tasks $T4$ and $T6$ are scheduled on P_2 and task $T5$ is mapped to P_1, Figure 6.2 (d).

However, 22 is not the optimal time for scheduling the example task graph on two processors. Our heuristic, presented later, can reduce the computation time to 19 as it integrates task scheduling and memory allocation into one step. The problem with the previous schedule is that it allocated T_3 to the same processor P_2 that is scheduled to execute T_2. This choice is the reason for the dead time in the schedule as T_2 cannot benefit much from more SPM memory which is clear from the Min, Avg, and Max values. A good heuristic should take those values into consideration where a better choice for T_3 is to be scheduled on P_1 with all available SPM memory is allocated to this processor

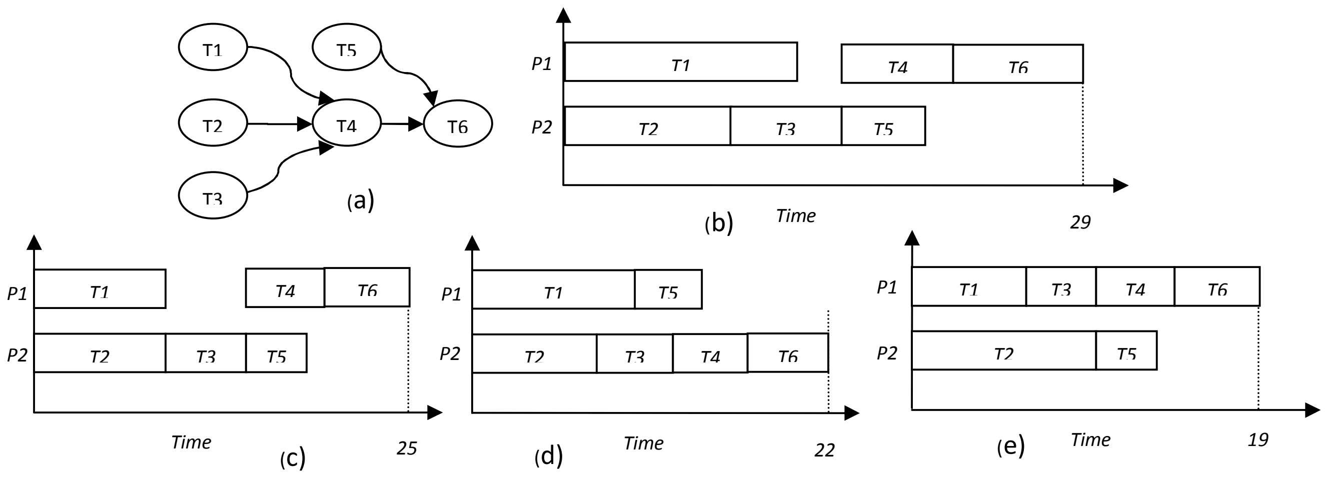


FIGURE 6.2. (a) TDG. Schedule based on: (b) no SPM. (c) equal partitioned SPM. (d) non-equal partitioned SPM. (e) our integrated approach.

and the result is a schedule with minimal end time of 19; see Figure 6.2 (e). A benchmark example is presented in Section 6.2.

6.1.3 Our Heuristic

A good heuristic for task scheduling and memory partitioning should take into consideration the dynamic (varying) execution time of a task throughout the process of building the schedule. This dynamic execution time is the result of the dynamic SPM budget assignment to processors throughout the course of the heuristic. Using profiling of the tasks in the embedded application, *Min, Avg,* and *Max* values (defined earlier) are calculated for each task on each of the available heterogeneous processors. We define *elasticity* of a task as the extent to which this task can benefit from a larger SPM. Although it can be defined in different ways, we define *elasticity* dynamically as the extent to which the computation cost of a task on P_i may decrease as the SPM budget of P_i is increased from the current budget to *size* where *size* is the maximum amount of SPM budget available in our model. Equation 6.1 defines *elasticity* of task T_i where *Cur* is the computation time of the task under the current memory budget. The *elasticity* of a task T_i is basically a measure of the room for computation time reduction of T_i with more SPM budget.

$$elasticity(T_i) = \frac{Cur_i - Min_i}{Cur_i} \tag{6.1}$$

A bigger value of *elasticity* means that the computation time of T_i is more amenable for reduction with the increase in the SPM allocated to that task. Note that $elasticity(T_i)$ is a dynamic value since the current computation time of T_i, Cur_i, may change as the SPM budget distribution changes.

Our heuristic in Figure 6.3 starts with profiling the application to extract important information. Using the profiling data, the embedded application will be divided into tasks with a necessary data communication between two tasks impose a certain kind of dependency. Based on the extracted tasks and the communication between them, the task dependence graph is created. In this graph, each task is represented by a vertex and each communication cost by a weighted directed edge. For each available task T_i and processor P_j, we calculate the number of variables, the size of the variables, Min_{ij}, Avg_{ij}, and Max_{ij} values. All those values are computed through profiling. Then the ASAP values for all tasks are calculated based on the Avg values that is assuming the SPM budget is equally divided among the available processors. Tasks will be sorted in increasing order of the ASAP values in a list $L1$. For each task, following the ASAP sort, we evaluate the best processor to assign this task to so that the overall computation time is minimally increased.

The minimum start time of a task T_i on processor P_j, $Start\_time(T_i, P_j)$, is equal to the maximum of the end time of processor P_j, $End\_time(P_j)$, and the maximum end time of all its parent tasks, $Max_{T_j \in Parent(T_i)}(T_j)$, plus the maximum communication time of all the parent tasks scheduled on P_k with $k \neq j$ (see Equation 6.5). Two dependent tasks mapped to the same processor will have zero communication cost. In general, task T_i will be scheduled on the processor P_j corresponding to the minimum additional overhead time in the schedule.

However, T_i may be scheduled on a processor P_l of higher overhead time provided that the predicted end computation time ($PEC(P_l)$) (defined by us in Equation 6.2) of this processor is at least δ % less than that of P_j. We choose δ of 10 in our experimental evaluations. This $PEC(P_l)$ value is a guide to the scheduler of how much this over head time may decrease with the SPM memory transfers in future steps if T_i is mapped to P_l. PEC is basically an estimate of how much the end time of processor P_l will be if more SPM budget is assigned to it.

The PEC of a processor is closely related to the *elasticity* of the tasks scheduled on that processor. The PEC value provides the dynamic essence of our heuristic as at each step the heuristic looks beyond the current SPM budgets distribution in its task mapping decision to an estimate of future distributions in future steps. In the case of equal additional end time, if task T_i is assigned to two different processor, then we avoid assigning it to a processor with no scheduled tasks. In this case, we schedule T_i on the processor with the higher *elasticity* under the current SPM budget. The *elasticity* of a processor is the average value of the *elasticity* of the tasks scheduled on this processor.

$$PEC(P_i) = End\_time(P_i) - \sum_{T_j \in P_i} \left(Cur(T_j) - \frac{Cur(T_j)}{1 + elasticity(T_j)} \right) \tag{6.2}$$

After scheduling any task, we try to balance the schedule in a way to decrease the total computation time. We do so by dynamically changing the SPM budget for each processor to reach a better balance. We start by trying to reduce the computation time of tasks on processor P_i with maximum end time so far. We do so by transferring an α % of the memory budget, Mem_j, corresponding to processor P_j with the minimum (*End_time*elasticity*) and such that $End\_time(P_j) < End\_time(P_i)$ and assigning it to processor P_i. Doing so will probably decrease the end time of processor P_i and in the same time increase the end time of processor P_j. Considering processor P_j to be of low total *elasticity* will give more room to reduce its SPM budget with a minimal increase in its End_time. We do memory transfer α% at a time as long as $End\_time(P_j) < End\_time(P_i)$. we choose an α equals to 10 in our experiments.

$$Time(T_i, Mem_j) = Time(T_i, 0) - Gain(T_i, Mem_j) \tag{6.3}$$

$$Gain(T_i, Mem_j) = \sum_{v_i \in T_i, v_i \in Mem_j} ((\beta_1 - \beta_2) * freq_i). \tag{6.4}$$

$$\begin{aligned} Start\_time\left(T_i, P_j\right) &= Max\left(Max\left(End\_time_{T_k \in Parent(T_i)}\left(T_k\right)\right), End\_time\left(P_j\right)\right) \\ &+ Max\left(Comm\_time_{T_k \in Parent(T_i)}\left(T_k\right)\right) \end{aligned} \tag{6.5}$$

$$End\_time(T_i) = Start\_time(T_i, P_j) + Time(T_i, Mem_j) \tag{6.6}$$

$$End\_time(P_j) = Max(End\_time_{T_k \in P_j}(T_k)) \tag{6.7}$$

After any SPM memory budget redistribution among different processors, the $Recompute()$ subroutine in Figure 6.5 will be invoked to recompute the start time, computation time, and end time of tasks T_i referred to, respectively, as $Start\_time(T_i)$, $End\_time(T_i)$, and $Time(T_i)$. First a *Gain* value, $Gain(T_i,Mem_j)$, is computed for T_i with the newly budget SPM memory, Mem_j, assigned to the processor to which T_i is mapped. This *Gain* value in Equation 6.4 , represents the execution cycles reduced due to allocating variables of T_i to Mem_j following the increasing order $byte/freq$ of the data variables where $byte_i$ is the size of the variable v_i and $freq_i$ is the number of times v_i is accessed. In Equation 6.4, β_1 is the cost of accessing a variable from the off-chip memory and β_2 is the cost of the SPM access. This is a simple data allocation technique that we adopted in our heuristic. The new computation time of T_i, $Time(T_i, Mem_j)$, is the time taken to execute T_i assuming no SPM memory, $Time(T_i,0)$, minus $Gain(T_i,Mem_j)$. We assume on-chip memory access costs only one clock cycle. The end time of a task T_i scheduled on processor P_j is then calculated as in Equation 6.6. The end time of each processor is thus the end time of the last task assigned to this processor (Equation 6.7).

After all the tasks are scheduled, we call the $Balance()$ procedure, Figure 6.4, to try to further reduce the schedule cost through reducing the end time of the processor with the largest end time by performing memory transfers. At this point we tune the $Balance()$ procedure so that it allows the last memory transfer between P_i and P_j that will result in $End\_time(P_j) > End\_time(P_i)$. We run this procedure t times where t is the number of tasks in the *TDG*. Notice that if a processor ends up with no scheduled tasks, then the SPM budget for such processor will be distributed among other processors using the $Balance()$ procedure to reduce the schedule time the most.

6.1.4 Pipeline Scheduling

An embedded application is usually executed many times for a stream of input data on an MPSoC. Such multiple executions make embedded applications amenable to pipelined implementation. Pipeline scheduling benefits from allowing tasks from different embedded application instances to be scheduled at each stage of the pipeline. Such a schedule does not necessarily decrease the computation time of one instance of embedded application but rather it decreases the time between the start of two consecutive iterations of the task graph. The objective is to decrease the pipeline stage time interval as after filling up the pipeline, an instance execution of the application is per-

Task scheduling and memory partitioning

```
1.  Divide the application into tasks $T_i$.
2.  Perform dependence analysis between tasks.
3.  Construct the $TDG$ based on dependence analysis and communication costs.
4.  Divide the SPM memory equally between the processors.
5.  For each task $T_i$ and processor $P_j$, extract the following:
6.      (i) Minimum computation time on $P_j$, $Min_{ij}$.
7.      (ii) Maximum computation time on $P_j$, $Max_{ij}$.
8.      (iii) Average computation time on $P_j$, $Avg_{ij}$.
9.  Find ASAP for all the tasks based on $Avg$ values.
10. $L_1$ = List of tasks in increasing order of ASAP.
11. While ($L_1$ not empty) do:
12.     Get the first task $T_f$ from $L_1$.
13.     For each processor $P_i$:
14.         Calculate the $elasticity$ and $PEC$ of $P_i$ if $T_f$ is mapped to $P_i$.
15.         Find the minimum start time of $T_f$ on $P_i$.
16.         Find $END\_time(P_i)$ if $T_f$ is mapped $P_i$.
17.         if $((END\_time(P_i) < min$ && $PEC(P_j) \geq (1 - \delta\%)PEC(P_i))||$
                    $(END\_time(P_i) > min$ && $PEC(P_i) \leq (1 - \delta\%)PEC(P_j)))$
              (Comment: $P_j$ = processor corresponding to the current $min$ value)
18.             $min = END\_time(P_i)$
19.         else if $(END\_time(P_i) = min)$
20.             $min = END\_time$ of processor with the higher $elasticity$.
21.     End For
22.     Assign $T_f$ to $P_j$ corresponding to $min$.
23.     Delete $T_f$ from $L_1$.
24.     Call Balance().
25. End While
26. For i = 1 to $t$ do:
27.     Call Balance().
```

FIGURE 6.3. Our scheduling heuristic.

Balance()

```
1.  $P_i$ = processor with maximum end time, $End\_time(P_i)$.
2.  $P_j$ = processor with minimum $End\_time(P_j)*elasticity$.
3.  while$(End\_time(P_j) < End\_time(P_i))$ do:
4.      $Mem_i = Mem_i + \alpha\, Mem_j$.
5.      $Mem_j = Mem_j - \alpha\, Mem_j$.
6.      Recompute().
7.      if $(End\_time(P_j) \leq End\_time(P_i))$.
8.          Perform the memory update.
```

FIGURE 6.4. The balance routine.

Recompute()

```
1.  Following the ASAP sort of scheduled tasks $T_i$ and the new SPM budget distribution:
2.      Recompute $time(T_i, Mem_j)$.
3.      Recompute $Gain(T_i, Mem_j)$.
4.      Recompute $Start\_time(T_i, P_j)$ where $T_i$ is mapped to $P_j$ of SPM = $Mem_j$.
5.      Update the $Start\_time$ of all the tasks on $P_j$ successor to $T_i$.
6.      Recompute $End\_time(T_i, P_j)$.
```

FIGURE 6.5. The recompute routine.

formed each pipeline stage. The maximum number of stages is equal to the number of processors in the MPSoC system.

Our technique finds all the paths from the *dummy* start node to the *dummy* end node where the *dummy* start node is a node with an outgoing edge to all the nodes in the TDG with zero ingoing edges and the *dummy* end node is a node with an ingoing edge from every node in the TDG with zero outgoing edges. Then our technique tries to remove some edges in the TDG to reduce the time on the critical paths. We find the critical paths based on the *PEC* values defined earlier. The removal of an edge means that the nodes at the subgraph corresponding to the head of the edge, SG_h, and that corresponding to the tail, SG_t, belong to two separate stages in the pipeline. Any time an edge from T_i to T_j is removed, all the edges that connect SG_h and SG_t will be removed. The *TDG* can be at most divided into s unconnected graphs where s is the number of stages in the pipeline. Our task scheduling/memory partitioning heuristic will then be applied to the resulting TDG. An example of our pipeline technique is presented in Section 6.2.

6.2 Example

In this section, we present a task graph example to illustrate our heuristic as well as to show the effectiveness of integrating task scheduling and memory partitioning for embedded programs on a MPSoC. This task graph example is based on the *lame* benchmark from *MiBench* that consists of four tasks with their corresponding execution times in *Mega* cycles are shown in Figure 6.6 (a) assuming no SPM. In this example, we assume a multiprocessor architecture of two homogeneous processors, 4 KB scratchpad memory, and unlimited off-chip memory.

Figure 6.6 (a) shows the *lame* task graph with 4 tasks with data communications between tasks are represented by edges. We assume equal communication costs. Since task T_1 has the longest execution time with no SPM, usually current schedulers will map it to a separate processor. This is the solution that decoupled task scheduling/memory partitioning heuristics will output as they don't take into consideration the considerable reduction in computation time of T_1 with a bigger SPM memory. The solution is presented in Figure 6.6 (c) with a total pipeline stage interval of 8.5. Task T_2 is of small *elasticity* which implies that adding more SPM memory to P_1 will not help much in reducing the execution time. Tasks T_i, T_i', and $T"_i$ represents three instances of the same task from different runs of the application. In this solution 12.1 KB SPM memory is allocated to processor P_1 and the rest to P_2.

Since we have two processors in our MPSoC model, at most two pipeline stages are allowed. Our heuristic in Section 6.1 will find that there are two paths from the dummy source task to the dummy end task, p_{134} and p_{234}. Since there are only two processors, the parallel tasks T_1 and T_2 will be mapped to different stages in the pipeline. The important question now is whether to assign T_3 and T_4 to the same stage of T_1 or T_2. Based on the high *elasticity* value of T_1 compared to T_2 and based on the *PEC* values of p_{134} and p_{234}, our heuristic will map T_3 and T_4 to the same stage as T_1, namely S_2, since *PEC* of $p_{234} >$ *PEC* of p_{134}. The *PEC* of a path, say p_{234}, is calculated as $PEC(P_n)$ in Equation 6.2 assuming tasks T_2, T_3, and T_4 are mapped to processor P_n. After dividing tasks into different stages, our integrated task scheduling memory partitioning algorithm will be applied to the TDG in Figure 6.6 (b).

Figure 6.6 (d) shows our pipeline schedule with a pipeline stage of 7.19M cycles. This solution starts by assigning T_2 to P_1 and T_1' to P_2. After applying the *Balance*() procedure, 2.8 KB of SPM memory will be assigned to P_2 to balance the schedule as much as possible. The scheduler will then assign T_3' to P_2 since its end time is lower on this processor as its *elasticity* is high. The *Balance*() subroutine will update the memory budget for each processor by moving 1 KB from P_1 SPM budget to P_2 to balance the schedule. T_4' will also be assigned to P_2 as its end time will be smaller on this processor at this step due to the high SPM memory budget allocated to P_2. After

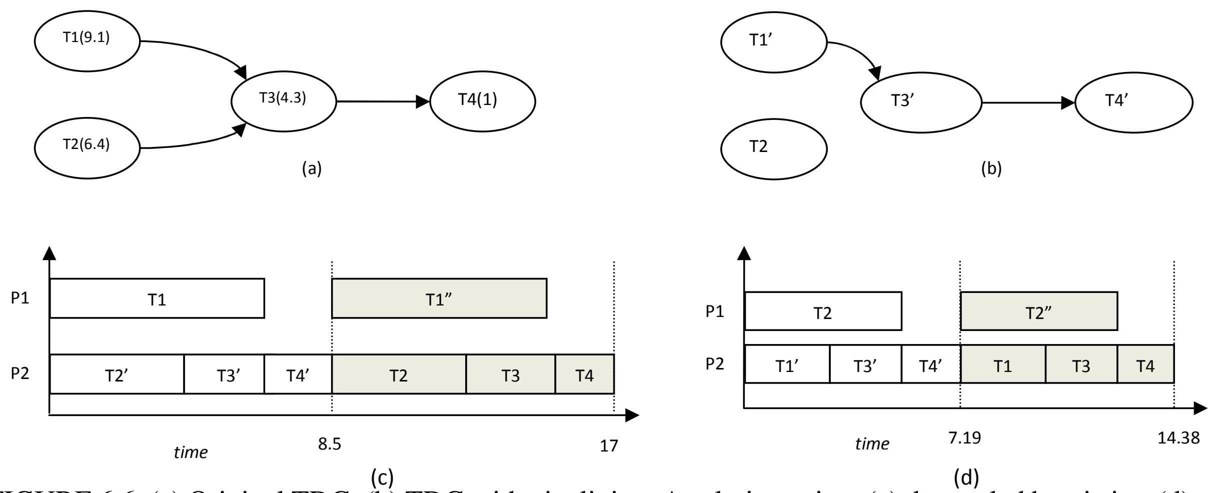


FIGURE 6.6. (a) Original TDG. (b) TDG with pipelining. A solution using: (c) decoupled heuristics. (d) our heuristic.

all the tasks are scheduled, the *Balance*() procedure will further reduce the cost by transferring the 0.2 KB SPM budget assigned to P_1 to P_2.

6.3 Experimental Results

We implemented five approaches to solve the task scheduling and memory allocation problem on MPSoC systems namely, (i) decoupled task scheduling and memory partitioning assuming equally partitioned SPM among all available processors *TSMP_EQUAL*; (ii) decoupled task scheduling and memory partitioning with SPM partitioned among different processors with any ratio, *TSMP_ANY*; (iii) our integrated task scheduling and memory partitioning heuristic described in Section 6.1, *TSMP_INTEG*; (iv) our heuristic with pipelining *TSMP_PIPE*; and (v) the optimal solution with pipelining based on the ILP formulation in [72], *ILP_PIPE* using the *CPLEX* ILP solver [1]. We used the following real life programs from the *Mediabench* and *MiBench, enhance*, *lame*, *osdemo*, *cjpeg*, *pgp*, *rasta*, *pegwit*, and *epic* as test benchmarks.

We used *Simplescalar* architectural simulation to profile the used benchmarks [9]. *Simplescalar* can simulate the execution of an application on a complex multiprocessor system on-chip architectures with different memory hierarchies. The MPSoC architecture used is similar to the one in Figure 6.1. The profiling is intended to (i) divide each application into computation blocks referred to as tasks, (ii) find the computation times for each task on each available processor in processor cycles, (iii) find the number of variables, (iv) the number of times each variable is used, *freq*, and

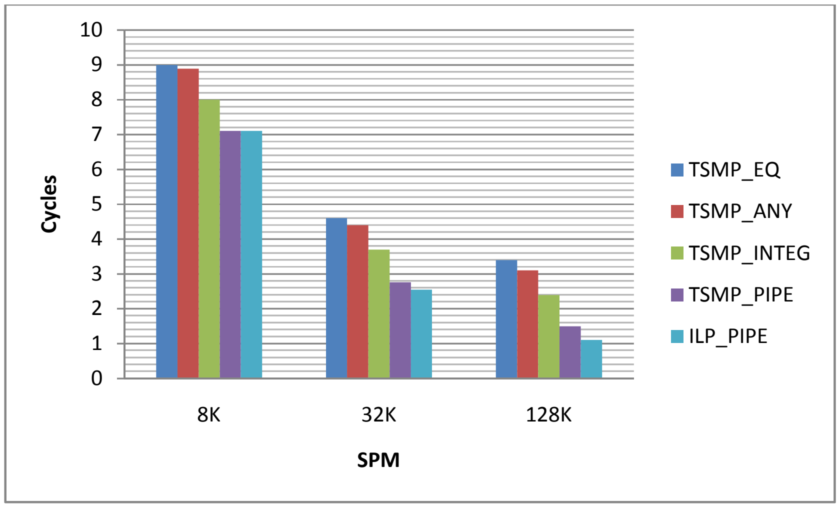


FIGURE 6.7. Results for lame benchmark

(v) the size in bytes for each variable in the current application. The profiler information is based on a system with only off-chip memory. Using the profile information and dependence analysis, a task graph is constructed with a vertex for each task and an edge to represent the communication cost between two tasks. The communication cost depends on the size of data to be communicated between the two tasks and it is calculated through profiling. We assume a 100 cycle latency for off-chip memory access compared to 1 cycle latency for the SPM on-chip memory.

We tested the benchmarks, *enhance*, *lame*, *osdemo*, and *cjpeg*, assuming a multiprocessor system on chip of two processors and a scratch pad memory with size that varies between 4KB and 4 MB. We tested each of our benchmarks under three SPM budgets chosen based on the size of the benchmark. The choice of SPM sizes for each benchmark is essential as too little SPM or too much SPM for a certain embedded application may not reflect the effectiveness of our heuristic. The off-chip memory size is assumed to be unlimited ,that is, it can hold all the data variables needed by the embedded application.

The first three columns in Figures 6.7–6.10 shows the comparison between $TSMP\_EQ$, $TSMP\_ANY$, and $TSMP\_INTEG$. The number of cycles in the results graphs are shown as $1.0E-06$ of the actual number of cycles. All of those results are based on α value of 10 meaning that 10% of SPM memory is being transferred between two tasks at a time in the $Balance()$ procedure. The improvement greatly depends on the structure of the embedded application. $TSMP\_ANY$ improved

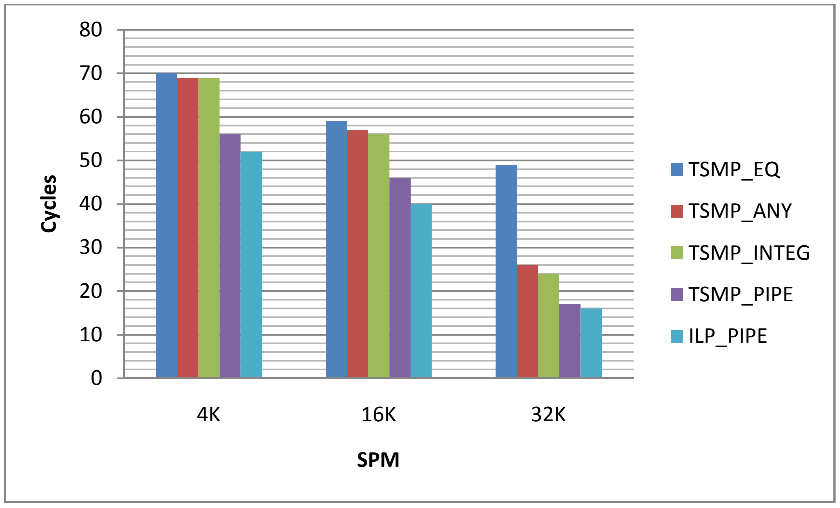


FIGURE 6.8. Results for osdemo benchmark

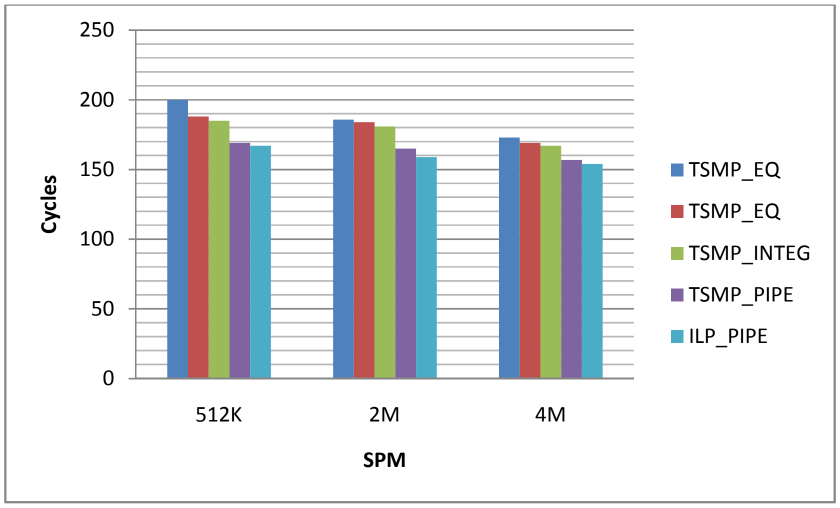


FIGURE 6.9. Results for enhance benchmark

over $TSMP\_EQ$ from little improvement close to 0% to dramatic improvement of 47%. Such improvements show that static memory allocation, that is, partitioning the SPM budget equally among the processors limits the effectiveness of SPM memories as it does not consider the characteristics of the tasks assigned to a processor in its memory partitioning decision.

Our integrated approach for task scheduling and memory partitioning, $TSMP\_INTEG$, further improved the results over the decoupled approach, $TSMP\_ANY$. $TSMP\_INTEG$ improved over $TSMP\_ANY$ from little improvement close to 0% in some cases to dramatic improvement of 22%. This improvement is due to the guidance that our integrated approach uses to partition the memory

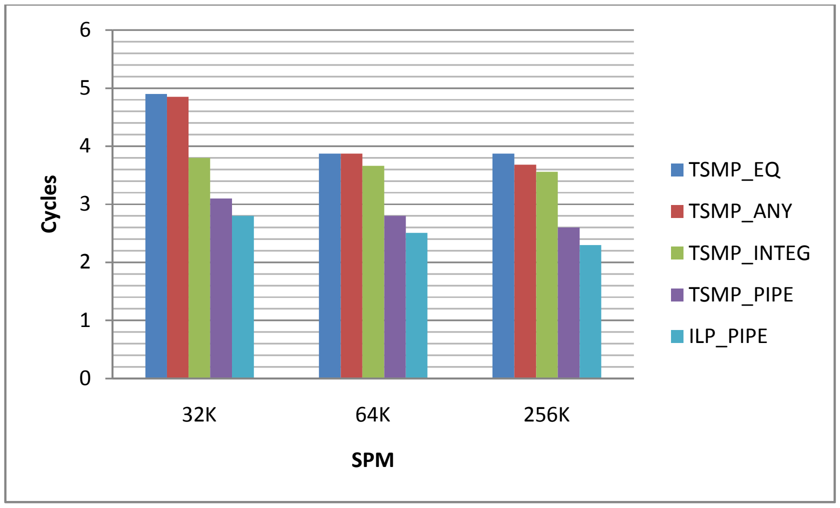


FIGURE 6.10. Results for cjpeg benchmark

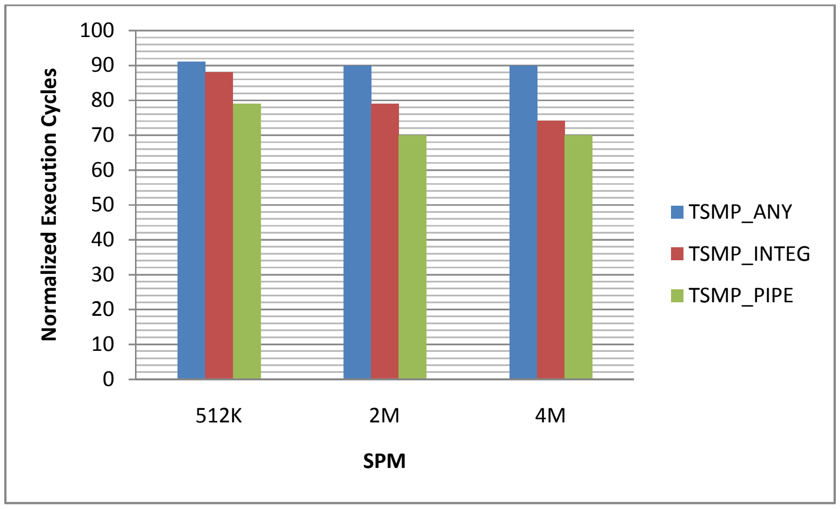


FIGURE 6.11. Results for pgp benchmark

based on the fact that the SPM configuration of a certain processor depends on the tasks mapped to that processor.

The fourth columns in Figures 6.7–6.10 show the result of our technique with pipelining, *PIPE*. The results emphasis the fact that such embedded applications can benefit significantly from pipelining. The pipeline cost is the computation time needed for one pipeline stage. As expected, our embedded applications greatly benefit from pipelining as the execution time is decreased by 27% in some cases.

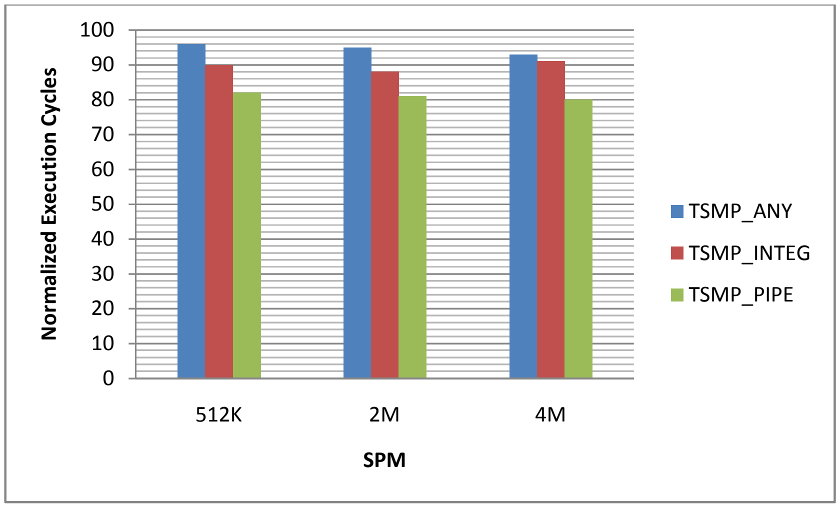


FIGURE 6.12. Results for rasta benchmark

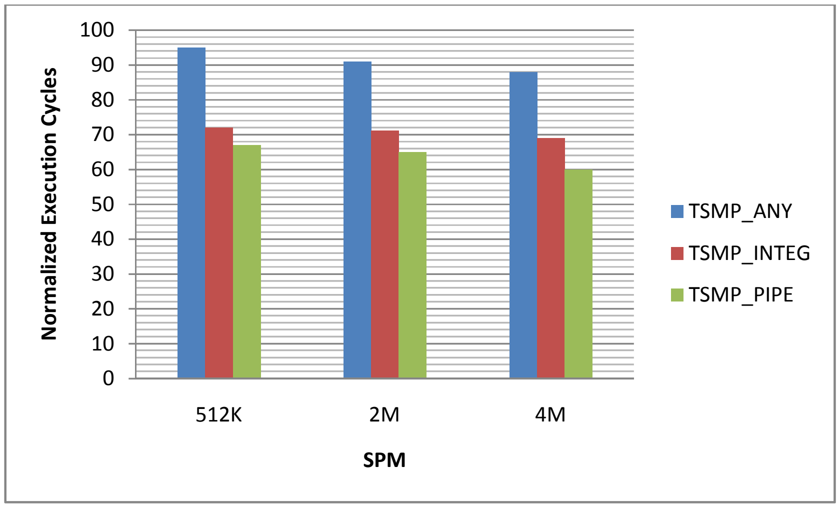


FIGURE 6.13. Results for pegwit benchmark

In order to show the effectiveness of our task scheduling/memory partitioning heuristic, *INTEG*, we compared it to an optimal integer linear formulation (ILP) based on the ILP formulation of this problem in [72], *ILP_PIPE*. The ILP solver is stopped after 35 minutes in some cases due to the long execution time taken by the ILP to produce optimal results. Following the same assumptions concerning the MPSoC system model and SPM memory budget, our $TSMP\_INTEG$ heuristic is in the range of 0% to 13% off the optimal solution in a negligible amount of time, Table 6.2. This shows the effectiveness of our heuristic as in most of the cases our solution was close to the ILP one.

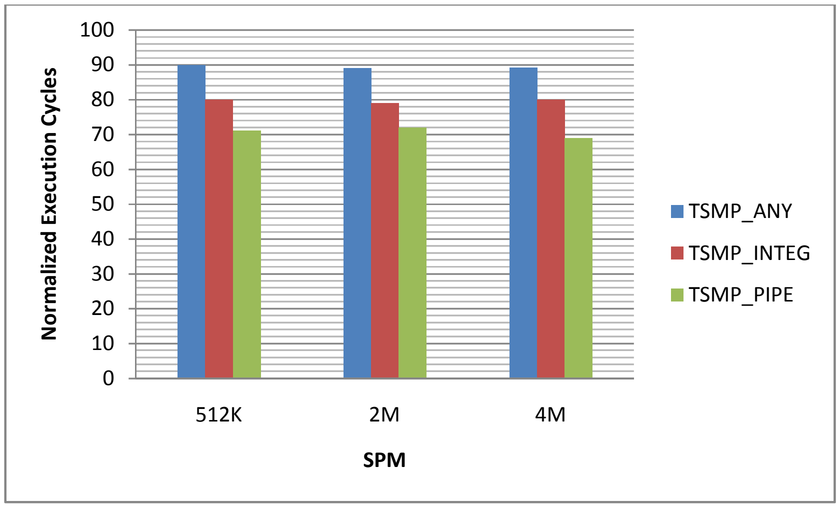


FIGURE 6.14. Results for epic benchmark

The ILP formulation is not scalable as the run time is exponential with the number of variables in the application. For large scalar based embedded application, the number of variables is usually large and thus the ILP will take very long time (hours) that makes the use of ILP infeasible for such applications. On the other side, our heuristic is of polynomial run time and thus it scales well with big applications as clearly shown in the run time in Table 6.2. We tested our heuristic on the following four large embedded applications mainly, *pgp*, *rasta*, *pegwit*, and *epic*. Figures 6.11–6.14 show the results achieved by our heuristic when considering a system with 4 processors and an SPM budget ranging from 512K to 4M. The results in Figures 6.11–6.14 are the normalized execution cycles with respect to *TSMP_EQ*. *TSMP_ANY*, *TSMP_INTEG*, and *TSMP_PIPE* improved over *TSMP_EQ* up to 12%, 33%, and 40% respectively.

Keep in mind that our solutions can be further improved if we decrease the value of α. The reduction in the value of α will add some overhead on the execution time but will try to exchange smaller ratio of the SPM between different tasks and this will further improve the results by up to 2.1% as shown in Figure 6.15. More aggressive SPM data allocation techniques will further improve the results.

Finally, we tuned our MPSoC architecture in Figure 6.1 so that the processors can access the SPMs of each other in 5 cycles time. We tested our heuristics with such an architecture and found an average reduction of 4% in the execution cycles compared to the architecture in Figure 6.1.

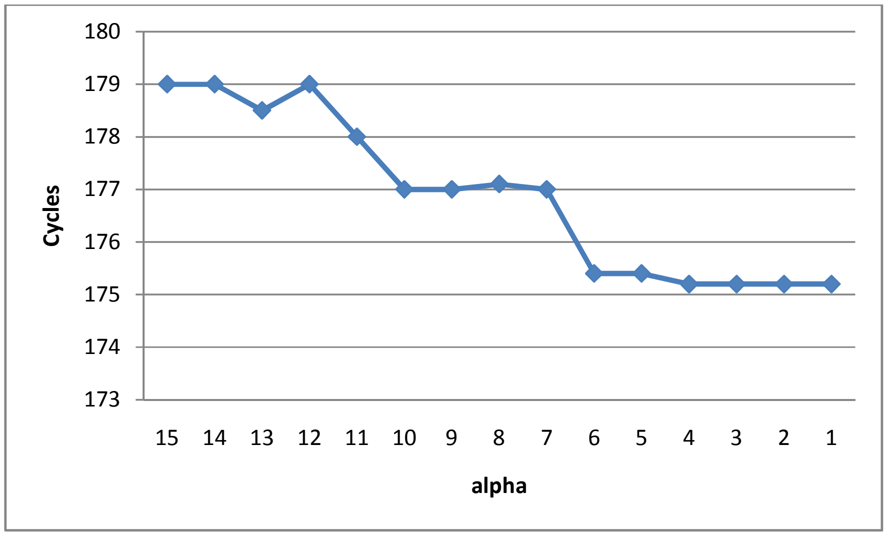


FIGURE 6.15. Cycles reduction as α decreases

TABLE 6.2. Run times for our heuristic and the ILP formulation.

Benchmark	# of variables	Our heuristic (sec)	ILP [72]
Lame	128	8	35 min
Osdemo	46	5	26 min
Enhance	44	3	34 min
Cjpeg	20	0.1	15 sec
Epic	410	51	–
Pgp	240	24	–
Rasta	300	38	–
Pegwit	320	42	–

6.4 Related Work

Many research groups have studied the problem of task scheduling for applications on multiple processors with the objective is to minimize the execution time. Benini et al. [15] solved the scheduling problem using constraint programming and the memory partitioning problem using integer linear programming. The authors argued why these two choices fit the two problems the best. Kwok and Ahmed [38] presented a comparison among algorithms for scheduling task graphs onto a set of homogeneous processors on a diverse set of benchmarks to provide a fair evaluation of each heuristic based on a set of assumptions. De Micheli et al. [51] studied the mapping and scheduling problem onto a set of processing elements as a hardware/software codesign. Neimann and Marwedel [55] used integer programming to solve the hardware/software codesign partitioning problem. A tool for hardware-software partitioning and pipelined scheduling based on a branch and bound algorithm was presented in [17]. Their objective was to minimize the initiation time, number of pipeline stages, and memory requirements. Similarly, Kuang et al. [37] proposed an ILP solution for the partitioning problem with pipelined scheduling. Cho et al. [20] proposed an accurate scheduling model of hardware/software communication architecture to improve timing accuracy.

Panda et al. [65, 66] presented a comprehensive allocation technique for scratchpad memories on uniprocessor to maximally utilize the available SPM memories to decrease the programs execution times. Optimal ILP formulations for memory allocation for scratch-pad memories were presented in [10, 22]. An ILP formulation for the SPM allocation problem to reduce the code size was presented in [68]. Steinke et al. [69] formulated the same problem with the objective to minimize the energy consumption. Angiolini et al. [6] optimally solved the problem of mapping memory locations to SPM locations using dynamic programming. Their algorithm works by mapping parts of the external memory to physically partitioned on-chip SPM banks.

Many authors have studied the memory allocation problem in MPSoCs. The main focus of their research work is data parallelism in the context of homogeneous multiprocessor systems. Meftali et al. [50] formulated the memory allocation problem as an ILP to obtain an optimal distributed shared memory architecture to minimize the global cost to access shared data as well as the memory cost. Kandemir et al. [33] presented a compiler-based strategy for optimizing energy and memory access latency of array dominated applications in an MPSoC. In [61], the authors proposed an ILP formulation for the memory partitioning problem on MPSoC. Suhendra et al. [72] studied the problem of integrating task scheduling and memory partitioning among a heterogeneous multiprocessor system on chip with scratch pad memory. This is the only paper, to the best of our knowledge, that addressed this problem in an integrated approach for MPSoC. They formulated this problem as an integer linear problem (ILP) with the inclusion of pipelining. Other works [12, 32, 73] have studied issues related to task scheduling and memory partitioning.

6.5 Chapter Summary

In this chapter, we have presented an effective heuristic that integrates task scheduling and memory partitioning on multiprocessor systems-on-chip with scratchpad memory. Compared to the widely-used decoupled approach, our integrated approach further improved the results since the appropriate partitioning of SPM spaces among different processors depends on the tasks scheduled on each of those processors and vice-versa. Results on several benchmarks from *Mediabench* and *MiBench* show the effectiveness of our approach compared to the decoupled approaches.

Chapter 7
Task Scheduling and Memory Partitioning for MPSoC: Multiple Applications

In the previous chapter, we presented an integrated framework for task scheduling and SPM memory partitioning for one application executing on a MPSoC at a given time. In such case, the SPM space and the processor cores available are assumed to be managed by a single application. Although the one-application scenario is common, some multiprocessors system on chips are used to execute multiple applications. In this case, multiple applications need to share the available resources, i.e., SPM and processors.

This chapter focuses on the multiple application scenario and develops an effective framework for task scheduling and memory partitioning for multiprocessor system on chip with multiple concurrent applications. The developed solution is a two-level solution where in the first level the available resources are dynamically partitioned among the available applications, and in the second level, the tasks in a single application are then scheduled based on the integrated approach of task scheduling and memory partitioning presented in the previous chapter.

Most of the previous works in this area [27, 64, 63, 76, 77, 23, 34, 49] focused on single application scenarios where the available resources are assumed to be managed by only one application at any given time.

7.1 Motivation

The heuristic in the previous chapter cannot directly be applied to the case of multiple applications since the execution start time of those applications may be different. Consider a simple example of three applications A, B and C to be executed on an MPSoC. Let the first application A start executing at time t_i. Suppose, then at time $t_i + s$, application B starts executing. Let A finish executing at time $t_i + n$ (where $n > s$). Also, let application C start executing some time after A finishes, i.e., C starts at time $t_i + m$ where $m > n$. Here, it is clear that not all applications are executing at the same time. At first only one application (A) is using the resources (processors and memory). Starting at time $t_i + s$, two applications A and B are competing for the available resources. Thus, some of the resources already allocated to application A will now be allocated to application B. Then during the time range, i.e., during the open-onterval $(t_i + n, t_i + m)$, all the resources can be allocated to only application B since B is the only active application in the system. Then at time t_i + m some of the resources will be deallocated from application B and allocated to application C. After B finishes executing, all the resources can now be allocated to application C as needed. From this example, the necessity for allocation and deallocation is clear since not all the applications may start and end at the same time.

The heuristic for task scheduling and SPM memory partitioning presented in the previous chapter will be applied repeatedly to each application based on the resources allocated to that application. These resources are dynamic based on how many applications are using the system at a certain time and thus the allocation of the resources among tasks inside a single application is dynamic and based on the active applications at that point in time.

7.2 Architectural Model

The architectural model assumed in this chapter consists of a number of processor cores and a scratchpad budget that is shared concurrently by all applications. A large off-chip memory of unlimited size is assumed. Each executing application in our model is mapped to a set of resources. The number of cores mapped to a certain application depends on the structure of the application and the degree of potential parallelism. A simple view of our model is presented in Figure 7.1, where a set of processors are mapped to each application and then the tasks of such application are scheduled on an available processor in the set. As mentioned earlier, the schedule is SPM budget-aware, i.e., the schedule considers the dynamic execution time of a task based on the processor to which it is mapped as well as the SPM memory budget assigned to that processor.

7.3 Our Approach

Problem Definition: *Given (i) an MPSoC architectural model of a set of processors, on-chip SPM budget, and large off-chip memory and (ii) a set of applications to be executed at this architecture with unknown start time, dynamically divide the processor cores and the SPM budget among all*

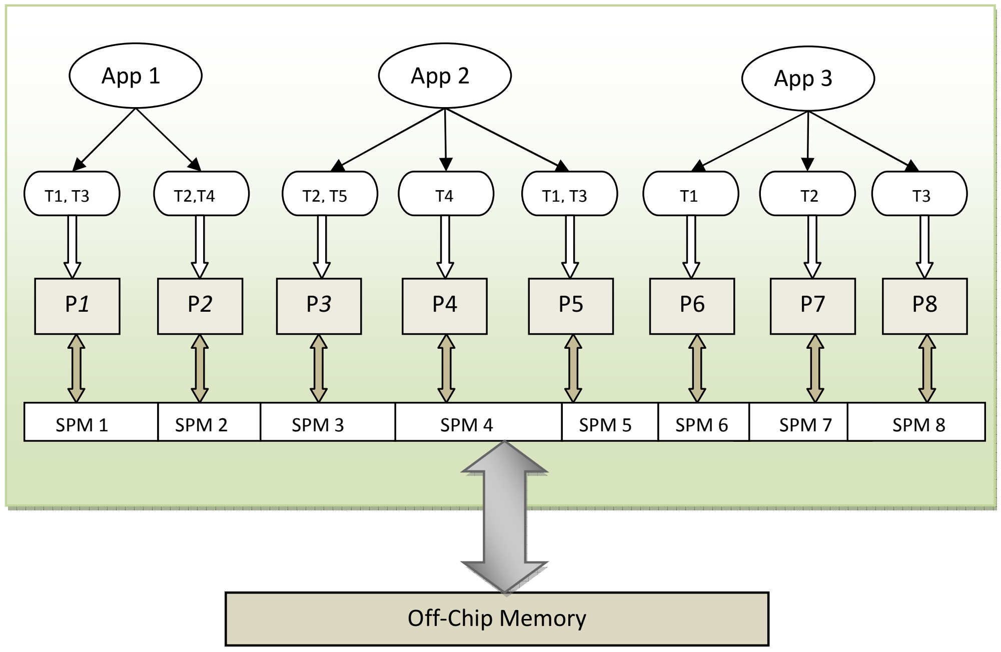


FIGURE 7.1. An architectural model example with three applications, eight processors, an SPM budget, off-chip memory, and interconnection buse.

concurrently executing applications then divide the resources mapped to each application among its tasks. The objective is to minimize the run time over all applications, that is, construct a schedule of minimum time.

The question is how to divide the available processors and memory budget among different applications at different times. The number of processors and the SPM memory budget allocated to an application depends on the nature of the application. More processors will usually be allocated to an application that has larger potential for parallelism compared to an application of a more sequential nature.

Our framework shown in Figure 7.2 consists of three major components: (i) the compiler, (ii) the resource partitioner, and (iii) the scheduler. The compiler is responsible of profiling a certain application. Once the compiler receives a new application, it will extract a set of information that will be later used by the resource partitioner. Once the resource partitioner receives such information, it will allocate resources of processor cores and SPM budget to such application based on the nature and structure of the application as well as the number of applications currently using the system. Then the scheduler will further partition the processors and SPM budget assigned to a

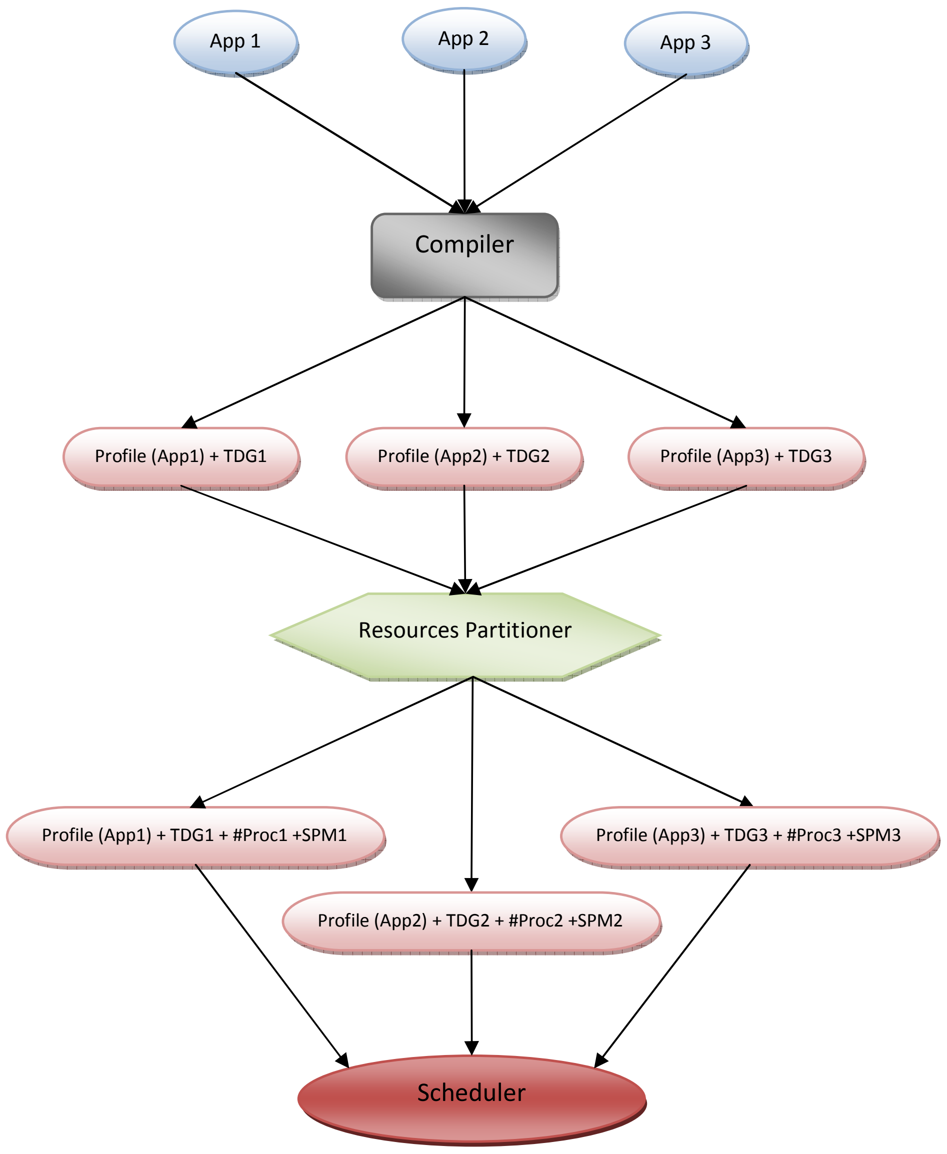


FIGURE 7.2. Our framework consists of three components: the compiler, the resources partitioner, and the scheduler.

certain application among the set of tasks that constitute such application. The scheduler will use the heuristic presented in the previous chapter.

The number of resources assigned to a certain application is based on the structure of the application. For instance, it makes sense to assign more processors to a highly parallel application in nature so that more tasks can run in parallel. Also, typically a memory-intensive application should be assigned a bigger scratchpad memory budget compared to an application that requires less memory. Note that a memory-intensive application is one in which memory accesses are a significant fraction of the execution time; whereas, an application where most of its computation time is consumed for instance in big loops rather than memory accesses is classified as non-memory-intensive.

The processor cores and the scratchpad memory budget are shared resources and thus they need to be carefully partitioned among the competing applications in the system. Once this partitioning is over, the scheduler will schedule the tasks of a certain application on the processors mapped to that application under the memory budget mapped to it. Notice that as an application enters or leaves our system, the resources will be redistributed and the tasks will be rescheduled and thus the framework is very dynamic based on the applications using the system at a certain point in time.

7.3.1 The Compiler

Once the compiler receives a new application that needs to be scheduled on the multi-processor system on chip, it analyzes the structure of this application and extracts important information that will be sent to the resource partitioner and the scheduler as annotations. For example, the task dependence graph is a very basic part that is needed by the resource partitioner and the scheduler. To refresh our memories, a task dependence graph (*TDG*) is a directed acyclic graph with weighted edges where each task in the embedded application is represented by a vertex. An edge between two tasks, say T_i and T_j in the *TDG*, represents some kind of a scheduling order due to the fact that T_j needs data to be transferred from T_i after T_i is already executed. Thus a certain processor cannot start executing task T_j unless all the necessary data communication is performed.

Another important piece of information extracted by the compiler is a set of [Max_{ij}, Avg_{ij}, and Min_{ij}] values for each of the tasks. As defined in the previous chapter, the Min_{ij} represents the computation time for task T_i on processor P_j assuming all of the available SPM budget is assigned to P_j. The Avg_{ij} represents the computation time for task T_i on processor P_j assuming $1/n$ of the available SPM budget is assigned to P_j where n is the number of processors. And the Max_{ij} represents the computation time for task T_i on processor P_j assuming no SPM budget is assigned to P_j which means that all the data variables will be accessed from the slow off-chip memory.
In summary, the profiling by the compiler is intended to:

- divide each application into computation blocks referred to as tasks;
- find the computation times for each task on each available processor in processor cycles;
- find the number of variables;
- the number of times each variable is used, $freq$; and
- the size in bytes for each variable in the current application.

The profiler information is based on a system with only off-chip memory. Using the profile information and dependence analysis, a task graph is constructed with a vertex for each task and an edge to represent the communication cost between two tasks. The communication cost depends on the size of data to be communicated between the two tasks and it is calculated through profiling.

7.3.2 The Resource Partitioner

The resource partitioner is responsible of dividing the available resources among the concurrently executing applications. Given a system of p heterogeneous processor cores and an SPM budget of size m and n executing applications, divide the available resources fairly among the applications so that the schedule time of all the applications is minimized. The partitioner will receive the profiling information from the compiler and then decide how much SPM budget and how many processor cores should be assigned to such an application. Needless to say, the scheduler will most probably assign fewer resources to an application than the optimal resources required as we are assuming a system of limited resources and with more than one concurrently executing application competing for the available resources.

Once the partitioner receives a new application, it will read its structure and computes the level of parallelism based on the structure of the *TDG* to figure out the degree of benefit from assigning more processors to such an application. Also, based on the *elasticity* value, the scheduler will be able to figure out how much this application benefit from more SPM budget. As in Chapter 6, we define *elasticity* of a task as the extent to which this task can benefit from a larger SPM. We define *elasticity* dynamically as the extent to which the computation cost of a task on P_i may decrease as the SPM budget of P_i is increased from the current budget to *size* where *size* is the maximum amount of SPM budget available in our model. Equation 7.1 defines *elasticity* of task T_i where *Cur* is the computation time of the task under the current memory budget. The *elasticity* of a task T_i is basically a measure of the room for computation time reduction of T_i with more SPM budget. A bigger value of *elasticity* means that the computation time of T_i is more amenable for reduction with the increase in the SPM allocated to that task. Note that $elasticity(T_i)$ is a dynamic value since the current computation time of T_i, Cur_i, may change as the SPM budget distribution changes.

$$\text{elasticity } (T_i) = \frac{Cur_i - Min_i}{Cur_i} \tag{7.1}$$

As mentioned earlier, the number of processor cores mapped to a certain application is directly related to the degree of parallelism (DP) of such an application. The degree of parallelism is defined in Equation 7.2. A large DP value means that the application's structure has high degree of parallelism where as a small DP value means that the application is more sequential in nature. The degree of parallelism of an application will be extracted from the structure of its task dependence graph. We define the DP in a way to reflect the degree of parallelism between the tasks in the TDG. Two independent tasks can be executed in parallel on two different processors (if possible) whereas two dependent tasks A and B (B depends on A) must be executed sequentially as task B needs to wait for some input information from task A.

Given a *TDG* with t tasks. A pair of tasks (Ti, Tj) can be run in parallel if there is no path in the *TDG* between Ti and Tj and thus they are independent. As the number of such pairs is bigger, the degree of parallelism of the application is bigger. The DP value will be used by the resource partitioner as a guideline when dividing the processors in the system among the concurrently executing applications.

$$DP(APP_i) = paths_i + \frac{pairs_i}{paths_i} \tag{7.2}$$

Given a TDG, find all the paths between a dummy start node and a dummy end node. The dummy start node S is a node with an edge to each task (node) in the TDG with zero incoming edges and the dummy end node E is such that there is an edge between this node and all the tasks in the TDG with no outgoing edges. Two paths are distinct if they have at least one task not in common. For any two paths p_i and p_j, find all the pairs (T_i belongs p_i, T_j belongs p_j) such that T_i and T_j can be run in parallel. As shown in Equation 7.2, the degree of parallelism is made up of the sum of two components. The first component represents the number of distinct paths in the TDG. Intuitively speaking, an application with a TDG of high number of distinct paths is more parallel in nature and thus it can benefit more from additional processors. However, the number of distinct paths is not enough to represent the degree of parallelism in an application.

An application with two more balanced paths will benefit more from two processors compared to an application with two unbalanced paths. Consider for instance the two TDG examples in Figure 7.3. The two applications corresponding to these two TDGs have the same number of tasks. Now assume that two processers are assigned to the TDG in Figure 7.3 (a). After processor 2 is used to execute (lets say) task 6, that processor is no more needed and thus it will be idle. However, for the application corresponding to the TDG in Figure 7.3 (b), the two processors will be more effectively utilized during the executing time of the application and thus such an application should have a higher DP.

To take this observation into consideration, we added the second part, $\frac{pairs}{paths}$, to our DP definition. The value *pairs* is the number of pairs that can be executed in parallel. Considering again the two TDG examples in Figure 7.3. The parallel pairs in Figure 7.3 (a) are (1, 6), (2, 6), (3, 6), (4, 6), and (5, 6) whereas the parallel pairs corresponding to the TDG in Figure 7.3 are (1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), and (3, 6). Thus the DP value for the first TDG is 2 + 5/2 =

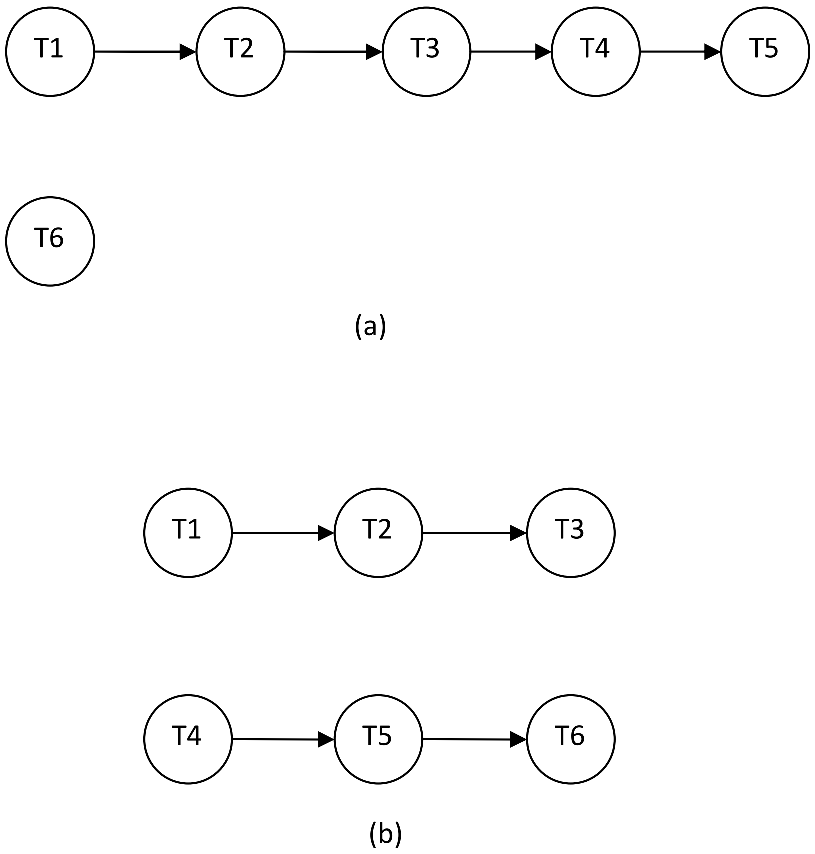


FIGURE 7.3. (a) An unbalanced TDG. (b) A balanced TDG

4.5 and the DP value for the second TDG is 2 + 9/2 = 6.5. As a result, the ratio pair/paths loosely represents how balanced the paths in the TDG are. Now in an architectural model with 3 available processors, the resource partitioner will use the two DP values to decide that only one processor will be assigned to the first application whereas two processors will be assigned to the second application. We will see later on how the resource partitioner will decide on how many processors to assign to each executing application.

Although the *pairs* value represents the number of parallel paths, that number is mostly an exaggeration of the number of actual tasks that can run in parallel. To clarify this point, assume that we have the following pairs of tasks between two paths p_i and p_j (Ti belongs p_i, Tj belongs p_j), (T1, T3), (T1, T4), and (T1, T5). What this means is that task T1 that belongs to path *Pi* can run in parallel with T3, T4 or T5. Since T3, T4, and T5 belong to the same path p_j, they are dependent and should be run sequentially. Thus, T1 can be run in parallel with only one of such tasks at a time. As a result, the *pairs* variable in our DP definition is in no way a reflection of how many pairs can run in parallel at the same time but rather a reflection of how much the paths are balanced in a certain *TDG*.

The processors will be divided among all the applications in the system in a fairly fashion so that the overall run time of the scheduler is minimized. As will be discussed in Subsection 7.4.4, each application will receive a number of processors proportional to its number of distinct paths and its degree of parallelism DP.

The SPM budget in the system will be partitioned among the concurrently executing applications in the system in a fairly fashion so that the overall computation time of the scheduler is minimized. If the system does not have enough SPM budget to cover all the variables in the system, then each application will receive an SPM budget proportional to its needed SPM as well as to its elasticity. This concept will be discussed in the Subsection 7.4.4. The elasticity of an application is the average elasticity of its tasks and is defined in Equation 7.3.

$$elasticity(App_i) = \frac{\sum_j elasticity(T_j)}{n} \tag{7.3}$$

7.3.3 The Scheduler

The scheduler receives all the applications with their structure and assigned number of cores and SPM budget from the resource partitioner. Basically, for each application i, the scheduler will receive its task dependence graph, TDG_i, the cores mapped to this application P_i, and its SPM budget S_i as $App_i(TDG_i, P_i, S_i)$. The scheduler then schedules the tasks of each application i on the resources assigned to it; i.e., at this stage, the tasks will be mapped to the available processors P_i and the SPM budget S_i will be partitioned among the processors so that the overall computation time is minimized.

The scheduler will follow the heuristic presented in Chapter 6 (also in Figure 7.4). The main idea of this heuristic, as discussed in Chapter 6, is that it integrates the scheduling part with the SPM budget partitioning part in one step. Unlike the traditional scheduling heuristics that assign the task to the processor such that the increase in the computation time under the current SPM budget distribution is minimum, the heuristic will schedule a task T_i on a processor P_l of higher overhead time provided that the predicted end computation time ($PEC(P_l)$) (defined by us in Equation 7.4) of this processor is at least δ % less than that of P_j. In this sense, the heuristic looks beyond the current SPM budget distribution to predict what will happen with future SPM repartitioning among the processors. As discussed in Chapter 6, the predicted estimated computation time (*PEC*) is basically an estimate of how much the end time of processor P_i will be if more SPM budget is assigned to processor P_i. The *PEC* of a processor is closely related to the *elasticity* of the tasks scheduled on that processor. The *PEC* value provides the dynamic essence of this heuristic as at each step the heuristic looks beyond the current SPM budget distribution in its task mapping decision to an estimate of future distribution in future steps.

$$PEC(P_i) = End\_time(P_i) - \sum_{T_j \in P_i} (Cur(T_j) - \frac{Cur(T_j)}{1 + elasticity(T_j)}) \tag{7.4}$$

In the case of multiple applications executing concurrently on an MPSoC, the scheduler will schedule the tasks for each application separately following the single application heuristic. The question now is what happens when the system gets a new application or one application finishes executing and leaves the systems. As mentioned in the previous subsection, when a new application wants to use the system, the resource partitioner will repartition the resources among all the executing applications guided by the level of parallelism of an application and its elasticity. At this point, the scheduler will receive a new set of information about the applications in the system as well as the new application. Some of the applications that are already scheduled under the previous resource partitioner need to be rescheduled according to the new resource partitioning.

Some of the resources that will be assigned to the new application will come from the unused cores and SPM budget whereas others will come from resources already assigned to some other already executing applications. Applications may lose some processor cores and a part of the SPM budget assigned to them under the previous resource partitioning prior to the arrival of the new application. The scheduler, however, will wait for all the tasks currently executing to finish before redistributing the resources. A new schedule will be generated based on the new resources assignment and on the updated TDG for each application. The updated TDG of a certain application is the TDG including only non-scheduled tasks.

The same thing will happen when an application finishes executing. The resources that were assigned to this application need to be distributed among other still executing applications. When an application finishes, the scheduler will inform the resource partitioner which will, in its part, repartition the available resources among the currently executing applications and will then send the new resource assignment to the schedule that will generate a new schedule.

7.3.4 Our Heuristic

The main task of our heuristic is to partition the resources among different applications and then schedule the tasks on these resources such that the overall time of the schedule is minimized. The resource partitioner heuristic is mainly made up of two major parts, the SPM partitioner (see Figure 7.5), and the processors partitioner (see Figure 7.6). The job of the partitioner is to partition the available resources among the concurrently executing applications in as fairly fashion as possible.

In the SPM partitioner case, the heuristic operates in two steps. In the first step, it determines the memory requirements of the applications from the annotations sent by the compiler and then in the second step, it allocates the available SPM memory space to the applications. The SPM space

```
Scheduler($App_i$, $S_i$, $P_i$, $TDG_i$)

1.  Divide the SPM memory equally between the processors.
2.  For each task $T_i$ and processor $P_j$, extract the following:
3.      (i) Minimum computation time on $P_j$, $Min_{ij}$.
4.      (ii) Maximum computation time on $P_j$, $Max_{ij}$.
5.      (iii) Average computation time on $P_j$, $Avg_{ij}$.
6.  Find ASAP for all the tasks based on $Avg$ values.
7.  $L_1$ = List of tasks in increasing order of ASAP.
8.  While ($L_1$ not empty) do:
9.      Get the first task $T_f$ from $L_1$.
10.     For each processor $P_i$:
11.         Calculate the elasticity and PEC of $P_i$ if $T_f$ is mapped to $P_i$.
12.         Find the minimum start time of $T_f$ on $P_i$.
13.         Find $END\_time(P_i)$ if $T_f$ is mapped $P_i$.
14.         if (($END\_time(P_i) < min$ && $PEC(P_j) \geq (1 - \delta\%)PEC(P_i)$)||
                ($END\_time(P_i) > min$ && $PEC(P_i) \leq (1 - \delta\%)PEC(P_j)$))
15.             $min = END\_time(P_i)$
16.         else if ($END\_time(P_i) = min$)
17.             $min = END\_time$ of processor with the higher elasticity.
18.     End For
19.     Assign $T_f$ to $P_j$ corresponding to $min$.
20.     Delete $T_f$ from $L_1$.
21.     Call Balance().
22. End While
23. For i = 1 to $t$ do:
24.     Call Balance().
```

FIGURE 7.4. Our scheduler heuristic.

```
SPM_Partitioner(n, m)
______________________________________________
1.  For i = 1 to n do:
2.    SPM = SPM + SPM_requested(i)
3.  End For
4.  If (SPM ≤ m)
5.    For i = 1 to n
6.      SPM_received(i) = SPM_requested(i)
7.      SPM_Elasticity = SPM_Elasticity + (1 + elasticity(i)) * SPM_requested(i)
8.    End For
9.  Else
10.   For i = 1 to n
11.     SPM_ received(i) = MIN(SPM_requested(i),
                        ((1+ elasticity(i)) * SPM_requested(i)/(SPM_Elasticity) * n))
12.     SPM_Elasticity = SPM_Elasticity - SPM_ received(i)
13.   End For
```

FIGURE 7.5. Our SPM partitioning heuristic.

```
Processor_Partitioner(n, p)
______________________________________________
1.  For i = 1 to n
2.    Reconstruct_TDG(i)
3.    DP(i) = Compute_DP(i)
4.  End For
5.  For i = 1 to n
6.    Path = Path + path(i)
7.    Path_DP = Path_DP + (1 + α DP(i)) * path(i)
8.  End For
9.  If (Path ≤ p)
10.   For i = 1 to n
11.     Processor_received(i) = path(i)
12.   End For
13.  Else
14.   For i = 1 to n
15.     Processor_received(i)= MIN(path(i),((1 + α DP(i)) * path(i)/(Path_DP) * p))
16.     Path_DP = Path_DP - Processor_ received(i)
17.   End For
```

FIGURE 7.6. Our processor partitioning heuristic.

```
Task Scheduling Memory Partitioning(n, m)
________________________________________
1.  Processor_Partitioner(n, p)
2.  SPM_Partitioner(S, p)
3.  Scheduler($App_i$,$S_i$, $P_i$, $TDG_i$)
```

FIGURE 7.7. Our task scheduling memory partitioning heuristic.

allocated to each application is then divided amongst its tasks based on the elasticity of such tasks. The SPM partitioner takes the SPM size (*m*) and the number of applications concurrently running (*n*) as an input. Based on the data requirements of each application obtained by the function call *Receive(),* the heuristic calculates the total SPM budget, *SPM*, needed to satisfy all the available applications. If this *SPM* value is less than or equal to the available SPM budget (*m*), then each application takes all the SPM space it requested. In this case, we do not have a space partitioning problem.

However, if the total requested *SPM* is less than the available SPM budget, the SPM partitioner tries to distribute the available SPM among all the concurrently executing applications in a fairly possible way. It does that in Lines 10–13 in the heuristic by allocating to each application an SPM budget proportional to its requested size in a manner that an application with higher elasticity will receive an SPM budget closer to it requested budget compared to an application with smaller elasticity. Remember that the overall objective is a short schedule. From Lines 10–13, an application with higher elasticity value will get a push up in presenting its case of receiving more SPM budget.

The processor partitioner heuristic in Figure 7.6 works in a similar fashion to the SPM partitioner. The processor partitioner takes the number of processors in the system (*p*) and the number of applications concurrently running (*n*) as an input. It first sums up the total number of requested processors by all the applications, namely the value *Path*. We assume that the optimal number of processors for an application is equal to the number of distinct paths in its corresponding TDG. If the total number of processor requested (*Paths*) is less than or equal the number of available processors, then each application will be granted a number of processors equals to its number of distinct paths.

However, in the case that the *Paths* value is less than the available processors (p), then the partitioner will distribute the available processors among the applications proportional to their number of paths. This core distribution will be performed in a way that an application with a higher degree of parallelism (DP) will receive a number of cores closer to its number of distinct paths compared to an application of a smaller DP value, lines (14 - 17). A good value for α in Figure 7.6 is 0.1. Two applications with the same number of paths may end up receiving different number of processors based on the DP values. The application with the higher DP value will probably receive more processors compared to the one with smaller DP value. This is fair as the DP value loosely reflects the degree of balance between all the paths of an application. A more balanced-path application will more effectively utilize the processors as more tasks will be executing in parallel.

Our heuristic for the complete task scheduling memory partitioning framework is presented in Figure 7.7. This heuristic will be called any time a new application enters or leaves the system.

7.4 Chapter Summary

In this chapter, we have presented an effective heuristic that integrates task scheduling and memory partitioning on multiprocessor systems-on-chip with scratchpad memory in the presence of multiple applications. Compared to the widely-used decoupled approach, our integrated approach will further improve the results since the appropriate partitioning of SPM spaces among different processors depends on the tasks scheduled on each of those processors and vice-versa. The framework fairly divides the available resources among all the concurrently executing applications. The techniques are very dynamic depending on the applications concurrently using the system.

Chapter 8
Conclusions and Future Work

Applications that are executed on embedded processors have become extremely important. The amount of memory on embedded processors has not kept pace with the increase in the size of embedded applications. With embedded processors such as digital signal processors (or DSPs), effective compilation techniques are one way to reduce memory requirements. With multiprocessor system-on-chip (MPSoC), issues such as mapping, memory management and scheduling are important. In this thesis, we have addressed different compilation issues for DSPs and scheduling techniques for MPSoCs.

8.1 Conclusions

The main concern of the first part of this thesis in Chapters 2 thru 5 is on compilation techniques for digital signal processors (DSPs). DSPs are special-purpose processors that are widely used in many embedded systems. Such systems have very tight constraints in terms of size, memory, cost, etc. Memory usually constitutes a large fraction of the cost and size of an embedded system. Thus, reducing the memory requirement of an application is essential in embedded systems. In the first part of the thesis, we studied and presented different approaches and heuristics to reduce the code size and the memory requirement to store the variables. Reducing the code size automatically reduces the silicon area needed to save this code as in such DSP systems the code is stored in a ROM. Increasing the proximity of the embedded application variables (including the temporary variables) in the memory is critical to reduce the code size as well as the stack size.

In Chapters 2 and 3, we addressed the problem of offset assignment with variable coalescing (COA). COA is a method used to decrease the size of the code. Decreasing the size of the code is essential in such systems as the size of the code directly translates into silicon area. Studies have shown that up to 50% of the program bits are used for addressing. Decreasing the number of address arithmetic instructions is equivalent to decreasing the memory and size of such systems.

DSPs usually have a dedicated address generation unit (AGU) that can be utilized to update the address with the current load/store instruction. When there is a need to update the current address d to $d \pm r$ where r is in the auto-modify range, then no addressing instruction is needed. Offset assignment is a method used to maximally utilize auto-increment and decrement modes in such systems to decrease the code size.

We studied the problem of offset assignment with variable coalescing as a single offset assignment (CSOA) when there is only one address register in Chapter 2 and as general offset assignment (CGOA) in Chapter 3 when there are multiple address registers. Our CSOA heuristic is used as the basis for our CGOA heuristic. Results on several *MediaBench* benchmarks show the significant benefit of our techniques compared to the best known techniques in the literature. The results were further improved through the utilization of simulated annealing (SA).

In Chapters 2 and 3, the importance of variable coalescing was demonstrated through our results on large real life benchmarks. It was assumed in some of the previous works [47] that variable coalescing is not effective in decreasing the code size through minimizing the number of address arithmetic instructions. To clarify this point, one needs to understand the relationship between edge selection and variable coalescing. Recall that edge selection or variable coalescing should respect the two conditions mentioned in Chapter 2 and 3, namely, (i) the access graph is still acyclic, and (ii) no node in the access graph has more than two selected edges incident at it. Thus an edge selection may prevent some variable coalescing opportunities and vice-versa. However, due to the large number of temporary variables, many coalescing opportunities are available that minimized the cost drastically on the larger real life benchmarks used in our experiments.

In Chapter 4, an optimal integer linear programming (ILP) solution is presented for the offset assignment problem with variable coalescing. The ILP formulation is then extended to include variable (or operand) permutation. Variable permutation is used to find the best possible access sequence so that the overall cost is minimal. A new version of the general offset assignment problem is also suggested where the main idea is to partition variable accesses rather than variables. In this new CGOA approach, different instances of the same variable can be accessed by different address register. The optimal solution to the new GOA is at least as good as the traditional GOA as the latter is a special case of the former.

In Chapter 5, we studied the problem of array register allocation with the intention to decrease the code size as well as decrease the number of address registers used. This problem is utilized for array-intensive applications, which are very common in embedded systems. Optimal integer linear formulations (ILP) are presented for two related problems: (i) finding the minimum number of address arithmetic instructions for a given number of address registers in the system, and (ii) finding the minimum number of address registers for a zero cost. Due to the exponential complexity of the ILP, very long computation time is needed with big embedded applications. To overcome this, a genetic algorithm is utilized to get near optimal solutions in a reasonable amount of time.

In the second part of the thesis (Chapters 6 and 7), we shifted our attention to multi-processor system on chip (MPSoC). The continued increase in the complexity of applications ported to SoC architectures places a tremendous burden on the computational resources needed to deliver the required functionality. An emerging architectural solution places multiple processor cores on a single chip to manage the computational requirements. In Chapters 6 and 7, we presented an integrated approach to the task scheduling and memory partitioning problems in order to improve the quality of the schedule.

In Chapter 6, we developed a memory-aware task scheduling heuristic for embedded applications on an MPSoC. There, we assumed that only one application consisting of multiple tasks is executed at a given time on the MPSoC system. Usually scratch-pad memory (SPM) partitioning and task scheduling are studied as two separate problems; however, we showed through extensive results on real life benchmarks that those two problems are inter-related and thus they should be studied as one problem. Our integrated task scheduling and memory partitioning heuristic for applications on an MPSoC outperformed the decoupled techniques and was very close to the optimal solution in most cases. Pipelining was also studied to further improve the results where the main objective is to minimize the time needed for a pipeline stage.

In Chapter 7, we extended our task scheduling/SPM partitioning problem from Chapter 6 to handle the case of multiple applications utilizing the system at the same time. Those applications compete for the available resources (processor cores and SPM budget). We presented a two level approach to solve this problem. In the first level, the partitioner distributes the processors and the SPM budget among the concurrently executing applications based on the structure of each application. More processors are assigned to a highly parallel application and more SPM budget is given to a more memory-intensive application. Then, in the second part, the scheduler schedules the tasks of each application based on the technique presented in Chapter 6. The technique presented is dynamic as the number of applications using the system may change at different instances of time. Thus at any given time, a new application starts executing or an application ends executing, allocation/deallocation of the resources is performed.

In summary, our offset assignment approaches are the best approaches in the literature so far. We showed that throughout the thesis chapters through extensive experimental results on real life applications. We are the first to present optimal solution to the offset assignment problem with variable coalescing and with the inclusion of operand (i.e., variable) permutation. We introduced a new approach to the general offset assignment problem that partitions the variable instances rather than the variables. We presented a heuristic approach to this problem that can be further developed and implemented.

We also presented the first heuristic in the literature that performs integrated task scheduling and memory partitioning. We showed the importance of such approach compared to decoupled techniques. We are the first to present a way to fairly partition the available processor cores and SPM budget among concurrently executing embedded applications on a multi-processor system on chip. We believe that the problems studied in this thesis are essential to improve the effectiveness of embedded systems and that the implementation of such approaches in commercial compilers can be highly beneficial.

8.2 Possible Future Work

8.2.1 DSPs: The Offset Assignment Problem

Simulated Annealing In Chapters 2 and 3, a special simulated annealing (SA) algorithm is used. In our approach, the final solution from our heuristic is used as the initial solution to our SA. SA is run for a short period of time with a low probability of accepting bad solutions. The main reason for this choice is that the SA usually takes a long time especially when tested on big benchmarks. However, such a solution is not the real SA but rather closer to a downhill greedy solution. One possible future work is to solve the offset assignment problem using the traditional simulated annealing. The initial solution for this SA is a random solution and the probability of accepting a bad solution at the beginning is not very small so that the SA will be able to search the whole search space rather than ending up with a premature solution in some local optima.

Heuristics for permuting variables in offset assignment In Chapter 4, an ILP solution is presented to solve the offset assignment problem with variable permutation. The computation time of such formulation is usually exponential and thus it may become inconvenient to use for large embedded applications. One possible future work is to develop a heuristic based solution to the offset assignment problem with variables permutation. Such solution should take into consideration the structure of the access graph and try to apply variable permutation to change the AS into one with smaller cost.

Incorporating statement reordering in the ILP formulation The ILP formulation in Chapter 4 considers only variable permutation. Statement reordering is not considered. Statement reordering can be an effective approach to further improve the results. An ILP model that takes both variable permutation and statement reordering into account can be a possible future work. Statement reordering is usually based on the dependence analysis so that the solution is legal.

A new approach to general offset assignment In Chapter 4, a new approach to the general offset assignment problem is presented. In this new approach, the access sequence is partitioned based on partitioning the variable accesses rather than the variables. Partitioning the variables means that a variable can be accessed by one and only one address register. However, partitioning the accesses allows two accesses of the same variable to be addressed by different address registers. An ILP solution is presented to this new CGOA. One possible future work is to construct a heuristic solution based on the approach presented in Chapter 4 and to perform extensive testing on different benchmarks to show how powerful this new approach is compared to the traditional CGOA solution.

8.2.2 MPSoCs: Memory-Aware Scheduling and SPM Management

Data allocation and experimental evaluation in MPSoC scheduling In Chapter 6, we studied the task scheduling problem for embedded applications on an MPSoC concurrently with SPM memory partitioning. In scheduling such applications, we assumed a simplistic data allocation technique. Our data allocation technique follows the increasing order $byte/freq$ of the data variables where $byte_i$ is the size of the variable v_i and $freq_i$ is the number of times v_i is accessed. A more involved data allocation technique is expected to further improve the results. In Chapter 7, we presented a framework for resource partitioning and task scheduling for multiple concurrently executing applications in a multi-processor systems on chip system. Experimental evaluation of this framework is on our agenda as a future work.

Scratchpad memory management The trend nowadays in embedded systems (and some heterogeneous multicore designs such as the IBM Cell processor) is to use fully software-controlled fast memories known as *scratchpad memories (SPMs).* SPMs demonstrate many benefits that ranges from predictability of access time, to low energy consumption. Being of limited size, usually not all the variables in the program can be allocated to such memories at the same time and thus a big chunk of the variables will reside in the slow main memory. Accessing variables from the main memory is usually within the range of hundred times slower than accessing them from the SPM. Hence, the proper allocation of variables to the SPM can make a big difference in the speed of the application execution. One possible future work is to address the problem of deciding which variables should be brought to the SPM and at what time during the execution. The choice of the variables or the chunk of arrays elements to be brought to the SPM should be based on the reuse distance, i.e., heavily reused variables should have higher priority to be in the SPM. Another problem is whether the variable allocation should be static or dynamic. Static allocation means that the variables allocated in the SPM will stay there throughout the program execution. Dynamic allocation gives the choice of allocating the variables to the SPM for a certain amount of time. Since a program may access non-contiguous elements of an array in a certain iteration, bringing the whole column (row) of arrays to the SPM at a time means that we may be bringing many array elements that will not be used in this iteration or a nearby (in time) future iteration. A way to solve this problem is to introduce local arrays and to map only the used elements of the original arrays in this iteration to the local arrays. When such arrays are brought to the SPM, most of their elements will be used in the current iteration and thus the SPM memory will be effectively utilized.

Locality enhancement Another issue for MPSoCs is management of locality in applications. Locality enhancement in a certain program means making accesses to reused elements closer in time through reducing the reuse distance. Smaller reuse distance means that heavily reused elements will be brought from the main memory to the SPM/cache in a certain iteration and thus more hits and less misses will occur when executing the program. The main idea is to maximize the access of variables from the fast SPM memories and to minimize the slow trips to the main memory. A related important aspect is exploiting parallelism. A possible future work is to explore building a framework for automatic locality enhancement and program parallelization based for MPSoCs.

References

[1] Ilog inc., ilog cplex 8.1 reference manual. *http://www.ilog.com/products/cplex*.

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

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

[4] Nomadik: A multimedia application processor. *http://www.st.com*.

[5] Offsetstone. *http://www.address-code-optimization.org.*

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

[7] G. Araujo and S. Malik. Register allocation for indirect addressing in loops. 8(1), 1999.

[8] S. Atri, J. Ramanujam, and M. Kandemir. Improving offset assignment for embedded processors. *Languages and Compilers for High-Performance Computing, S. Midkiff et al. (eds.), Lecture Notes in Computer Science, Springer-Verlag*, 2001.

[9] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[10] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1), 2002.

[11] F. Balasa, F. Catthoor, and H. Man. Background memory area estimation for multidimensional signal processing systems. *IEEE Transactions on VLSI Systems*, 3(2):157–172, 1995.

[12] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *International Conference on Hardware-Software Codesign (CODES)*, 2002.

[13] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software-Practice and Experience*, 22(2):102–111, 1992.

[14] A. Basu, R. Leupers, and P. Marwedel. Array index allocation under register constraints in dsp programs. *12th Int. Conf. on VLSI Design*, 1999.

[15] L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation and scheduling for mpsoc via decomposition and no-good generation. In *International Joint conferences on Artificial Intelligence (IJCAI)*, 2005.

[16] F. Boesch and J. Gimpel. Covering the points of a digraph with point-disjoint paths and its application to code optimization. In *Journal of the ACM*, 1977.

[17] K. S. Chatha and R. Vemuri. Hardware-software partitioning and piplined scheduling of transformative applications. *IEEE Transactions on VLSI*, 10(3), 2002.

[18] G. Chen and M. Kandemir. Optimizing address code generation for array-intensive dsp applications. In *Proc. International Symposium on Code Generation and Optimization*, 2005.

[19] G. Chen, M. Kandemir, M. J. Irwin, and J. Ramanujam. Reducing code size through address register assignment. *ACM Transactions on Embedded Computing (TECS)*, 5(1):225–258, 2006.

[20] Y. Cho, N.-E. Zergainoh, S. Yoo, A. Jerraya, and K. Choi. Scheduling with accurate communication delay model and scheduler implementation for multiprocessor system-on-chip. *Design Automation for Embedded Systems*, 2007.

[21] M. Collin, R. Haukilahti, M. Nikitovic, and J. Adomat. Socrates a multiprocessor soc in 40 days. In *Proceedings of Design Automation and Test in Europe (DATE)*, 2001.

[22] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 2005.

[23] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st annual conference on Design automation*, 2004.

[24] J. G. Ganssle. The art of programming embedded systems. *Academic Press Inc., San Deigo, California*, 1992.

[25] D. Gaski and R. H. Kuhn. Guest editors introduction-new vlsi tools. *IEEE Computer*, 16(2):14–17, 1983.

[26] C. Gebotys. Dsp address optimization using a minimum cost circulation technique. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1997.

[27] P. Hanlai, L. Ming, and J. Jing. Extended control graph based performance optimization using scratch-pad memory. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2005.

[28] J. L. Hennessy and D. A. Patterson. Computer architectures: A quantitative approach. *Morgan Kaufmann*, 1996.

[29] J. Hong. Memory optimization techniques for embedded systems. *Ph.D. Thesis, Dept. of Electrical and Computer Engineering, Louisiana State University*, 2002.

[30] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. In *Journal of the ACM*, 1973.

[31] J. Huynh, J. N. Amaral, P. Berube, and S. A. Touati. Evaluation of offset assignment heuristics. *Proceedings International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2007.

[32] M. Kandemir and N. Dutt. Memory systems and compiler support for mpsoc architectures. *Multiprocessor Systems-on-Chips*, 2005.

[33] M. Kandemir, J. Ramanujam, and A. Choudhury. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Design Automation Conference (DAC)*, 2002.

[34] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Conference on Design Automation*, 2001.

[35] S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi. Optimization by simulated annealing. 220(4598):671–680, 1983.

[36] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. In *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, 1999.

[37] S.-R. Kuang, C.-Y. Chen, and R.-Z. Liao. Partitioning and pipelined scheduling of embedded systems using integer linear programming. In *International Conference on Parallel and Distributed Systems (ICPADS)*, 2005.

[38] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3), 1999.

[39] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings IEEE International Symposium on Microarchitecture*, pages 330–335, 1997.

[40] R. Leupers. Code generation for embedded processors. In *Proceedings 13th International System Synthesis Symposium (ISSS)*, 2000.

[41] R. Leupers. Offset assignment showdown: Evaluation of dsp address code optimization algorithms. In *Proceedings 12th International Conference on Compiler Construction (CC) Warsaw, Poland, Springer LNCS 2622*, 2003.

[42] R. Leupers, A. Basu, and P. Marwedel. Optimized array index computation in dsp programs. In *Proc. ASP-DAC*, 1998.

[43] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proceedings 11th International System Synthesis Symposium (ISSS)*, 1998.

[44] R. Leupers and P. Marwedel. Algorithms for address assignment in dsp code generation. In *Proceedings International Conference on Computer-Aided Design (ICCAD)*, 1996.

[45] B. Li and R. Gupta. Simple offset assignment in presence of subword data. In *Proceedings International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2003)*, pages 12–23, 2003.

[46] S. Liao. Code generation and optimization for embedded digital signal processors. *Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology*, 1996.

[47] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[48] Z. Ma, C. Wong, S. Himpe, E. Delfosse, F. Catthoor, J. Vounckx, , and G. Deconinck. Task concurrency analysis and exploration of visual texture decoder on a heterogeneous platform. In *IEEE Workshop on Signal Processing Systems (SiPS)*, 2003.

[49] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *Proceedings of the conference on Asia South Pacific design automation: electronic design and solution fair*, 2004.

[50] S. Meftali, F. Gharsalli, F. Rousseau, and A. Jerraya. An optimal memory allocation for application-specific multiprocessor system-on-chip. In *International Symposium on Systems Synthesis (ISSS)*, 2001.

[51] G. D. Micheli, R. Ernst, and W. Wolf. Readings in hardware/software co-design. *Morgan Kaufmann*, 2002.

[52] M. Mitchell. An introduction to genetic algorithms (complex adaptive systems). *The MIT Press Publisher*, 1998.

[53] S. Muchnick. Advanced compiler design and implementation. *Morgan Kaufmann Publishers Inc.*, 1997.

[54] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluating alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, 1996.

[55] R. Neimann and P. Marwedel. Hardware/software partitioning using integer programming. In *Design Automation and Test in Europe (DATE)*, 1996.

[56] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[57] D. Ottoni, G. Ottoni, G. Araujo, and R. Leupers. Improving offset assignment through simultaneous variable coalescing. In *7th International Workshop on Software and Compilers for Embedded Systems (SCOPES'03),*, 2003.

[58] D. Ottoni, G. Ottoni, G. Araujo, and R. Leupers. Offset assignment using simultaneous variable coalescing. *ACM Transactions on Embedded Computing Systems*, 5(4), 2006.

[59] G. Ottoni and G. Araujo. Efficient array reference allocation for loops in embedded processors. In *Proceedings of the IEEE Workshop on Embedded System Codesign (ESCODES'02)*, 2002.

[60] G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan, and S. Malik. Optimal live range merge for address register allocation in embedded programs. In *Proceedings 10th International Conference on Compiler Construction, CC 2001 LNCS 2027, pp. 274–288. Springer*, 2001.

[61] O. Ozturk and M. Kandemir. An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In *IEEE computer society Annual Symposium on VLSI (ISVLSI)*, 2006.

[62] O. Ozturk, M. Kandemir, and S. Tosun. An ilp based approach to address code generation for digital signal processors. *Proceedings Great Lakes Symposium on VLSI (GLSVLSI)*, 2006.

[63] P. Panda, N. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In *Proceedings of the 10th International Symposium on System Synthesis*, 1997.

[64] P. Panda, N. Dutt, and A. Nicolau. Efficiant utilization of scratch-pad memory in embedded applications. In *Proceedings of the European Conference on Design and Test*, 1997.

[65] P. Panda, N. Dutt, and A. Nicolau. Memory issues in embedded systems-on-chip: Optimization and exploration. *Kluwer Academics Publisher*, 1999.

[66] P. Panda, N. D. Dutt, and A. Nicolau. On chip vs off chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3), 2000.

[67] A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded dsps. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 128–138, 1999.

[68] J. Sjodin and C. V. Platen. Storage allocation for embedded processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[69] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design Automation and Test in Europe (DATE)*, 2002.

[70] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmetic capabilities in custom dsp architectures. In *Proceedings Design Automation Conference*, pages 287–292, 1997.

[71] N. Sugino, S. Iimuro, A. Nishihara, and N. Jujii. Dsp code optimization utilizing memory addressing operation. *IEICE Transaction Fundamentals*, pages 1217–1223, 1996.

[72] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architecture. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2006.

[73] F. Sun, N. Jha, S. Ravi, and A. Raghnunathan. Synthesis of appication-specific heterogeneous multiprocessor architectures using extensible processors. In *International Conference on VLSI Design*, 2005.

[74] R. E. Tarjan. Data structures and network algorithms. *SIAM*, 1983.

[75] S. Udayanarayanan and C. Chakrabarti. Address code generation for digital signal processors. In *Proceedings 38th Design Automation Conference (DAC)*, 2001.

[76] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratch pad allocation algorithm. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2004.

[77] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the end IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2004.

[78] B. Wess and M. Gotschlich. Optimal dsp memory layout generation as a quadratic assignment problem. In *Proceedings International Symposium on Circuits and Systems (ISCAS)*, 1997.

[79] B. Wess and T. Zeitlhofer. Optimum address pointer assignment for digital signal processors. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2004.

[80] N. Zervas, K. Masselos, and C. Goutis. Code transformations for embedded multimedia application: Impact on power and performance. In *Proceedings of the ISCA Power-Driven Microarchitecture Workshop*, 1998.

[81] X. Zhuang, C. Lau, and S. Pande. Storage assignment optimizations through variable coalescence for embedded processors. In *Proceedings ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 220–231, 2003.

[82] V. Zivojnovic, J. Velarde, and C. Sclaager. Dspstone, a dsp benchmarking methodology. *Technical report, Aachen University of Technology*, 1994.

Vita

Hassan Salamy is a native of Lebanon. He was born in Beirut in February, 1982. He received his Bachelor of Engineering degree in computer engineering from Lebanese American University, Byblos, Lebanon, in 2003. He took four masters level courses at the Lebanese American University before moving to the USA. In the fall of 2004, he joined the graduate program in the Department of Electrical and Computer Engineering at Louisiana State University (LSU). He received his Master of Science degree in electrical engineering from LSU in 2006. He is expected to receive his doctoral degree in electrical engineering in August 2009.