Abstract

Title of dissertation: COMPILER-DECIDED DYNAMIC
MEMORY ALLOCATION FOR SCRATCH-PAD
BASED EMBEDDED SYSTEMS

Sumesh Udayakumaran, Doctor of Philosophy, 2006

Dissertation directed by: Professor Rajeev Barua
Department of Electrical and Computer Enginnering

In this research we propose a highly predictable, low overhead and yet dynamic, memory allocation strategy for embedded systems with scratch-pad memory. A *scratch-pad* is a fast compiler-managed SRAM memory that replaces the hardware-managed cache. It is motivated by its better real-time guarantees vs cache and by its significantly lower overheads in energy consumption, area and overall runtime, even with a simple allocation scheme.

Scratch-pad allocation methods primarily are of two types. First, software-caching schemes emulate the workings of a hardware cache in software. Instructions are inserted before each load/store to check the software-maintained cache tags. Such methods incur large overheads in runtime, code size, energy consumption and SRAM space for tags and deliver poor real-time guarantees, just like hardware

caches. A second category of algorithms partitions variables at compile-time into the two banks. However, a drawback of such static allocation schemes is that they do not account for dynamic program behavior.

We propose a dynamic allocation methodology for global and stack data and program code that (i) accounts for changing program requirements at runtime (ii) has no software-caching tags (iii) requires no run-time checks (iv) has extremely low overheads, and (v) yields 100% predictable memory access times. In this method data that is about to be accessed frequently is copied into the scratch-pad using compiler-inserted code at fixed and infrequent points in the program. Earlier data is evicted if necessary. When compared to an existing static allocation scheme, results show that our scheme reduces runtime by up to 39.8% and energy by up to 31.3% on average for our benchmarks, depending on the SRAM size used. The actual gain depends on the SRAM size, but our results show that close to the maximum benefit in run-time and energy is achieved for a substantial range of small SRAM sizes commonly found in embedded systems. Our comparison with a direct mapped cache shows that our method performs roughly as well as a cached architecture in runtime and energy while delivering better real-time benefits.

Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems

by

Sumesh Udayakumaran

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:

Professor Rajeev Barua, Chair/Advisor
Professor Donald Yeung
Professor Shuvra Bhattacharrya
Professor Peter Petrov
Professor Chau Wen Tseng

This dissertation is dedicated to my loved ones, without whose support I could not have completed this journey.

ACKNOWLEDGMENTS

At times our own light goes out and is rekindled by a spark from another person. Each of us has cause to think with deep gratitude of those who have lighted the flame within us.

Albert Schweitzer

With these thoughts in mind, I embark on this humble duty to acknowledge different people who have contributed to that light in their own ways.

First and foremost, I acknowledge my parents, my brother and my sister-in-law whose concern for my health and wellbeing, always drives me to work harder towards my goals.

I would also like to acknowledge those who have helped me to complete this dissertation. My advisor, Dr. Rajeev Barua, has been an outstanding motivator. I have learnt several important aspects of research from him, including technical writing and preparing presentations. But most of all, I am extremely indebted to him for all the valuable time he spent discussing my research and his help with various publications and presentations.

I would also like to thank my committee members who gave valuable feedback on my dissertation.

I also wish to thank several of my colleagues. I am most thankful to Angelo Dominguez, whose help with the infrastructure was extremely invaluable. I am grateful to Surupa Biswas, Tom Carley, Steve Haga, Bhuvan Midha, Nghi Nguyen, Mathew Simpson for their help with paper reviews and comments on my presentations. I am also indebted to several of my fellow PhD friends – Brinda Ganesh, Ankush Verma, Wanli liu, Sadagopan Srinivas – for their various inputs to my dissertation.

Doing my PhD has been a tremendous learning experience for me. The 4 years I have spent has given me the opportunity to learn and grow as a human being. The journey would not have been so joyful and enriching if not for my friends in and outside Maryland. I especially recognize the contribution of my friends – friends at Yoga classes, tennis/racquetball sessions and friends sharing thoughts on spirituality, politics and computer architecture – in keeping my spirits up through this journey.

Lastly, I believe the acknowledgement would be incomplete if I do not thank the light within all of us that leads and guides us to live constructively.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

Chapter 1

Introduction

The growing use of computing power in different aspects of daily life has led to embedded systems becoming an important focus of computer architecture research today. In these systems, performance requirements are accompanied by other goals such as low power consumption and suitable form factor. Designers also often face the challenge of ensuring that the task finishes its execution within a specified time.

Performance also being an important requirement in embedded systems, the processor memory gap problem is also an issue in embedded systems. Towards bridging this gap, design of memory systems has received a lot of attention. Memory systems generally are organized using a variety of devices that serve different purposes. Devices like SRAM and flash memory are fast but expensive. On the other hand, devices like DRAM and magnetic disks are slower but being cheaper can be used to provide capacity. Designing a memory system therefore involves using small amounts of faster devices like SRAM along with slower devices like DRAM to obtain satisfactory performance while keeping a check on the overall dollar cost.

In desktops, the usual approach to adding SRAM is to configure it as a hardware cache. The cache dynamically stores a subset of the frequently used data in on-chip memory. Caches have been a big success for desktops, a trend that is likely to continue in the future. Caching is essential in desktop systems for another reason. Using non-cached SRAM is usually not feasible for desktops; one reason is the lack of binary-code portability. Caching, due to its runtime nature, delivers an important benefit for desktop systsms – that of transparency with respect to the cache parameters like the size and associativity. Allocations that are decided when the application is compiled require that the size of the on-chip memory be known; otherwise, they cannot reason about what variables will fit into it. This contrasts with cache allocation which is decided only at run-time; and hence does not require compile-time knowledge of the size of cache. Binary portability is valuable for desktops, where independently distributed binaries must work on any cache size.

Memory organization for embedded systems has other design constraints and issues. Apart from performance, memory organizations now also have to meet other embedded systems goals like real time guarantees, low power consumption and form factor restrictions. However, unlike in desktop systems allocation solutions do not have to provide binary portability. In embedded systems, software is configured along with the hardware in the factory and rarely changes thereafter. This means that embedded system designers can afford to customize the SRAM to a particular size to reap the additional cost savings from customization. This makes alternatives like deciding the allocation strategy at compile time possible. However, the appropriate allocation strategy depends on the environment and requirements of the embedded device. Thus, the choice of allocation strategy is an integral part of memory system design of en embedded system.

Runtime memory management strategies such as hardware caching are often not useful for embedded systems. Typically, embedded systems are used in resource-constrained environments that places constraints like deterministic behavior, low power consumption and small form factor on them. Subsequently, for embedded systems the serious overheads of caches are less defensible. The extra tag hardware that accompanies a cache adds significant power, area and latency overhead; not to mention the non-deterministic behavior of the memory system in the presence of cache. A recent study [13] has shown that non-cached SRAM's use 34% lesser area and consume 40% lower power than a cache of the same capacity. These savings are significant since the on-chip cache typically consumes 25-50% of the processor's area and energy consumption, a fraction that is increasing with time [13]. Moreover, the cacti tool [24] shows that the access latency of caches is significantly lower than that of non-cached SRAM memory of the same capacity. So using a non-cached SRAM is an promising alternative for embedded systems. Such memory is also called *scratch-pad memory.* Given the power, cost, performance and real time advantages of scratch-pad, it is not surprising that scratch-pads are the most common form of on-chip SRAM in embedded CPUs today. Some examples of processors with scratch-pad memory are the Intel IXP network processor, ARMv6 and ARM968E-S, IBM 440 and 405, Motorola's MCORE, 6812 and Dragonball, TI TMS-370, Hitachi's MS32R-32192 and SuperH-SH7050, Infineon XC166, Philips LPC2290, and Atmel AT91-C140; there are many others. Trends in recent embedded designs indicate that the dominance of scratch-pad will likely consolidate further in the future [13] [49].

Although many embedded processors with scratch-pad exist, using the scratch-pad effectively has been a challenge. In contrast, caches have been effectively used in desktop systems for a long time. Central to the effectiveness of caches is their ability to maintain, at each time during program execution, the subset of data that is frequently used *at that time* in the fast memory. The contents of cache constantly change during runtime to reflect the changing working set of data across time. Unfortunately, two of the existing allocation approaches for scratch-pad – program annotations and the recent compiler-driven approaches [11, 12, 73] – are static allocators, *i.e.,* they do not change the contents of scratch-pad at runtime. This is a serious limitation. For example, consider the following thought experiment. Let a program consist of three successive loops, the first of which makes repeated references to array A; the second to B; and the third to C. If only one of the three arrays can fit within the scratch-pad, then any static allocation suffers DRAM accesses for two out of three loops. In contrast, a dynamic strategy can fit all three arrays in the scratch-pad at different times. Although this example is oversimplified, it intuitively illustrates the benefits of dynamic allocation.

In this thesis, we present a new compiler method for allocating three types of program objects – global variables, stack variables and program code – to scratch-pad. The method's feature is that it is able to change the allocation at runtime while avoiding the overheads of runtime methods. The method (i) accounts for changing program requirements at runtime; (ii) has no tags like used by runtime methods; (iii) requires no run-time checks per load/store; (iv) has extremely low overheads; and (v) yields 100% predictable memory access times.

In the rest of the chapter, we first outline our method. We then outline the layout of the thesis. Finally, we summarize the important contributions of the thesis.

1.1 Our dynamic method for scratch-pad management

Our method is outlined as follows. The compiler analyzes the program to identify locations we call *program points* where it may be beneficial to insert code to copy a variable from DRAM into the scratch-pad. It is beneficial to copy a variable into scratch-pad if the latency gain from having it in scratch-pad rather than DRAM is greater than the cost of its transfer. A profile-driven cost model estimates these benefits and costs. The compiler ensures that the program data allocated to scratch-pad fits at all times by occasionally evicting existing variables in scratch-pad to make space for incoming variables. In other words, just like in a cache, data is moved back and forth between DRAM and scratch-pad, but under compiler control, and with no additional overhead.

Key components of our method are as follows. (i) To reason about the contents of scratch-pad across time, it helps to attach a concept of time to the above-defined program points. Towards this end, we introduce a new data structure called the *Data-Program Relationship Graph (DPRG)* which associates a *timestamp* with each program point. (ii) A detailed cost model is presented to estimate the runtime cost of any proposed data transfer at a program point. (iii) A compile-time heuristic is presented that uses the cost model to decide which transfers minimize the runtime.

Program Code Our base method can allocate global and stack objects. We also separately show how our method can also easily be extended to allocate program code objects. Although, code objects are accessed more heavily than data objects (one fetch per instruction), dynamic schemes like ours are not likely to be applicable in all cases for two reasons. First, compared to data caches the use of instruction caches is more feasible due to their effectiveness at much smaller sizes. So it is not uncommon to find instruction caches (but not data caches) especially in high end embedded systems like Motorola's STARCORE, MFC5xx and 68HC. Second, for low and medium end embedded systems code is typically stored in ROM/flash. An example of such a system is Motorola's MPC500. Unlike DRAM devices, ROM/flash devices have lower seek times (in the order of 75ns-120ns, 20 ns in burst/page mode) and power consumption. For low end embedded systems, this would mean an access latency of a cycle or two. For such low end embedded systems using ROM/Flash where cost is also a lot more important factor, speeding up accesses to code objects as compared to accesses to data objects in DRAM is not very attractive. Nevertheless, for high end systems, which store code in ROM/flash such as the Motorola MCORE and Motorola 6812, methods to speed up accesses to code can improve performance immensely. Our proposed extension for handling code would thus enable our dynamic method to be used for speeding up code accesses in such systems.

1.1.1 Scope and restrictions

Profile dependence Our method is profile-dependent; that is, its improvements are dependent upon how representative the profile data set really is. *Indeed, all existing scratch-pad allocation methods, whether compiler-derived or programmer-specified, are inherently profile-dependent.* This cannot be avoided since they all need to predict which data will be frequently used. Further, our method does not require the profile data to be like the actual data in all respects; so long as the relative re-use trends between variables are similar in the profile and actual data, good allocation decisions will be made, even if the re-use factors are not identical. A regions gain may even be higher with non-profile data if its data re-use is more than in the profile data.

Heap data Our method does not allocate heap data in the program to the scratch-pad. Programs with heap data still work; however, all heap data is allocated to DRAM and the global stack and program code can still use the scratch-pad using our method, but no SRAM acceleration is obtained for heaps. Heap data is difficult to allocate to the scratch-pad at compile-time because the total amount and lifetime of heap data is often data-dependent and therefore unknowable at compile-time. Software caching strategies [36, 59] can be used for heap, but they have significant overheads. Method for allocating heap data to scratch pad [32] can be easily integrated with our method; indeed that paper shows how.

1.1.2 Impact

Here we briefly preview the benefits of adopting our method over both existing compile-time technologies like static allocation and hardware mechanisms like caching.

Quantitative benefits If adopted, the impact of this work will be a significant improvement in the cost, energy consumption, and runtime of embedded systems. Our results show up to 39.8% reduction in run-time for our method for global and stack data and code vs. an existing static allocation scheme. With hardware support for DMA, present in some commercial systems, the runtime gain increases to up to 42.3% respectively. *The actual gain depends on the SRAM size, but our results show that close to the maximum benefit in run-time and energy is achieved for a substantial range of small SRAM sizes commonly found in embedded systems.* Using an accurate power simulator, our method also shows up to 31.3% reduction in energy consumption vs. an existing static allocation scheme. Our method also does marginally better than a cached architecture both in runtime and energy consumption. Finally, integrating our method with optimizations for partial variable handling gives it ability to provide gains at much smaller SRAM sizes. At the same time, the method still remains generally applicable for a large variety of programs. The details of our results are provided in chapter 10.

Real-time guarantees Not only does our method improve run-time and energy consumption, it also improves the real-time guarantees of embedded code. Our method like all compiler-decided allocation methods, guarantees that the latency of each memory instruction is known for sure. This translates into the behavior of the memory system becoming totally predictable, thus immensely aiding in obtaining a better WCET. Such real time benefits of scratch-pad have been observed before too [89]; by improving runtime our method aims to improve the WCET's more than the existing static methods.

1.2 Overview of thesis

This section gives an overview of the thesis. First, it summarizes the significant contributions of the thesis. Second, it gives an overview of the rest of the chapters.

Contributions Our thesis makes the following contributions.

- This research – the first version of which was published in [83] – represents the first solution to the problem of compile-time dynamic allocation of the scratch-pad memory using whole program analysis.

- To aid in a whole program analysis, the thesis introduces the notion of a timestamped representation of the program. Such a representation, we believe, can be useful in other region based optimizations.

- Towards making the solution comprehensive, the method can handle various program structures and data types like pointer variables.

- The thesis provides an extension of the base solution to also allocate code variables. This makes it a general solution capable of handling all non-heap variables.

- Further, the thesis extends the method to a framework that can incorporate a variety of optimizations that can create partial variables.

- To prove the effectiveness of the framework, the thesis presents an affine analysis based optimization that can exploit the presence of affine accesses to arrays in the code and create smaller partial variables.

- Finally, the method is implemented inside the gcc compiler as a proof of its implementation viability in a commercial compiler.

Organization

The rest of the thesis is organized as follows. Chapter 2 introduces scratch-pad memory and some aspects concerning its management. Chapter 3 overviews some basics of our method. Chapters 4 through chapter 6 describe the details of the method are described. Chapter 4 describes the precise method used to determine the memory transfers of global and stack variables at each program point. Chapter 5 describes how the algorithm is modified to correctly handle certain language features. Chapter 6 describes the layout of variables in scratch-pad and the process of code generation. Chapter 7 addresses how to handle programs with pointers. Chapter 8 extends the method to a framework that can use various partial-variable optimizations. It presents an example of how this can be done with the help of affine-analysis-based optimization. Chapter 9 overviews related work. Chapter 10 presents an evaluation of our methodology. Chapter 11 concludes and discusses some future direction.

Chapter 2

Scratch-Pad Memory

In this chapter, we discuss the concept of a scratch-pad memory. We further look at its benefits for embedded systems and importance of its management.

Scratch-pad memory refers to on-chip memory that has a separate address space as seen by the CPU. Such memory can be on chip SRAM or DRAM. The terminology, *perhaps*, owes its origin to the use of such memory by CPU in some superscalar processors for storing intermediate values or instructions, in other words used as a scratch-pad memory. They are also sometimes referred as on-chip memory or tightly coupled memory. The main characteristic of such memory is that it is an explicit part of the complete address space. In other words, while a cache memories address space is part or whole of the off-chip address space, a scratch-pad memory has an address space outside the off-chip memory. Figure 2.1 and Figure 2.2 illustrates this difference.

The rest of the chapter is organized as follows. Section 2.1 describes the benefits of using scratch-pad memory in an embedded device compared to using a hardware cache. Section 2.2 overviews the importance of appropriate scratch-pad

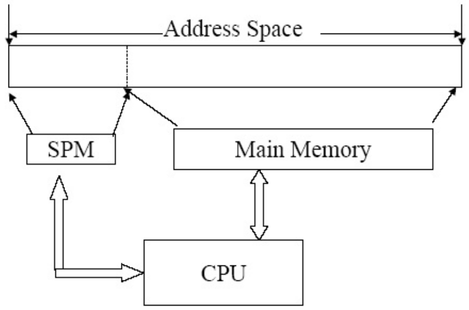


Figure 2.1: Architecture with scratch-pad.

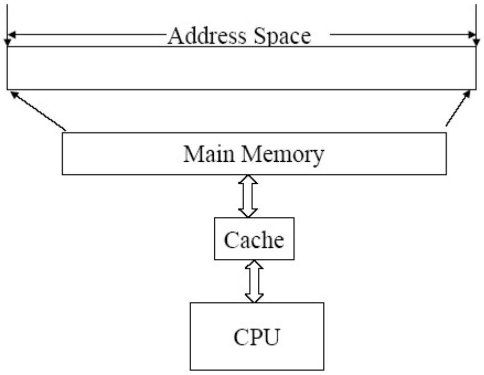


Figure 2.2: Architecture with cache.

management technology with the help of two existing management technologies. Finally, section 2.3 illustrates a few architectures that use the scratch-pad.

2.1 Relevance of scratch-pad memory in embedded system design

Scratch-pad memories offer some inherent advantages that work towards making them a more favorable alternative in embedded system design. Banakar et al in [13] argued the case for use of scratch-pad in embedded systems. They reported significant area, energy and latency gains for scratch-pad based embedded systems. With the help of experiments similar to ones described by them, we show comparison between different cache configurations and scratch-pad memory in figures 2.3, 2.4 and 2.5. These figures respectively show how area, per-access energy cost and per-access latency vary for different sizes and different associativities. The sizes vary from 64K to 512 K. The associativity is varied from direct mapped to associativity of 4. The cache line size is chosen as 8 words and the technology fixed at 0.5 micron.

The numbers for the scratch-pad memory for a particular size is obtained from the numbers of a direct mapped cache for that size by subtracting the contribution of the tags.

From figure 2.3, we see that across all configurations, scratch-pad memory consumes the least energy per-access. Energy is an important criterion in embedded system design. Embedded systems are often battery powered. Due to weight, size and cost constraints, these systems have to carry batteries limited in their power availability. Power consumption also impacts the heat dissipation of the device, which is factor in the usability of the device. With a lower per-access energy cost, scratch-pad memory devices promise lower power consumption. The reduction in energy is further helped by the reduction in area. In figure 2.4, we see that scratch-pad devices also use lesser area for the same SRAM size. Area reduction in turn leads to reduction in leakage current. Area reduction also means an improvement in the production efficiency of the chips.

Scratch-pad devices also have lower per-access latency. This is seen in figure 2.5. Although this by itself does not translate into overall performance benefit over caching, it offers the promise that good overall performance can be achieved.

These gains can be attributed to the absence of tags and other overhead bits in a scratch-pad memory. The lack of tags also means that the designer is free to choose his own memory management. Such a choice can be based on the specific needs of his system. This, as we will see, is very useful as allocation strategies can have significant impact on the constraints of an embedded system, particularly the predictability of the memory system. The predictability of the memory system is an important

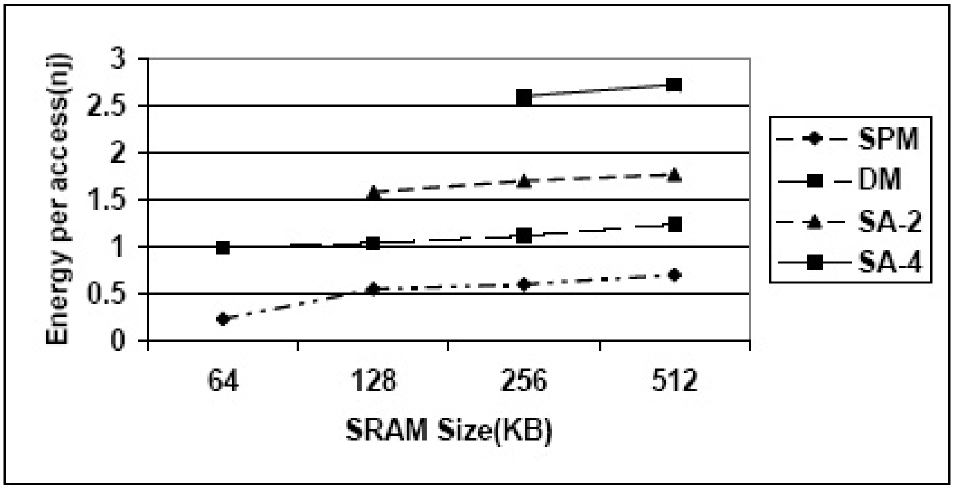


Figure 2.3: Per-access energy comparison between scratch-pad memory and various cache configurations.

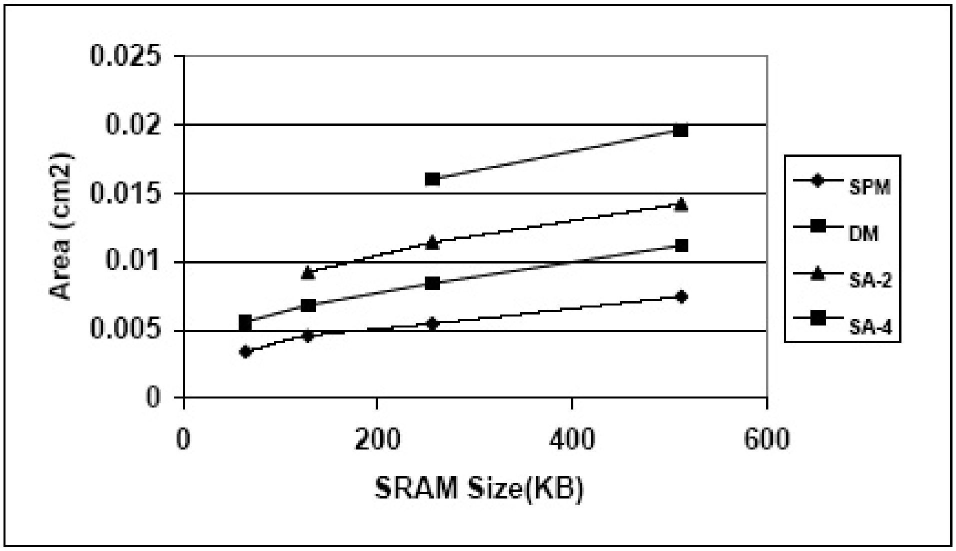


Figure 2.4: Area comparison between scratch-pad memory and various cache configurations.

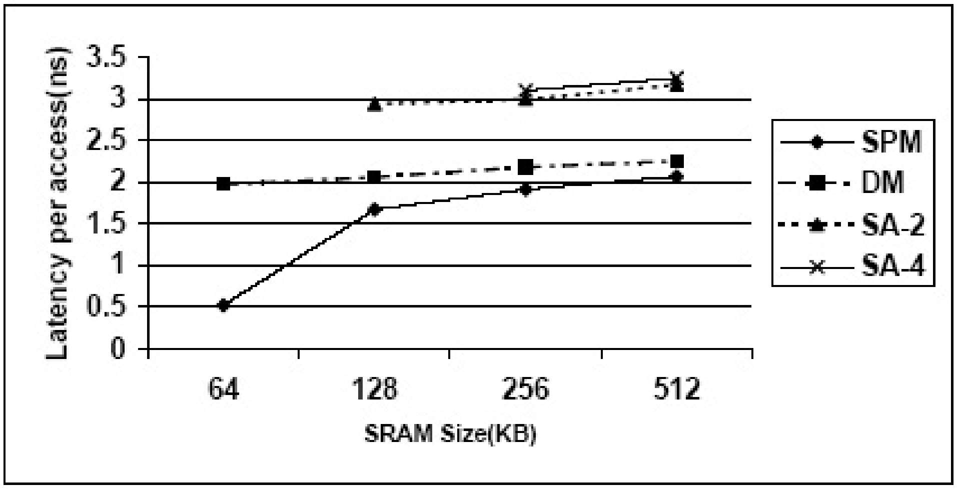


Figure 2.5: Per-access latency comparison between scratch-pad memory and various cache configurations.

component in providing real time guarantees. A proper allocation strategy is also important in translating the inherent advantages of a scratch-pad memory to benefits for the system. The absence of tags positions scratch-pad based systems to better meet the goals of embedded systems compared to cached architectures.

2.2 Scratch-pad management

In order that the above said advantages of scratch-pad over cache translate into benefits for the system, the choice of scratch-pad allocation strategy is very important. Different strategies favor different criterion. We will now using two existing strategies, software caching and static allocation, see how a particular criterion is affected by allocation strategies.

2.2.1 Software caching

Software caching [36, 59] represents a dynamic allocation strategy. This class of methods emulate the behavior of a hardware cache in software. In particular, a tag consisting of the high-order bits of the address is stored for each cache line in software. Before each load or a store, additional instructions are compiler-inserted to mask out the high-order bits of the address, access the tag, compare the tag with the high-order bits and then branch conditionally to hit or miss code. On a hit, in the hit code, using a mapping table the address is mapped to a new address in the scratch-pad where the data resides. Otherwise it is a cache miss and new data is brought into the scratch-pad. At that point, the index tables are updated to reflect the presence of this new data in the scratch-pad. Being a runtime strategy like caching, the strategy can adapt to runtime conditions. This makes it particularly suited for workloads whose runtime conditions continuously vary.

However, the use of the extra inserted instructions also leads to some drawbacks. Some methods are able to reduce the number of such inserted overhead instructions [59], but much of it remains, especially for non-scientific programs. Needless to say, the inserted code adds significant overhead, including (i) additional runtime; (ii) higher code size and dollar cost; (iii) higher data size and cost from tags; (iv) higher energy consumption; and (v) memory latency that is as unpredictable as hardware caches. Similar to a cache, it is hard to predict if the latency of a memory access would be that of a cache hit or a cache miss. Unpredictable memory latency makes such a scheme unsuitable for applications with hard real time deadlines.

2.2.2 Static allocation

In static management of scratch-pad memory, the contents of the scratch-pad are decided at compile time itself. A simple way of doing this is based on programmer hints. Thus, the programmer provides annotations that tell if a variable is to be put into the scratch-pad address space or not. In such a scheme, it is the programmer responsibility to ensure that a particular variable fits completely inside the scratch-pad address space. To make this process efficient, the selection strategy can be based on compiler analysis of the application. Examples of such static allocation strategies include [12, 13, 73, 77]. The advantage of a static scheme is that it does not involve any additional per-access overhead like that of software caching as no translation is required. So the inherent gains of a scratch-pad memory are still preserved. No translation gives scratch-pad memory yet another advantage that makes it suited for real time applications. For most memory memory access, its exact location is known and hence also known is its latency. This enables predicting the memory system behavior accurately.

A drawback of the static scheme is that being a static scheme, the allocation cannot adapt to changing runtime conditions. This makes it unsuitable for memory bound embedded applications where performance heavily depends on the program locality. Examples of such applications are high performance scientific and DSP applications. In spite of the drawback, researchers have shown for a limited set of benchmarks that the inherent advantages of a a scratch-pad over the cache can be translated to overall runtime, energy gains when compared to a cache [13].

2.3 Example architectures

Due to the numerous advantages offered by scratch-pad memory, it is dominant in embedded systems. The prevalence of scratch-pad memory in embedded systems is evident from the large variety of chips with scratch-pad memory available today in the market (*e.g.,* [1, 20, 41, 42, 60, 61, 81]).

A large number of systems exist that only use scratch-pad memory. Examples of such architectures include low-end chips such as the Motorola MPC500, Analog Devices ADSP-21XX, Motorola Coldfire 5206E; mid-grade chips such as the Analog Devices ADSP-21160m, Atmel AT91-C140, ARM 968E-S, Hitachi M32R-32192, Infineon XC166 and high-end chips such as Analog Devices ADSP-TS201S, Hitachi SuperH-SH7050, and Motorola Dragonball. However, sometimes in some high-end systems, scratch-pad memory is used in conjunction with either only Icache or Dcache or sometimes both. This is done with the objective of combining the benefits of both cache and scratch-pad memory. A majority of these systems also provide DMA hardware for fast data transfers. We now illustrate some representative processors with varying scratch-pad memory based memory systems. Apart from showing how scratch-pad memory is useful in these processors, these examples also point to how the scratch-pad memory fits into the overall architecture of the processor.

ARM968E-S The ARM968E-S [9], which belongs to ARM9E family, is targeted for embedded real-time applications. Its key characteristic is that it is small and low power. Its size and power advantages are partly contributed by separate scratch-pad memories for instruction and data. The scratch-pad memory is also termed tightly coupled memory and has customizable sizes. The ARM968E-S has a dual-banked scratch-pad to store data and a DMA controller to share access to the scratch-pad memory. The DMA interface accesses the data scratch-pad (also called data tightly coupled memory or DTCM) through two separate ports, D0TCM and D1TCM. The processor and the DMA controller alternately access the D0TCM and D1TCM ports. This unique feature enables the DMA port to move external data blocks into the Data scratch-pad without stalling the processor access during the DMA block move. Some of the applications its particularly found use in are: networking systems, wireless devices, storage devices consumer devices like audio players.

MPC565 The MPC565 [62] from Motorola belongs to the MPC500 family. The MPC500 family is targeted at a variety of Global Positioning Systems (GPS), ranging from high-velocity, fast acceleration aircraft applications to low-speed, high precision agriculture applications. Its memory sub-system consists of 1Mbyte of embedded Flash memory and 36 KB scratch-pad memory. The scratch-pad is useful in storing the data read by the sensors, while the embedded flash stores the code. The 1Mbyte of embedded Flash is divided into 2 blocks of 512 Kbytes. Because the memory is configured into two separate blocks, program code can be executed from one block of Flash while programming into the other.

ADSP-21262 The ADSP-21262 [4] from Analog Devices belongs to the SHARC programmable DSPs. The ADSP-21261 chip is used in a range of processors such as high-quality audio and automotive entertainment systems, voice recognition, medical appliances and measurement. The memory system of ADSP-21262 represents how scratch-pad memory can be combined with different devices to provide a sophisticated memory system. The memory system includes 256 KB of on-chip dual-ported scratch-pad memory, 512 KB of mask programmable ROM memory and an on-chip instruction cache of size 192 KB. The scratch-pad is dual-ported and enables sustained processor and I/O performance without the need for external memory. For fast delivery between the scratch-pad and the main memory, it has 22 zero-overhead Direct Memory Access (DMA) channels, thus avoiding processor intervention. The cache is used only when there is a conflict for the program memory bus between instruction fetches and data fetches. This cache allows full-speed execution of core and looped operations such as digital filter multiply-accumulates.

2.4 Summary

Scratch-pad memory offers several inherent advantages like lesser area, faster per-access time and lower per-access energy cost. The choice of the allocation strategy is important in not only translating these advantages to benefits for the system but also ensuring other requirements like real time guarantees. Due to these advantages, scratch-pad memory systems find use in many embedded systems in variety of ways.

Chapter 3

Dynamic Allocation Overview

In the absence of hardware that helps un runtime management of memory, ensuring good performance in the case of scratch-pad memory becomes the task of the allocation algorithm. The role of the allocation algorithm is crucial in preserving the advantages of a scratch-pad memory over a cache like lower per access energy consumption, lesser area. Allocation algorithms have their most influence on the behavior of the application – whether its memory system behaves deterministically or not. Totally dynamic allocation algorithms that modify the memory contents based on run-time conditions cannot provide good real time guarantees. Static allocation schemes, although they provide for deterministic memory behavior, do not adapt to runtime conditions; thus limiting their performance. So while the allocation being dynamic is good for performance, it conflicts with providing real time guarantees.

The motivation of this thesis is to explore a dynamic strategy that also offers good real time guarantees. With such a strategy we target embedded real time application that also require performance and low energy consumption. In this chapter we discuss some of the basic aspects of such a strategy. Section 3.1 defines some of the parameters of our solution strategy. Section 3.2 looks at some of the characteristics of our method, some of which manifest as challenges in the problem.

3.1 Designing a dynamic strategy

Our choice of these features is guided by the applications we have targeted – applications that have severe real time requirements. Additionally, we assume that these applications are run in resource constrained environment and hence are required to consume less power, have small memory footprint while also having a performance requirement.

Greedy versus non-greedy solutions

We first consider if the dynamic algorithm should be heuristic driven or otherwise. Towards that we prove that the problem is NP-hard.

Theorem 1. *The dynamic memory allocation problem for Scratch-pad memory systems is NP-hard.*

Proof. The dynamic allocation problem can be shown to NP-hard by reducing the problem from register allocation; register allocation has been proven NP-Complete [69].

Consider the problem of global register allocation with k registers; each register of size v words and an input variable set A. An equivalent dynamic memory allocation problem can be constructed by the following two steps. First, determine the scratch-pad size as k*v. Second, enforce a alignment restriction of v bytes. Each of these steps is trivially done. Thus, from the NP-Completeness of register allocation [69], it follows that the dynamic allocation problem is NP-hard. □

The consequence of theorem 1 is that polynomial-time optimal solutions are highly unlikely. With that in mind, we develop a heuristic solution. Our heuristic-driven algorithm exploits several well known algorithm and program properties. First, we restrict our program points to only points those that are likely to be useful. Second, we use a divide-and-conquer approach to divide our problem into several subproblems. Each of these problems falls into variants of well-known classical problems namely the 0/1 knapsack problem and the bin packing problem that have good greedy solutions.

Pseudo dynamic versus dynamic Next we consider if the algorithm should be purely dynamic or not. The drawback of truly dynamic allocations is that the memory behavior is then non-deterministic, which then impacts the real time guarantees. On the other end, static allocations do not exploit the locality of the application and are hence likely to be inferior in runtime.

We choose a profile-guided pseudo-dynamic allocation strategy and hence avoid a purely dynamic strategy. Using the profile information we find a dynamic allocation at compile time. This involves using the compiler to insert code at different program points to change the contents of the scratch-pad memory at different points. While this can exploit the locality to some extent, the contents of the scratch-pad are known exactly at different program points. This leads to total predictability of latency for each memory access.

Number of allocations at a program point Due to the presence various control structures, program execution can pass through a particular program point multiple times; some of the paths through the point may be different from the others. This leads to the following design question: how many different allocation should be associated with a particular program point. A purely dynamic solution can be considered one where allocation at any point is totally unconstrained. An example of such a strategy is software caching discussed earlier. However, software caching has drawbacks of non-determinism and per-access translation cost. A slightly less dynamic allocation could be one where at any point a set of allocations is determined for different execution paths through the point. We here consider if such alternative methods can be designed that avoid the per-access translation overhead and non determinism in truly dynamic allocations. We will see why two such approaches we speculate on are not likely to be successful[1].

First, *cloning* makes a copy of each code region for each of its possible dynamic allocations. The program then checks runtime conditions and jumps to the most suitable of the cloned alternatives. In this way, the overhead is reduced from being per-access to per-region, making it manageable. Unfortunately such a solution has a serious drawback: the code growth can be exponential in the number of program variables in a region – for n variables, each in SRAM or DRAM, the number of possible allocations is 2^n. In addition, selective cloning of a few variables is infeasible since a factor of 2^n increase in code size is un-acceptable even for small n. Further,

[1] These two hypothetical approaches are not used in any existing scratch-pad allocation method. Further, these are the only approaches *we* could imagine for truly dynamic allocations – that is not to say that others are not possible.

selectively cloning only a few time-critical regions is difficult. This is because allocation uncertainty in one region causes allocation uncertainty in subsequent regions, requiring more cloning.

A second method to reduce per-access overhead is *conditional allocation.* Here variables are accessed indirectly through conditionally assigned temporaries, eliminating cloning, but an exponential number of check cases is still needed per region to assign its temporaries. This is because the number of allocation possibilities is still exponential, and the allocation decisions for different variables are not independent since whether a variable fits in scratch-pad depends on the allocation of other variables. Further, the run-time overhead increases because of the access overhead from indirection through temporaries. Because of the large overheads and complexity of these alternatives, we do not believe these alternatives to be promising, and do not study them further. This leads us to the next design criterion for our algorithm– *we restrict the number of allocations at any program point to one.*

Need for Interprocedural Analysis Our next criterion is whether the algorithm should be interprocedural or not. The need for interprocedural analysis arises due to the large amount of data that a dynamic allocation strategy is likely to handle. Without a interprocedural mechanism, it would mean the scratch-pad is emptied at the boundaries of a procedure. In particular, if the function calls are present inside a loop, then transfers accumulated over the whole loop would be a significant fraction of the total runtime. Not only is this expensive, it is also unnecessary. Aggregate data types like arrays and structures often exhibit cross function reuse. Unnecessary eviction of such large data at the boundaries of function can severely degrade the overall runtime. In general, as will be shown by statistics in the results chapter 10, interprocedural analysis enables the same allocation to be retained with minimal transfers.

To summarize, our method is motivated by the following design goals.

- The strategy should be pseudo-dynamic and compiler-decided strategy so that it would provide real time guarantees while adapting to runtime conditions.
- The strategy should be heuristic-driven to efficiently explore the large design space.
- The strategy should have one allocation at every point to avoid unnecessary code growth.
- The allocation strategy should be interprocedural to avoid unnecessary transfers.

3.2 Characteristics of our allocation strategy

The choice of being a compiler-decided and dynamic scheme leads to several characteristics that are inherent to both compiler-decided and dynamic methods. Some of them are advantageous while there are some which a designer has to be careful about. We now look at some of these characteristics.

Compiler schemes have a unique set of advantages and disadvantages due to their view of memory that is different from how hardware schemes view memory.

While compiler schemes view memory as made up of various data types, hardware schemes have a uniform view of memory made of bytes or words. Consequently, compiler schemes can utilize a lot of program information; this is not easy for hardware schemes. This difference can also work in favor of hardware schemes. Some of the differences and their impact is discussed below.

Data size While a compiler scheme can handle precisely only those addresses that belong to a particular variable, a hardware scheme handles all the addresses that belong to the cache lines it is fetching. This characteristic can work in the favor of hardware schemes as well. Hardware schemes can be totally oblivious of the size of the variable. In the case of compiler schemes, size of a variable is an important parameter that influences lot of decisions like if the variable would fit into the memory, what addresses to assign the incoming variable.

Dead data Similarly, while a compiler scheme can use program information to identify stale data in memory, hardware schemes cannot do so.

Code generation In the case of a hardware scheme, code can be generated independent of the underlying memory. Such binary portability is not possible in the case of compiler schemes. Information about the scratch-pad has to be taken into account while generating the code.

Correctness issues Issues pertaining to correctness of the program also arise because of the program view of memory. One such issue is due to the presence of pointers. Hardware schemes can be totally oblivious of the kind of programs run. In contrast, presence of pointers causes issues for compiler schemes. We elaborate on the issue of pointers in Chapter 7.

Our method also has some characteristics due to being a dynamic scheme. Some of these are discussed below.

Coherence issue Due to dynamic nature of the algorithm, several versions of the same variable can exist in memory. For example, when a variable is copied from the DRAM to the SRAM, two versions of the variable exist. If then a new copy of the variable is created either from its copy in SRAM or from DRAM a third new version results. Hence, the allocation algorithm has to keep track of the latest copy of the variable to avoid coherence issues.

Runtime adaptability Being a dynamic scheme, our method has the ability to adapt changing runtime conditions.

3.3 Dynamic solution overview

We next overview our method. The process flow of our method is illustrated in figure 3.1. A brief description of the entire flow follows. The figure shows two parallel paths of flow. The data inputs are shown in rounded rectangles. In the first path, profile information is collected and stored in the stucture called the DPRG. Depending on the application, the code can be optionally optimized and analyzed for partial variable references. The results of the partial variable analysis is also stored in the DPRG. The second path is the allocation strategy. The strategy is centered around three steps that form the core of our method. After pointer analysis is done, the main allocator takes in profile input and the memory model and outputs an allocation. This is followed by address assignment and code generation. The memory model provides memory device information like the latencies, energy model and sizes of the different memory devices.

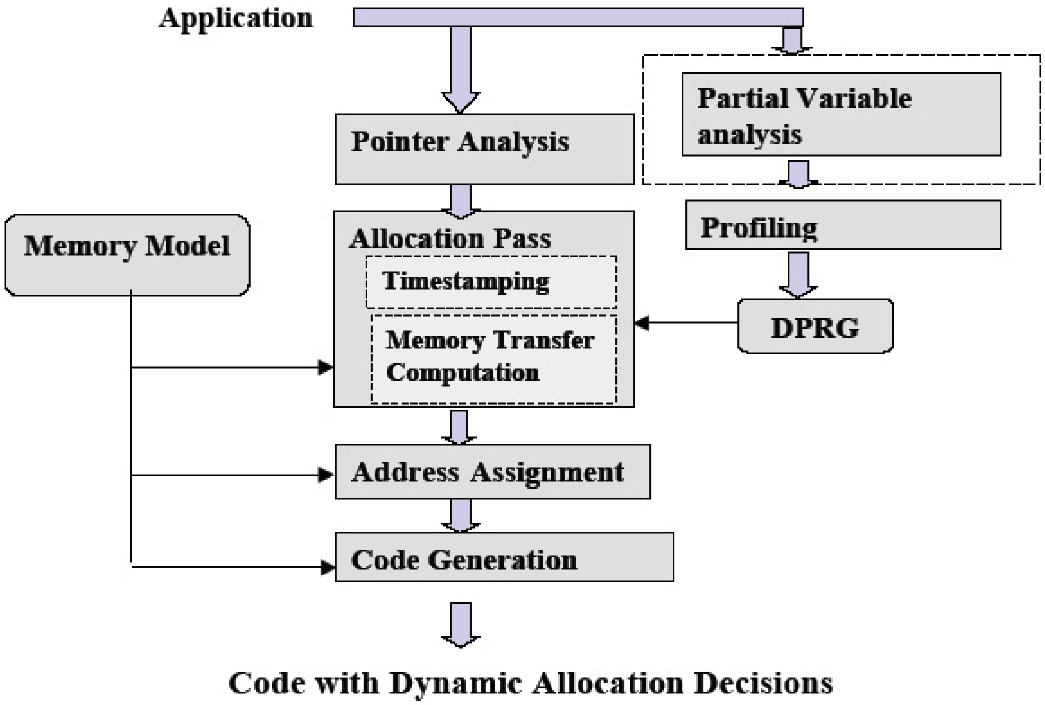


Figure 3.1: Dynamic memory allocation methodology

3.4 Dynamic memory allocation parts

Profiling The objective of profiling is to find the relative use of different variables in different portions of the program. To achieve this, it proceeds in three steps. First, it partitions the program into regions where the start of each region is a *program point*. Then, instrumentation code is inserted at these program points. The instrumentation involves code to capture information like how many times variable around that point are accessed. Then when the instrumented code is run, profile information is collected which is stored in the data structure called the DPRG. We discuss this structure elaborately in section 4.1.

Allocator

The allocator forms the central part of our method. The allocator can be further divided into two sub-parts. The first part takes the DPRG as input from the profiler and associates a timestamp with every program point such that (i) the timestamps form a partial order; and (ii) the program points are reached during run-time roughly in timestamp order. Timestamping is discussed in detail in section 4.1 The second sub-part determines the contents of the scratch-pad memory at different points. Changes in allocation are made only at program points by compiler-inserted code that copies data between the scratch-pad and the DRAM. The method visits these different points in a timestamped order and determines the variables to be transferred between memories at each program point by using the cost-model. The cost-model takes into account different factors like the current contents, possible transfer points (eg. just outside the different loops of a loop nest), transfer costs and parameters of the memory model. We discuss the algorithm and various aspects of it in detail in chapters 4 and chapter 5.

Address assignment In the next step, the allocator decides on the addresses of the scratch-pad memory variables in different regions. This involves deciding which available address range in the scratch-pad memory (free hole) to use to accommodate a variable allocated to the scratch-pad memory. To achieve this, the methods again visits the regions in timestamp order and attempts to fit the variables newly allocated to the scratch-pad memory into the available free holes in the scratch-pad memory in a best-fit manner. Since over time memory might get fragmented and consequently free holes of desired size might not be available, the method invokes a limited compaction mechanism when it runs out of adequate-sized holes. The details of this step are discussed in section 6.1

Code generation Code generation involves using new temporaries to access a variable allocated to the scratch-pad memory. A temporary represents a variable allocated to the scratch-pad memory at a specific address. Use of temporaries also implies that at any point multiple versions of a variable may exist, but with one latest version known to the compiler. Code generation also includes memory transfers inserted before and after regions to copy between the temporary and the original variable. The details of this step are discussed in section 6.2

Optimizations and partial variable handling Finally, optimizations can be integrated into the process flow to improve the allocation solutions. In chapter 8, we discuss how our method can be extended to incorporate partial-variable handling.

3.5 Summary

In this chapter, we overviewed some fundamental aspects of our algorithm. First, based on the requirements we wanted to meet, we fixed the different design choices for our method. The design choices decided several characteristics and challenges that our method would need to address. Finally, we briefly described the process flow of our solution.

Chapter 4

Dynamic Allocation Algorithm

This chapter describes the proposed algorithm for determining the memory transfers of global and stack variables at each program point.

An outline of the chapter follows. We first in section 4.1 discuss the data structure that is a key input to our algorithm. Then, we discuss our method that allocates global and stack variables in 4.2. In section 4.3, we show how this method can be extended to also allocate program code.

4.1 Deriving regions and timestamps

Our algorithm relies on identifying promising parts of the programs where certain variables can be allocated to the scratch-pad memory. So an essential task is identifying these parts. As the first task, the algorithm partitions the program into *regions* where the start of each region is a *program point*. Changes in allocation are made only at program points by compiler-inserted code that copies data between SRAM and DRAM. The allocation is fixed within a region. The choice of regions is discussed in the next paragraph. Then it associates a *timestamp* with every program point such that (i) the timestamps form a partial order; and (ii) the program points are reached during runtime in timestamp order. In general, it is not possible to assign timestamps with this property for all programs. Later in this section, however, we show a method that by restricting the set of program points and allowing multiple timestamps per program point, is able to define timestamps for all programs.

The choice of program points and therefore regions, is critical to our algorithm's success. Regions are the code between successive program points. Promising program points are (i) those after which the program has a significant change in locality behavior, and (ii) those whose dynamic frequency is less than the frequency of its following region, so that the cost of copying into SRAM can be recouped by data re-use from SRAM in the region. For example, sites just before the start of loops are promising program points since they are infrequently executed compared to the insides of loops. Moreover, the loop often re-uses data, justifying the cost of copying into SRAM. With the above two criteria in mind, we define program points as (i) the start and end of each procedure; (ii) just before and just after each loop (even inner loops of nested loops); (iii) the start and end of each **if** statement's **then** part and **else** part as well as the start and end of the entire **if** statement; and (iv) the start and end of each case in all **switch** statements in the program as well as the start and end of the entire **switch** statement. In this way, program points track most major control-flow constructs in the program. Program points are merely candidate sites for copying to and from SRAM – whether any copying code

```
main () {
  if (...)   { proc-D()}   else   {...}
  proc-A()
  proc-B()
}

proc-A () {
  proc-C()
}

proc-B () {
  proc-C()
  while (...){Y =...}
}

proc-C () { X =...}

proc-D () { ...}
```

Figure 4.1: Example showing a program outline.

is actually inserted at those points is determined by a cost-model driven approach, described later in section 4.2.

Figures 4.1 and 4.2 shows an example illustrating how a program is marked with timestamps at each program point. Figure 4.2 shows the program outline. It consists of five procedures, namely *main()*, *proc-A()*,*proc-B()*, *proc-C()* and *proc-D()*, one loop and one **if-then-else** construct. The only program constructs shown are loops, procedure declarations and calls, and **if** statements – other instructions are not. Accesses to two selected variables X and Y are also shown.

Figure 4.2 shows the *Data-Program Relationship Graph* (DPRG) for the pro-

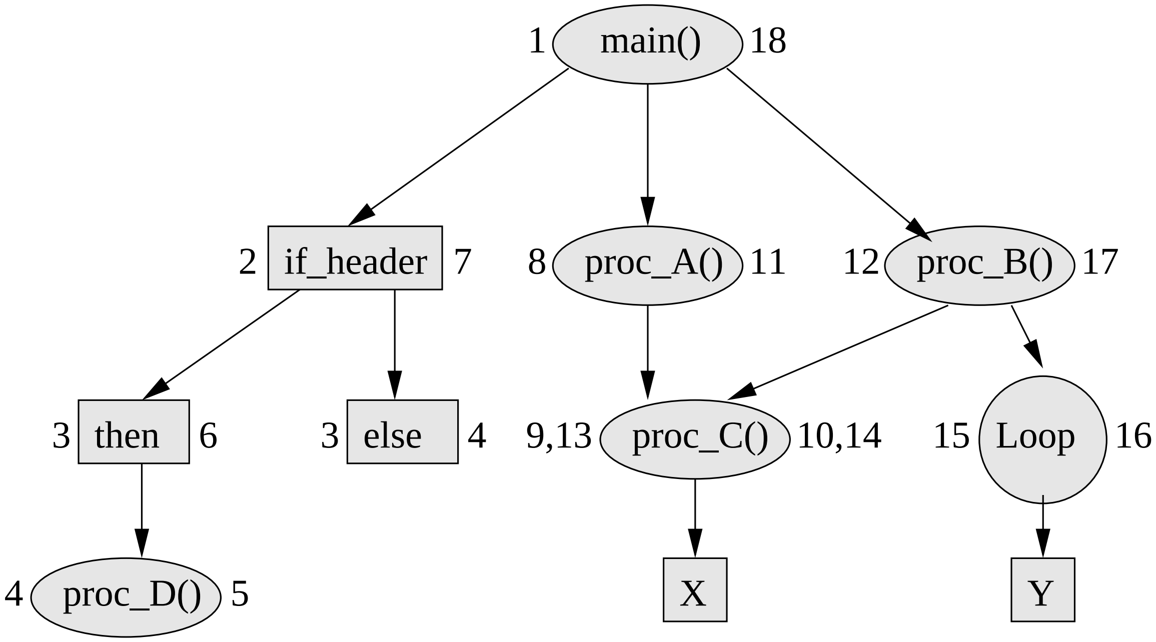


Figure 4.2: Example showing the DPRG showing nodes, edges & timestamps.

gram in figure 4.1. The DPRG is a new data structure we introduce that helps in representing regions and reasoning about their time order. The DPRG is the program's call graph appended with new nodes for loops, if-then's and variables. In the DPRG shown in figure 4.2, there are five procedures, one loop, one if statement, and two variables represented by nodes. Separate nodes are shown for the entire **if** statement (called if-header) and for its **then** and **else** parts. On the figure, oval nodes represent procedures, circular nodes represent loops, rectangular nodes represent **if** statement nodes, and square nodes represent variables. Edges to procedure nodes represent calls; edges to loop and if nodes shows that the child is in its parent; and edges to variable nodes represent memory accesses to that variable from its parent. Continue and break statements are not modeled separately. The DPRG is usually a directed acyclic graph (DAG), except for recursive programs, where cycles occur.

Figure 4.2 also shows the timestamps (1-18) for all program points, namely the beginnings (shown on left of nodes) and ends (shown on right) of every *procedure*, *loop*, *if-header*, *then* and *else* node. The goal is to number timestamps in the order they are encountered during the execution. This numbering is computed at compile-time by the well-known depth-first-search (DFS) graph traversal algorithm. Our DFS marks program points in the order seen with successive timestamps. Our DFS is modified, however, in two ways. First, our DFS is modified to number *then* and *else* nodes of **if** statements starting with the same number since only one part is executed per invocation. For example, the start of the *then* and *else* nodes shown in the figure both are marked with timestamp 3. The numbering of the end of *if-header* node (marked 7 in the figure) follows the numbering of either the *then* and *else* parts, whichever consumes more timestamps. Second, it traverses and timestamps nodes *every time they are seen*, rather than only the first time. This still terminates since the DPRG is a DAG for non-recursive functions. Such repeated traversal results in nodes that have multiple paths to them from *main()* getting multiple timestamps. For example, node *proc-c()* gets timestamps 9 & 13 at its beginning, and 10 & 14 at its end.

Now we can see that the timestamps are a partial order rather than a total order. This is because timestamps should not be used to derive an order between two nodes such that one is a child of the *then* part of some *if-header* node, and the other is a child of the *else* part of the same *if-header*. Such nodes have no relative order.

Timestamps are useful since they reveal dynamic execution order: the runtime order in which the program points are visited is roughly the order of their timestamps. The only exception is when a loop node has multiple timestamps as descendants. Here the descendants are visited in every iteration, repeating earlier timestamps, thus violating the timestamp order. Even then, we can predict the *common case time order* as the cyclic order, since the end-of-loop backward branch is usually taken. Thus we can use timestamps, at compile-time, to reason about dynamic execution order across the whole program. This is a useful property, and we speculate that timestamps may be useful for other compiler optimizations as well that need to reason about execution order, such as compiler-controlled prefetching [53], value prediction [48] and speculation [22].

Timestamps have their limitations in that they do not directly work for **goto** statements or the insides of recursive cycles; but we have work-arounds for both which are mentioned in chapter 5.

4.2 Allocation of Global and Stack Objects

Before running this algorithm, the DPRG is built to identify program points and mark the timestamps. Next, profile data is dynamically collected to measure the frequency of access to each variable separately for each region. This frequency represents the weight of the edge from a parent node to a child variable. Profiling also measures the average number of times a region is entered from a parent region. This represents the edge weight between two non-variable nodes. The edge weight between a variable and a non-variable represents the access frequency of the variable in the parent region. The total frequency of access of a variable is the product of all the edge weights along the execution path from the *main()* node to the variable.

An overview of the first part of our memory transfer algorithm is as follows. At each program point, the algorithm determines the following memory transfers: (i) the set of variables to copy from DRAM into the scratch-pad and (ii) the set of variables to evict from DRAM to the scratch-pad to make way for incoming variables. The algorithm computes the transfers by visiting each program point (and hence each region) once in an order that respects the partial order of the timestamps. For the first region in the program, variables are brought into the scratch-pad in decreasing order of frequency-per-byte of access. Thereafter for subsequent regions, variables currently in DRAM are considered for bringing into the scratch-pad in decreasing order of frequency-per-byte of access, but only if a *cost model* predicts that it is profitable to do so. Variables are preferentially brought into empty space if available, else into space evicted by variables that the compiler has proven to be dead at this point, or else by evicting live variables. Completing this process for all variables at all timestamps yields the complete set of all memory transfers.

The order in which different regions are visited by our method is guided roughly by the timestamps of the regions. It is important to note that such reliance on the timestamps does not have any correctness impact. Timestamps are used only as an indicator of the common-case ordering between regions. No assumption is made that the execution order at run-time is the same as the timestamp order. Moreover, since timestamps define a partial order and not a total order, no common-case ordering is assumed in the case of incomparable nodes For example, nodes such as ones under the *then* part of some *if-header* node, and the *else* part of the same *if-header* do not have any relative ordering. In such cases, our method derives allocation for each branch independent of the other branch. The details are discussed in chapter 5. Thus, using the timestamps to guide allocation does not have any correctness impact.

The cost model works as follows. Given a proposed incoming variable and one-or-more variables to evict for the incoming variable, the cost model determines if this proposed swap should actually take place. In particular, copying a variable into the scratch-pad may not be worthwhile unless the cost of the copying and the lost locality of evicted variables is overcome by its subsequent reuse from scratch-pad of the brought-in variable. The cost model we use models each of these components to derive if the swap should occur.

Detailed algorithm Algorithm 1 describes the above algorithm in pseudo-code form. A line-by-line description follows in the rest of this section.

Algorithm 1 begins by declaring several compiler variables. These include V-fast and V-slow to keep track of the set of application variables allocated to the scratch-pad and DRAM, respectively, at the current program point. Bring-in-set, Swap-out-set and Retain-in-fast-set store their obvious meaning at each program point. Dead-set refers to the set of variables in V-fast in the previous region whose lifetime has ended. The frequency-per-byte of access of a variable in a region, collected from the profile data, is stored in freq-per-byte[variable, region].

We now consider the top level function **Memory-allocator**. Line 12 is the main **for** loop that steps through all the subsequent program points in timestamp or-

Algorithm 1 Algorithm for determining dynamic memory allocation

```
Define    ▷ The values of all of the quantities defined below change at each program point
Set V-slow    ▷ Set of variables in DRAM at this point
Set V-fast    ▷ Set of variables in the scratch-pad at this point
Free space    ▷ Free space in scratch-pad memory
Set Bring-in-set    ▷ Variables to bring into the scratch-pad at this program point
Set Swapout-set    ▷ Set of variables for eviction to DRAM
Set Retain-in-fast-set    ▷ Set of variables to retain in the scratch-pad
Set Dead-set    ▷ Set of variables in V-fast whose lifetimes have ended
float freq-per-byte[variable,region]    ▷ Access frequency per byte of variable in region in profile data
procedure MEMORY-ALLOCATOR
    initial-candidate-list ← Sort variables accessed in first region in decreasing order of freq-per-byte[variable, first region
    for all timestamped program points in the application visited in partial order of their timestamps, starting at second region do
        Swapout-set ← NULL_SET; Bring-in-set ← NULL_SET; Retain-in-fast-set ← NULL_SET
        Free-space = Free-space + sizeof(Dead-set)
        for all variables V accessed in this region in decreasing order of frequency-per-byte do
            Consider-for-Vfast(V)    ▷ Check if V can be allocated into scratch-pad. If so, update various
        end for
        V-fast ← V-fast ⋃ Bring-in-set − Swapout-set − Dead-set
        V-slow ← V-slow ⋃ Swapout-set − Bring-in-set − Dead-set
        Store V-fast and V-slow for this region
        Dead-set ← Variables which are no longer alive at this program point
    end for
    return
end procedure
```

```
procedure CONSIDER-FOR-VFAST(V)
    if V $\in$ V-slow then
        if sizeof(V) $\leq$ Free-space then        ▷ V fits no need to swapout variables
            Benefit-of-bring-in-V $\leftarrow$ Find-benefit(V, NULL_SET)
            if Benefit-of-bring-in-V $>$ 0 then
                Bring-in-set $\leftarrow$ Bring-in-set $\bigcup$ {V}
                Free-space $\leftarrow$ Free-space $-$ sizeof(V)
            end if
        else                                       ▷ V does not fit; try to swap out variables
            Swapout-set-for-V $\leftarrow$ Find-swapout-set(V)
            if Swapout-set-for-V $\neq$ NULL then
                Benefit-of-bring-in-V $\leftarrow$ Find-benefit(V,Swapout-set-for-V )
                if Benefit-of-bring-in-V $>$ 0 then
                    Bring-in-set $\leftarrow$ Bring-in-set $\bigcup$ {V}
                    Swapout-set $\leftarrow$ Swapout-set $\bigcup$ Swapout-set-for-V
                    Bring-in-set $\leftarrow$ Bring-in-set $\bigcup$ {V}
                    Free-space $\leftarrow$ Free-space + sizeof(Swapout-set-for-V) $-$ sizeof(V)
                end if
            end if
        end if
    else                                           ▷ V $\in$ V-fast
        if V not in Swapout-set then               ▷ Has not been swapped out so far
      Retain-in-fast-set $\leftarrow$ Retain-in-fast-set $\bigcup$ {V}
        end if
    end if
end procedure
```

```
procedure FIND-SWAPOUT-SET(V)
    Swapout-set-for-V $\leftarrow$ NULL_SET
    Swapout-candidate-list $\leftarrow$ Sort variables in the scratch-pad in ascending order of size. In case of a tie, choose variable with the higher next-timestamp-of-access. Exclude variables that have become dead in this region.
    Swapout-candidate-list $\leftarrow$ Swapout-candidate-list $-$ (Swapout-set $\bigcup$ Bring-in-set $\bigcup$ Retain-in-fast-set)
                                                ▷ Update candidate-list with decisions taken until now
    Size-required $\leftarrow$ sizeof(V) - Free space
    while (((Swapout-candidate $\leftarrow$ next-element(Swapout-candidate-list)) $\neq$ NULL) and (Size-required $>$ 0)) do
        Benefit-of-swap $\leftarrow$ Find-Benefit(V, Swapout-candidate)
        if Benefit-of-swap $>$ 0 then
            Swapout-set-for-V $\leftarrow$ Swapout-set-for-V $\bigcup$ {Swapout-candidate}
            Size-required $\leftarrow$ Size-required $-$ sizeof(Swapout-candidate)
        end if
    end while
    if (Size-required $>$ 0) then          ▷ Could not find required space by swapping out
        return (NULL)                      ▷ Do not swap
    end if
    return (Swapout-set-for-V)             ▷ Found required space by swapping out
end procedure
```

```
procedure FIND-BENEFIT(V,Swapout-candidate)
    Latency-gain ← freq-per-byte[V, this region] ⋆ size(V) ⋆ (Latency_slow_mem − Latency_fast_mem)
    Latency-loss ← freq-per-byte[Swapout-candidate, this region] ⋆ size(Swapout-candidate)* (Latency_slow_mem − Latency_fast_mem)
    Migration-overhead ← Time for copying Swapout-candidate (if modified) to DRAM + Time for copying V to the SRAM
    Benefit-of-swap ← latency-gain − latency-loss − Migration-overhead
    return Benefit-of-swap
end procedure
```

der. At each program point, 15 steps through all the variables, invoking **Consider-for-vfast**. Consider-for-vfast considers if a variable can be allocated into SRAM. Finally, after looping through all the variables, lines 18- 21 update, for the next program point, the set of variables in scratch-pad and DRAM respectively and stores this resulting new memory map for the region after the program point.

Now we consider function **Consider-for-vfast**. So in this function each variable V in DRAM (line 2), it tries to see if it is worthwhile to bring it into the scratch-pad (lines 2- 18). If the amount of free space in the scratch-pad is enough to bring in V, V is brought in if the cost of the incoming transfer is recovered by the benefit (lines 3- 8). Otherwise, if variables need to be evicted to make space for V, the best set of variables to evict is computed by procedure **Find-swapout-set()** called on line 10 and the swap is made (lines 12- 18). If the variable V is in the scratch-pad (line 21), then it is retained in the scratch-pad provided it has not already been swapped out so far by a higher frequency-per-byte variable.

Next we explain **Find-swapout-set()** called in line 10 in **Consider-for-vfast**. It calculates and returns the best set of variables to copy out to DRAM when its argument V is brought in. Possible candidates to swap out are those in scratch-pad, ordered in ascending order of size (line 3); but variables that have already been decided to be swapped out, brought in, or retained are not considered for swapping out (line 4 . Thus variables with higher frequency-per-byte of access are not considered since they have already been retained in scratch-pad. Among the remaining variables of lower frequency-per-byte, as a simple heuristic small variables are considered for eviction first since they cost the least to evict. Better ways of evaluating the swapout set of least cost by evaluating all possible swapout sets are avoided to avoid an increase in compile-time; moreover we found these to be unnecessary since only variables with lower frequency-per-byte than the current variable are considered for eviction. The **while** loop on line 6 looks at candidates to swap out one at a time until the space needed for V has been obtained. A cost model is used to see if the swap is actually beneficial (line 8); if it is the swapout set is stored (lines 9. More variables may be evicted in future iterations of the **while** loop on line 6 if the space recovered by a single variable is not enough. If swapping out variables that are eligible and beneficial to swap out did not recover enough space (line 13), then the swap is not made (line 14). Otherwise procedure Find-swapout-set() returns the set of variables to be swapped out.

Cost model Finally, we look at **Find-benefit()** , called from **Consider-for-vfast** in lines 4, 12 and **Find-swapout-set()** in 7. It computes whether it is worthwhile, with respect to runtime, to copy in variable V in its first argument by copying out variable Swapout-candidate in its second argument. The net benefit of this operation is computed in line 5 as the latency-gain − latency-loss − Migration-overhead. The three terms are explained as follows. First, the latency gain is the gain from having V in the scratch-pad in the next region (line 7). Second, the latency loss is the loss from not having Swapout-candidate in the scratch-pad in the next region (line 3). Third, the migration overhead is the cost of copying itself, estimated in line 4. The overhead depends on the point at which the transfer is done. So the overhead of transfers done outside a loop is less than inside it. We conservatively choose the transfer point that is outside as many inner loops as possible. The choice is conservative in two ways. One, points outside the procedure are not considered. Two, transfers are not moved beyond points with earlier transfer decisions. An optimization done here is that if variable Swapout-candidate in scratch-pad is provably not written to in the regions since it was last copied into the scratch-pad, then it need not be written out to DRAM since it has not been modified from its DRAM copy. This optimization provides functionality similar to the dirty bit in cache, without needing to maintain a dirty bit since the analysis is at compile-time. The end result is an accurate cost model that estimates the benefit of any candidate allocation that the algorithm generates.

Optimizations The well-known dataflow concept of liveness analysis [8] is used to eliminate unnecessary transfers – provably dead variables are not copied back to DRAM; nor are newly alive variables in this region copied in from DRAM to SRAM. Our current implementation only does dataflow analysis for scalars and a simple form of array dataflow analysis that can prove arrays to be dead only if they are never used again. If more-complex array dataflow analysis is included then our results can only get better. In programs where the final results (only global) need to be left in the memory itself, this optimization can be turned off in which case the benefits would be reduced. Such programs are likely to be rare. Typically data in embedded systems is used in a time critical manner. If persistent data is required, it is usually written into files or logging devices. This optimization also needs to be turned off for segments shared between tasks in multithreaded programs.

Another optimization that we consider is ignoring the multiple allocation decisions inside higher level regions and instead adopting one allocation inside the particular region. The static allocation adopted is found by doing a greedy allocation based on the frequency-per-byte value of the variables used in the region. Such an optimization can be useful in cases when transfers are done inside loops and the resulting transfer cost is very high. In such cases allthough our method would guarantee than the high cost can be recouped, it might be beneficial to adopt a simple one allocation for the particular region. This is because the transfer cost for a simple allocation may be smaller than when multiple allocations are present. To aid in making this choice, our method compares the likely benefit from a purely dynamic allocation with a static allocation for the region. Based on the result either the dynamic allocation strategy is retained or the static allocation used for the region.

We observe three desirable features of our algorithm. (i) No additional transfers beyond those required by a caching strategy are done. (ii) Data that is accessed only once is not brought into the scratch-pad, unlike in caches, where the data is cached and potentially useful data evicted. This is particularly beneficial for streaming multimedia codes where use-once data is common. (iii) Data that the compiler knows to be dead is not written out to DRAM upon eviction, unlike in a cache, where the caching mechanism writes out all evicted data.

Algorithmic Complexity The algorithmic complexity is $O(T * V^2)$ where T is the number of timestamps in the program, and V is the number of program objects. This can be explained as follows. Consider function **Memory-allocator**. The outermost loop in line 12 iterates through all the timestamps in the program; this contributes the factor T. For each timestamp, the algorithm then goes through the variable list to check if they can be allocated to the scratch-pad. This is done in line 15. While doing so, it may have to go through all the variables that are already present in the scratch-pad. This is done in the function **Find-swapout-set()**. These two steps contribute the factor due to V. We improve the complexity by restricting the number of timestamps to only some program points.

4.3 Algorithm extension for code objects

Next, we show how the above framework can be extended for allocating program code objects. The three key questions that that need to be answered are: First, at what granularity do we allocate code objects (basic-block/procedures/files); Second, how is an code object represented in the DPRG; and third, how is the algorithm and cost model modified. The first issue we look at is the granularity of the program objects. Like in the case of data objects, the smaller the size of the code objects, the larger the benefits of scratch-pad placement are likely to be. One way of achieving this is to consider code objects in units of basic blocks. However, code generation for allocations at such small granularity is likely to involve introducing too many branch instructions while also precluding the use of existing linker technology for its implementation. The other drawback is that complexity of profiling also increases. Another approach to obtaining smaller sized code objects is to selectively create procedures from nested loop structures in programs since it is profitable to place loops in scratch-pad. This optimization called outlining (inverse of inlining) is available in some commercial compilers like IBM's XLC compiler. Both methods can yield code objects of smaller size but at vastly different implementation costs. For its ease of implementation, we choose outlining to provide small-sized program objects.

The next issue is how to represent the code objects in the DPRG. Since our choice of program objects is at the level of procedures (native or outlined), we attach code objects to parent procedures, just like variables are attached(henceforth called code variable nodes). For simplicity of explantion, we assume promising loops have been already outlined. Later, we will explain how outlining can be integrated into the algorithm itself. Figure 4.3 shows an example of a DPRG which also includes code objects shown as rectangular nodes. Every procedure node has a variable child node representing the code executed before the next function call. For example in figure 4.3, code_A.1 represents all the instructions in proc_A executed before the procedure proc_C is called and code_A.2 represents all the instructions in

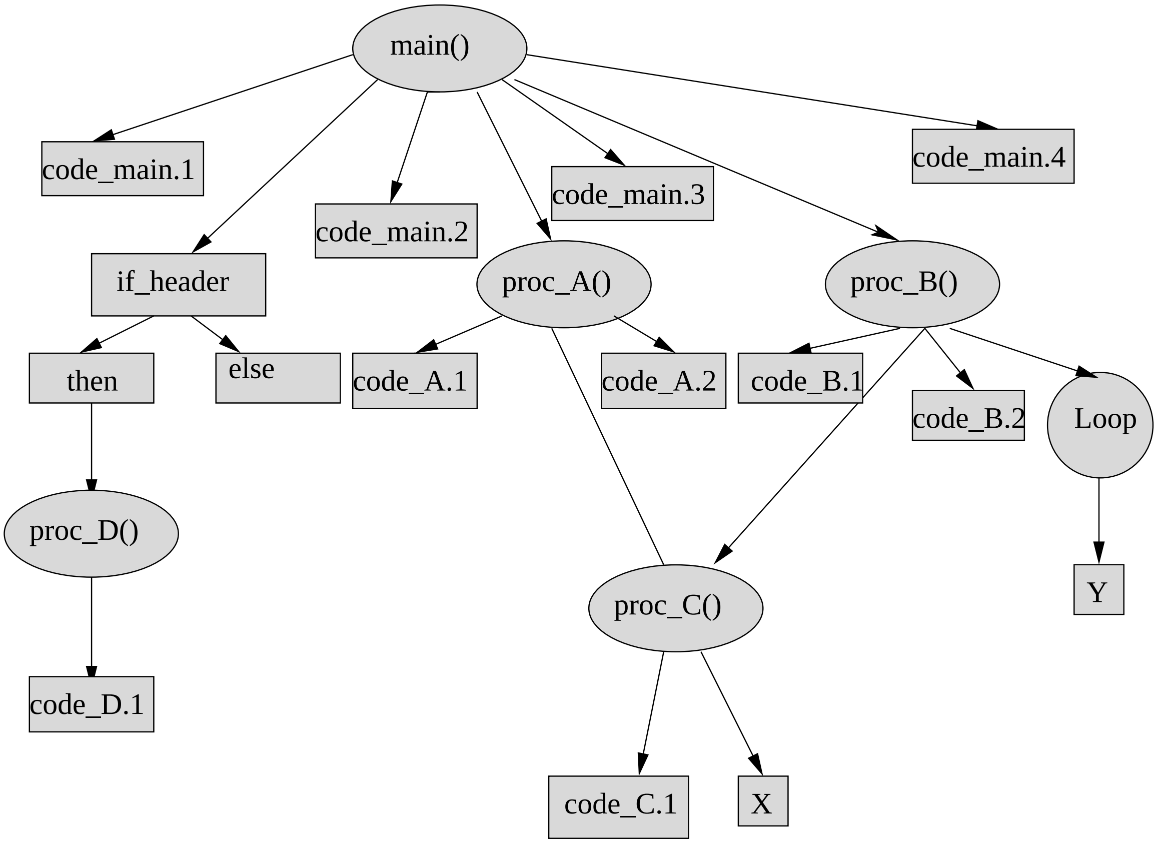


Figure 4.3: Example DPRG with code nodes.

proc_A executed after return from proc_C until the end the proc_A. An advantage of such a representation is that the framework for allocating data objects can be used with little modification for allocating code objects as well. As in the case of data objects, profiling is used to find for every node the frequency of access of each child code variable accessed by that node. For a code variable its frequency is given by its corresponding number of dynamic instructions executed. The size of the code variable is the size of the portion of the procedure until the next call site in the procedure. We also create a modified DPRG structure in which non procedure DPRG nodes other than data nodes have been coalesced into the parent procedure node. We call this new structure the *coalesced-dprg*. Figure 4.3 shows the

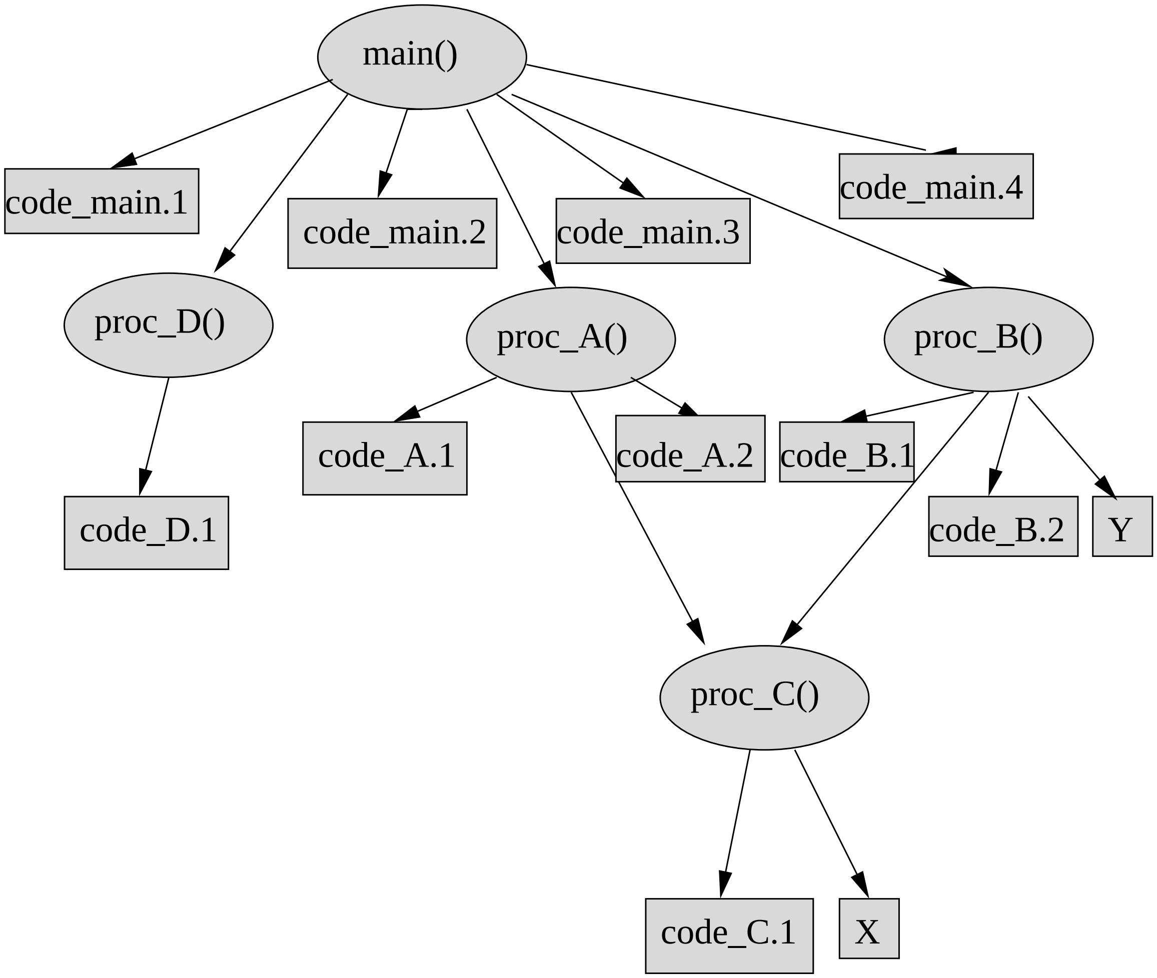


Figure 4.4: An example coalesced DPRG.

coalesced-dprg for the original DPRG in Figure 4.3. This structure does not replace the original dprg; instead it is used only for program codes as described below.

Now the original algorithm described in the previous section is modified as follows. When a procedure node in the DPRG is visited in the original algorithm, first we check if the code node associated can be allocated in the scratch-pad. To determine if a procedure node can be allocated to scratch-pad, it is helpful to use the coalesced-dprg. It suffices to find out if a hypothetical allocation done at the corresponding procedure node using the coalesced-dprg would allocate the procedure node to the scratch-pad. If the procedure gets allocated to the scratch-pad then the available scratch-pad memory is decreased by the size of the procedure node; the rest of the allocation decisions are discarded. Then the algorithm proceeds with the rest of the pseudo-code explained in the previous section using the original DPRG with the only other difference that the procedure node is ignored, that is it is neither considered for swapin or swap out. Thus the modified algorithm would allocate both data and code while retaining the same framework.

The decision to whether outline a particular loop nest as a procedure can be integrated into the algorithm. To do so, only thing required to do is consider outermost loops in the procedure as possible candidates that can be outlined as procedures. Loop-nests in procedure that contain only one loop-nests can be exempted from this modification. Then, in the modified DPRG, procedure nodes representing the code in the loop nest are attached to the outermost loop of a loop-nest. The scope of the extension described in the last paragraph is expanded to include all nodes in the DPRG that have code variables attached to them. So now if the algorithm decides to allocate the code node associated with the loop-nest, then the loop-nest has to be outlined as new procedure. The cost of outlining is included in the cost model.

Chapter 5

Handling Program Features

For simplicity of presentation, the algorithm in the previous chapter leaves some issues unaddressed. Solutions to some issues concerning program features are proposed in this chapter. *All the modifications proposed here are carried out by the algorithm presented before and are driven by the same cost model. They do not define a new algorithm.* An outline of the chapter is as follows. Section 5.1 looks at the issue of join nodes arising out of loops, conditional statements etc. This is followed by a discussion of handling goto statements and recursive functions in section 5.2 and 5.3 respectively.

5.1 Join nodes

One complication with the algorithm in the previous chapter is that for any program point visited along multiple paths (hence having multiple timestamps), the outermost loop visits these points more than once, and thus more than one allocation is made for that program point. An example is node proc C() in figure 4.2. We call such nodes with multiple timestamps *join nodes* since they join multiple paths from main(). Join nodes can arise due to many program constructs including (i) in the case of a procedure invoked at multiple call sites, (ii) at the end of conditional path or (iii) at the start of each loop. For parents of join nodes, considering the join node multiple times in our algorithm is not a problem - indeed it the right thing to do, so that the impact of the join node is considered separately for each parent. However, for the join node itself, multiple recommended allocations result, one from each path to it, presenting a problem. One solution is cloning the join node and the sub-graph below it in the DPRG along each path to the join node, but the code growth can be exponential for nested join nodes. Even selective cloning is probably unacceptable for embedded systems. Instead, our strategy avoids all cloning by choosing the allocation desired by the most frequent path to the join node for the join node. Subsequently, compensation code is added on all incoming edges to the join node other than for the most frequent path. The compensation code changes the allocation on that edge to match the newly computed allocation at the join node. The number of instances of compensation code is upper-bounded by the number of incoming edges to join nodes. We now consider the most common scenarios separately.

Join nodes: Procedure join nodes Our method chooses the allocation desired by the most frequent path to the procedure join node for the join node. Subsequently as discussed before, compensation code is added on all incoming edges to the join node other than for the most frequent path.

Join nodes: Conditional join nodes Join nodes can also arise due to conditional paths in the program. Examples of conditional execution include **if-then**, **if-then-else** and **switch** statements. In all cases, conditional execution consists of one or more conditional paths followed by an unconditional join point. Memory allocation for the conditional paths poses no difficulty – each conditional path modifies the incoming memory allocation in the scratch-pad and DRAM memory to optimize for its own requirements. The difficulty is at the subsequent unconditional join node. Since the join node has multiple predecessors, each with a different allocation, the allocation at the join node is not fixed at compile-time. The solution used is the same as for procedure join nodes and is used for similar reasons. Namely, the allocation desired by the most frequent path to the join node is used for the join node, just as above.

Join nodes: Loops A third modification is needed for loops. A problem akin to join nodes occurs for the start of such loops. There are two paths to the start of the loop – a forward edge from before the loop and a back edge from the loop end. The incoming allocation from the two paths may not be the same, violating the desired condition that there be only one allocation at each program point. To find the allocation at the end of the backedge, procedure **Find-swapout-set** is iterated once over all the nodes inside the loop. The allocation before entering the loop is then reconciled to obtain the allocation desired just after entering the loop – in this way, the common case of the back edge is favored for allocation over the less common forward edge.

5.2 Recursive functions

Our approach discussed so far does not directly apply to stack variables in recursive or cross-recursive procedures. With recursion the call graph is cyclic and hence the total size of stack data is unknown. Hence, for a compiler to guarantee that a variable in a recursive procedure fits in the scratch-pad is difficult. Our baseline technique is to collapse recursive cycles to single nodes in the DPRG, and allocate their stack data to DRAM. Edges out of the recursive cycle connect this single node to the rest of the DPRG. This provides a clean way of putting all the recursive cycles in a black box (not to be considered in the future). Our method can now handle the modified DPRG like any other DPRG without cycles. DRAM placement of stack variables is not too bad for two reasons. First, recursive procedures are relatively rare in embedded codes. Second, a nice feature of this method is that when recursion is present, all program objects other than stack frames of recursive procedures such as data in non recursive descendents and non stack data can still be placed in the scratch-pad by our method.

5.3 Goto statements

Our DPRG formulation in chapter 4 does not consider arbitrary **goto** statements. Because it is widely known that goto statements are poor programming practice they are exceedingly rare in any domain nowadays. Nevertheless, it is important to handle them correctly. We only refer to goto statements here; breaks and continues in loops are fine for DPRGs.

Our solution to correctly handle goto statements involves two steps. First, the DPRG is built and the memory transfers are decided without considering goto statements. Second, the compiler detects all goto statements and inserts memory transfer code along all goto edges in the control-flow graph to maintain correctness. The fundamental condition for correctness in our overall scheme is that the memory allocation for each region is fixed at compile-time; but different regions can have different allocations. Thus for correctness, for each goto edge that goes from one region to another, memory transfers are inserted just before the goto statement to convert the contents of scratch-pad in the source region to that in the destination region. In this way goto statements are handled correctly but without specifically optimizing for their presence. Since goto statements are very rare, such an approach adds little run-time cost for most programs.

The DPRG construct along with the extensions in this section enable our method to handle all ANSI C programs. For other languages, structured control-flow constructs likely will be variants, extensions or combinations of constructs mentioned in this paper, namely procedure calls, loops, if and if-then-else statements, switch statements, recursion and goto statements.

Chapter 6

Layout and Code Generation

Once the allocator has found the desired contents inside different regions, the next step is to assign addresses inside the scratch-pad to different objects allocated to the scratch-pad. The allocation at the end of this phase is the desired dynamic allocation solution. Finally, to use the solution, code then has to be generated that implements the allocation solution. This chapter addresses these issues. First, in section 6.1 it discusses the layout assignment of variables in scratch-pad. Second, in section 6.2 it discusses the code generation for our scheme. Code generation covers two different aspects. First, how to generate code to access the objects allocated into the scratch-pad; and second, how to generate code to transfer the objects between the scratch-pad and the main memory.

6.1 Layout assignment

The first issue we consider in this chapter is deciding where in the scratch-pad to place the program objects being swapped in. A good layout at a region should be able to place most or all of the program objects desired in the scratch-pad by the memory transfer algorithm in chapter 4. To increase the chances of finding a good layout, the layout assignment algorithm should have the following two characteristics. First, the layout should minimize fragmentation that might result when program objects are swapped out, so as to increase the chance of finding large-enough free holes in future regions. Second, when a memory hole of a required size cannot be found, compaction in scratch-pad should be considered along with its cost.

Our layout assignment algorithm runs as a separate pass after the memory transfers are decided. It visits the regions of the application in the partial order of their timestamps. At each region, it does the four tasks described in the following paragraphs.

The first task is concerned with maintaining the data structures involved. An essential input to the pass is the list of free holes and the memory locations occupied by the different objects. Towards maintaining this information, in the first task, the method updates the list of free holes in the scratch-pad by de-allocating the outgoing variables from the previous region.

The second task is concerned with actual assignment of the incoming variable. In this task, the method attempts to allocate incoming variables to the available free holes in the decreasing order of their size. The largest variables are placed first since they are the hardest to place in available holes. Also, as a measure to prevent fragmentation, the assignment considers all the stack objects that are scalars together as one object and does the layout assignment for that object. When more than one hole can be used to fit a variable, the *best-fit* rule is followed: the smallest hole that is large enough to fit the incoming program object is used for allocation. The best-fit rule is commonly used for memory allocation in varying domains such as segmented memory and sector placement on disks [80].

In the third task, when an adequate-sized hole cannot be found for a variable, compaction in the scratch-pad is considered. In general, compaction is the process of moving variables towards one end of memory so that a large hole is created at the end. However, we consider a limited form of compaction that has lower cost: only the subset of variables that need to be moved to create a large-enough hole for the incoming request are moved. Also, for simplicity of code generation, compaction involving blocks containing program objects used inside a loop is not allowed inside the loop. Although compaction is done at run-time, its behavior is completely decided at compile-time; hence, the change of addresses from it cause no problems for code generation. Compaction is often more attractive than leaving the incoming program object in DRAM for lack of an adequate hole; this is because compaction only requires two scratch-pad accesses per word, which is often much lower cost than even a single DRAM access. The cost of compaction is included in our layout-phase cost model; it is done only when its cost is less than the benefit. Compaction invalidates pointers to the compacted data and hence is handled just like a transfer in the pointer-handling phase (chapter 7) of our method. Pointer handling is delayed to after layout for this reason.

Finally as the fourth task, the method in the case that compaction is not profitable attempts to find a candidate program object to swap out to DRAM.

Again, the cost is weighed against the benefit to decide if the program object should be swapped out. If no program object in the scratch-pad is profitable to swap out, our approach decides to not bring in the requested-incoming program object to the scratch-pad. In our chapter on results, chapter 10, we show that this simple strategy is quite effective.

6.2 Code generation

Code generation involves two different aspects. First aspect is generating code such that the variables in the scratch-pad can be accessed. Second aspect is generating copying code to move variables between the scratch-pad and DRAM. In this section we discuss both these aspects.

6.2.1 Code generation for accessing variable in scratch-pad

After our method decides the layout of the variables in scratch-pad memory in each region, it generates code to implement the desired memory allocation and memory transfers. Code generation for accessing variables in scratch-pad in our method involves changing the original code in two ways. **First**, for each original variable in the application (Eg: a) which is moved to the scratch-pad at some point, the compiler declares a new variable(Eg: a_fast) in the application corresponding to the copy of a in the scratch-pad. The original variable a is allocated to DRAM. By doing so, the compiler can easily allocate a and a_fast to different offsets in memory. Such addition of extra symbols causes zero-to-insignificant code increase depending on whether the object formats includes symbolic information in the executable or not. **Second**, the compiler replaces occurrences of variable a in each region where a is accessed from the scratch-pad by the appropriate version of a_fast instead.

Since our method is dynamic, the fast versions of variables (declared above) have limited lifetimes. As a consequence, different fast variables with non-overlapping lifetimes may have overlapping offsets in the scratch-pad address space. Further, if a single variable is allocated to the scratch-pad at different offsets in different regions, multiple fast versions of the variables are declared, one for each offset. The requirement of different scratch-pad variables allocated to the same or overlapping offsets in the scratch-pad in different regions is easily accomplished in the backend of the compiler. Later in the chapter, we will see a very simple implementation to accomplish this by modifying the assembly file of a program.

Although creating a copy in scratch-pad for global variables is straightforward, special care must be taken for stack variables. Stack variables are usually accessed through the stack pointer which is incremented on procedure calls and decremented on returns. By default the stack pointer points to a DRAM address. This does not work to access the stack variable in scratch-pad; moreover the memory in scratch-pad is not even maintained as a stack! Allocating whole frames to scratch-pad means losing allocation flexibility. The other option of placing part of stack frame in scratch-pad and the rest in main memory requires maintaining two stack pointers which can be a lot of overhead. The easiest way to place a stack variable a in scratch-pad is to *declare its fast copy* a_fast *as a global variable but with the same limited lifetime as the stack variable.* Addressing the scratch-pad copy as a global avoids the difficulty that the scratch-pad is not maintained as a stack. Thus all variables in scratch-pad are addressed as globals. Having globals with limited lifetimes is equivalent to globals with overlapping address ranges.

It is instructive to see how exactly the executable changes because of adding new global variables. *Although adding new scratch-pad variables increases the size of the symbol table in the compiler, it does not necessarily increase the size of the executable.* The executable does not usually contain symbolic information. Instead declarations of new scratch-pad variables in the executable appear as labels (hexadecimal addresses) allocated to the scratch-pad range of addresses. Further, in most instruction sets, the uses of those variables appear as references to their labels in the code. In a few instruction sets (*e.g.,* Motorola's MCore [61]) where the addresses of the variables are stored in a memory table and accesses are through that table, the size of the memory table does increase since one word is needed per new added variable. This code size increase is incurred only for these instruction sets (or object formats) and is usually insignificant compared to the size of the executable. Also note that the above approach does not cause any runtime overhead.

Code generation for handling code blocks involves modifying the branch instructions between the blocks. The branch at the end of the block would need to be modified to jump to the current location of the target. This is easily achieved when the unit of the code block is a procedure, by leveraging current linking technology. Similar to the case of variables, the compiler inserts new procedure symbols corresponding to the different offsets taken by the procedure in the scratch-pad. Then it suffices to modify the calls to call the new procedures. The backend and the linker would (without any modifications) then generate the appropriate branches. As mentioned earlier, outlining or extracting loops into extra procedures can be used to create small-sized code blocks. For outlining to work, we promote the local variables that are shared between the loop and the rest of the code as global variables. These are given unique names prefixed with the procedure name. In our set of benchmarks, we observe the overhead due to these extra symbols to be very small.

Implementation of a simple code generator For a wide variety of programs a simple code generator can be implemented with minimal modifications inside the compiler. The implementation relies on the *ABS* section provided by the *ELF* object format. Essentially the ABS section contains all the symbols that have been assigned absolute addresses in the assembly file by a simple assignment $symbolname = address$. The only catch with this section is that the linker does not prevent overlap between symbols. In our case that is not a problem as the condition that no overlaps between symbols exist at a particular point in execution would be ensured by the layout assignment pass. Overlaps between symbols used at different execution points in the program as such does not affect the correctness. The advantage of such a strategy is its ease of implementation. The input to the implementation is the scratch-pad allocation inside different regions of the code. The various steps involved in the implementation are as below.

- Modify source or compiler intermediate code to introduce new global temporaries for every new offset that a program object takes in the scratch-pad.

- Now for every region where a variable is allocated to the scratch-pad, there exists a new temporary variable. Modify the source or intermediate code to use the new temporaries instead of the original variable.

- Generate the new assembly code. Address assignments can be now inserted to modify the starting addresses of different symbols. The starting address used is the address output by the layout pass corresponding to the different temporaries

The executable at the end of the above steps implements the allocation solution.

6.2.2 Memory transfer code

The next aspect of code generation is the memory transfer code. Memory transfers are inserted at each program point to evict some variables and copy others as decided by our method. The memory transfer code is implemented by copying data between the fast and slow versions of to-be-copied variables (Eg: between a_fast and a). This overhead is not unique to our approach – hardware caches also need to move data between scratch-pad and DRAM. Such copying is accomplished using copy functions. The code-size overhead of such copy function is minimized by using a generic optimized copy function. In their simplest form, the copy function takes a source address, destination address and the number of words to copy. In chapter 8, we introduce optimizations as a result of which the addresses of variables copied from need not be contiguous. We still ensure that the elements have a constant stride between them. In such a scenario, we extend the copy function to take a fourth parameter representing the stride. In the simple case that we discussed here, the stride would be 0. In addition, faster copying is possible in processors with the low-cost hardware mechanisms of Direct Memory Access (DMA) such as in ARMv6, ARM7. DMA accelerates data transfers between memories and/or I/O devices. The cost model is modified to take into account various parameters of the DMA hardware like the setup time, minimum transfer limit and block size.

Chapter 7

Handling Pointers

Many compiler optimizations are severely constrained by the presence of pointers. Extensive research in the past two decades has been done on the topic of pointer analysis as a way of managing this problem [38, 75]. All different pointer analysis share the same goal of finding statically what variables may be pointed to by pointers in the program. Such a set of variables is termed the *points-to set*. One simple scheme is to declare a pointer as pointing to all variables whose address has been assigned in the program [38]. Such a scheme is called the *address-taken* scheme.

Presence of pointers to global variables, stack variables and function pointers also causes issues for compiler-time dynamic scratch-pad memory allocators. In this chapter, we study this problem of pointers to global variables, stack variables and function pointers. The goals of this chapter are four fold:

- First, we describe the pointer problem in the context of scratch-pad memory allocation for global/stack and code objects. We explain how different kinds of pointers to globals, stack and program objects pose issues for dynamic allocators.

- Our first part deals with the problem of correctness because of invalid pointers. We propose two solutions- address constraining and pointer translation. We show that the above schemes need only simple points-to information and can be implemented with extremely low overhead. Our results also show that between the two, address constraining does better consistently.

- We also discuss the impact of function pointers on correctness of the code generated. Towards this, we propose a simple compensation strategy that ensures all the memory accesses access the intended addresses.

- Finally, pointers in addition to impacting correctness also impact available opportunities for optimization. We discuss how sophisticated pointer analysis can enable better liveness analysis.

The rest of the chapter is organized as follows. Section 7.1. describes the impact of pointers on correctness in programs. Section 7.2 looks at how presence of function pointers affects generating correct code. Section 7.3 studies the impact of pointers on opportunities for optimizations in dynamic allocation schemes. Section 7.4 summarizes.

7.1 Impact of invalid pointers on program correctness

Function pointers and pointer variables in the source code that point to global and stack variables can cause incorrect execution when the pointed-to variable is moved. For example, consider a pointer variable p that is assigned to the address of global variable a in a region where a is in the DRAM. Later if p is de-referenced in a region when a is in scratch-pad memory, then p points to the incorrect version of a.

Reference parameters A special case of the incorrect pointer problem happens with the use of call-by-reference parameters. A reference parameter is used when the parameter acts as an alias for a caller-provided argument. For correctness, it is essential that the reference pointer points to the correct location of the variable in each of the regions in the procedure.

We now discuss two alternatives for solving this problem.

7.1.1 Pointer translation

The first alternative that we present uses a runtime translator that corrects the address of the pointer variable at points where it is dereferenced. Correcting a pointer means finding out at each region which one among the variables in the points-to set of the pointer is currently being pointed to by the pointer. The translator then takes the current address of the pointer and translates it to the current location of the variable it is pointing to, which may be in a different memory bank because of the dynamic movement of the variables in our method.

To aid in the translation, pointer values are always maintained as DRAM addresses except when they are dereferenced. This is useful for finding which variable is pointed-to by the pointer since every variable has a unique ”home location” in

```
if ($p$ >= DRAM-starting-address-of-obj1 and $p$ < DRAM-end-address-of-obj1)
  $p$=Translate($p$,identifier-of-obj1,DRAM-starting-address-of-obj1)
else ($p$ >= DRAM-starting-address-of-obj2 and $p$ < DRAM-end-address-of-obj2)
  $p$=Translate($p$,identifier-of-obj2,DRAM-starting-address-of-obj2)
```

Figure 7.1: Code fragment with calls to the translate function for pointer p.

DRAM. With the help of a runtime-updated table that keeps track of current locations of every variable, the pointer can be then translated. The table entry is a two tuple – unique integer identifier for a variable (the lookup value) and current location for the variable. The unique identifier can be utilized to generate code to efficiently access the table entry corresponding to it. For a particular variables with identifier i, its entry is located at base-of-table+ (i-1)*size-of(table-entry)

Formally, translation code is constructed with multiple comparisons involving only the intersection set of the points-to set and the variables in the scratch-pad memory in the region. To illustrate, consider a one level pointer p that has points-to set as $\{obj1, obj2, obj3\}$,and inside a particular region p is dereferenced. Let $obj1$ and $obj2$ reside in the scratch-pad memory. The code fragment for translating needed to correct the address of pointer p is shown in figure 7.1. In the code fragment, DRAM-starting-address-of-obj1 is the starting address of the variable $obj1$ while DRAM-end-address-of-obj1 is the end address of the variable $obj1$. Similarly, DRAM-starting-address-of-obj2 is the starting address of the variable $obj2$ while

DRAM-end-address-of-obj2 is the end address of the variable $obj1$. These values can be easily known at compile time. The constants identifier-of-obj1 and identifier-of-obj1 are also compile time known unique identifiers for $obj1$ and $obj2$. These are used inside the **Translate()** function. The details of the function are discussed later.

If the pointer's address does not match with either the DRAM address of $obj1$ and $obj2$, then that means the pointer is pointing to obj3 which resides in DRAM and hence, the pointer value which is a DRAM address is already valid. In this case no translation is required. At the end of the region, similar code is used to retranslate the pointer value to make it point to the DRAM location to. This helps maintain our requirement that the pointer values be always DRAM address except when they are dereferenced. For multilevel pointers, such as **p, such translation has to be done recursively for all levels. The recursion ends when it reaches the end of the pointer chain. In the case of reference pointers, such translation and retranslation needs to be done before and after every region in the function.

The translation involves four steps. First, pointer analysis is done to find the points-to set of different pointers in the program. Second, at statements where the address of a global or a stack variable or a procedure is assigned including when they are passed as reference parameters, the address of the DRAM location of the variable is assigned. This is not hard since all compilers identify such statements explicitly in the intermediate code. As mentioned before, the advantage of DRAM addresses of variables is that they are unique and fixed during the variable's lifetime unlike its scratch-pad memory addresses which can be reused by other variables in

```
**procedure** Translate(p,identifier-of-a,DRAM-address-of-a)
{
    $p$ = current-location-of(identifier-of-a) + $p$ - DRAM-address-of-a
}

**procedure** Retranslate(p,identifier-of-a,DRAM-address-of-a)
{
    $p$ = DRAM-address-of-a + $p$ - current-location-of(identifier-of-a)
}
```

Figure 7.2: Translation and retranslation function.

other regions. Only direct assignments of addresses need handling in this manner; statements that copy address from one pointer to another do not need any special handling. The third step is that at each pointer dereference in the program the pointer address is translated by compiler-inserted code from the DRAM address it contains to the current address of the pointed-to variable. This translation is done by the comparison scheme described in the last paragraph. Once the current base address of the program object in the scratch-pad memory is obtained, the address value may need to be adjusted to account for any arithmetic that has been done on the pointer. This is done by using the function **Translate()** shown in figure 7.2 to adjust the value of the pointer. The expression p - DRAM-address-of-a gives the offset inside variable a, while *current-location-of(identifier-of-a)* retrieves the current address of variable a from the table that we described before.

The final step that we do is that after the dereference the pointer is again made to point to its DRAM copy. Similar to when translating, the pointer value may need adjustments again to account for any arithmetic done on the pointer. But retranslation differs in one way. The difference is that unlike translation no conditional code is required. Instead, the constant identifier of the object p was pointing to can be saved inside a temporary before translation. This can be used again during retranslation for call to the function **Retranslate()** shown in figure 7.2. As before, *current-location-of(identifier-of-a)* retrieves the current address of variable a from the table described before.

It appears that the scheme for handling pointers described above suffers from high run-time overhead since an address translation (and retranslation) is needed at every pointer dereference. Fortunately, this overhead in real programs is actually very low for four reasons. First, if a pointer has been proven to point to heap data alone, it does not require any translation since heap data is not moved by our method. Second, even when translation and hence a subsequent retranslation is needed in a loop (and thus is time-consuming) it is often loop-invariant and can be placed outside the loop. The translation is loop invariant if the pointer variable is single level and is never written to in the loop with a address taken expression.[1] For similar reasons, the retranslation can be done after the loop. Finally, one optimization that can be employed is that in cases where it can be conservatively shown that

[1] Pointer arithmetic is allowed in the loop, however, it is not supposed to change the pointed-to variable, since in ANSI-C, no inter-variable ordering is assumed.

the variable is not moved between the address assignment and pointer dereference, then since it's address does not change, none of the steps discussed is needed. Such instances most trivially happen in cases when for optimized array traversal, the address is assigned to a pointer just before the loop and then the pointer variable is used in the loop.

7.1.2 Address constraining

A second alternative is to use a strategy of restricting the addresses a pointed-to variable can take. The strategy involves the following constraint for correctness: *for all regions where the variable may be accessed through pointers, the variable must be allocated to a common memory location.*. Thus, all the variables in the points-to set of a pointer are restricted in the above manner in all the regions where the pointer is accessed. So for example if variable a has its address taken in region R_1, and may be accessed through a pointer in region R_5 and R_7 then all the regions R_1 , R_5 and R_7 must allocate a to the same memory location. This ensures correctness as the intended and pointed-to memory will always be the same obviating the need for translation. In an extreme case, when the points-to set is all the variables in the program, every variable in all the regions where the pointer is accessed is constrained to one address. We term variables that have to be constrained as *constrained variables* and the addresses they take as *constrained addresses.* In the event that such an address cannot be found by the layout assignment in the scratch-pad memory, then the constrained variable is allocated to the DRAM. On the other hand, all variables not in the points-to set of a pointer or in regions where no pointer is used can be allocated in an unconstrained manner. For reference parameters such constraining has to be done for all the scratch-pad memory addresses the variables can take inside all the regions in the function.

The above scheme also requires some modifications in the address assignment pass that we described in section 6.1. Apart from finding a free hole, the pass also needs to find a consensus address for a constrained variable. We modify the pass to enforce a simple consensus mechanism: the consensus region is decided by the first region which makes the decision and following regions attempt to obey this decision. As in the original algorithm, the modified pass visits all the regions in a timestamped manner. In each region, the pass first checks if any variable has been already assigned a constrained-address in an earlier region. If so, the variable is assigned the same constrained-address. Otherwise, if it is a variable in the points-to set of pointer accessed in the region, then it is assigned a free hole that has not been assigned to any other constrained-variable. After all constrained variables have been assigned addresses, the un-constrained variables are assigned free holes. This assignment is done as before in the best-fit manner. In the event that no hole can be found, compaction is used but not over address ranges occupied by constrained-variables.

7.2 Impact of function pointers on program correctness

Recall that multiple calls to the same procedure can give rise to the problem of procedure-join-nodes. In the scenario where at least one of the calls is using a function pointer, the default solution of adding custom compensation code before the call cannot be used unmodified. This is because the default solution relied on adding the code to immediately prior to the call site and with indirect function calls the location of these call sites is not known precisely at compile-time. In the worst case, pointer analysis may declare that a particular call site can call any of the functions in the program.

The solution of using compensation code for a procedure whose address is taken is modified in two ways. First, the compensation code for paths through function pointers is inserted inside the callee, so that it is executed after the procedure is entered. To enable detection of these paths, an additional parameter in the function parameter list is used. The compensation code is executed when this parameter value is 1. The parameter is set as 0 for calls along precisely known paths (i.e. paths without function pointer calls or when the pointer can be resolved at compile-time). For indirect function calls whose targets are not known precisely, the parameter passed is 1 and the compensation code is thus executed. The second way the solution is different is that unlike precisely known paths the compensation code cannot be customized to a particular allocation pair. Instead, it has to be a generic function that adapts any one allocation to any other allocation. Such a function would first find out using the symbol table which variables are not in the same location as in the new allocation. Then it would evict these and instead bring in the objects desired in the new allocation into the desired locations. If the cost of such compensation cannot be recouped by the benefits from the allocation, then the only alternative might be to allocate all the variables in the function to the DRAM.

7.3 Impact of pointers on liveness

The concept of liveness of a variable or in other words whether a variable is live or not pertains to the interval between the definition of the variable and its last use. In the case of scalar variables, the variable can be even considered dead between its use and its next redefinition. Aggregate objects like arrays have to use a simpler definition. An aggregate object can be declared dead only after its last use in the program. In the presence of pointers this problem of declaring a variables as dead becomes harder . This is because a variable can be declared dead only after its last use directly or through a pointer.

Liveness of a variable is an important question to answer for dynamic allocation methods. Since dynamic allocation schemes achieve better locality with the help of memory transfers, transfers can be reduced by avoiding swapping out data which is dead or not needed again. As variables cannot be assumed dead until all the pointers pointing to it are also dead, in the worst case when a variable can be pointed by all the pointers, the variable has to be considered alive for the whole duration of the program ! Thus, presence of pointers can sometimes mean that this optimization cannot be used. Hence, pointer analysis [38,75] helps in improving liveness analysis, and hence, scratch-pad allocation.

7.4 Summary

In this chapter, we presented low overhead pointer handling add-on's to dynamic allocation schemes that ensure the correctness of benchmarks. We discussed solutions for different issues arising from use of pointers. Towards handling pointers pointing to incorrect locations, we proposed two schemes - pointer translation and address constraining. A detailed evaluation of these two schemes is shown in chapter 10. We give a brief preview of the results here. Our results show that both these schemes are competitive and ensure correctness with little overhead. Between the two schemes, address constraining does better all the time and is a less risk prone strategy with a need for only simple pointer analysis. On the other hand, pointer translation using simple pointer analysis can in some cases do very badly. So barring the availability of sophisticated pointer analysis methods, constraining the address is a better strategy to handle pointers to global, stack data and function pointers.

Chapter 8

Framework For Partial Variable Optimizations

As discussed in chapter 2, compiler schemes are constrained to treating the memory contents of terms of data types. So unlike hardware that can treat data in granularity of cache lines independent of the sizes of the program objects, compiler schemes have to treat data in accordance with the size of the variable. This has both advantages and disadvantages. The disadvantage for hardware systems is that even if a variable spans only part of a cache line, the whole cache line has to be brought in. In modern day systems with large register sets, this is not such a serious issue. Most variables smaller than a cache line (typically 8-32 bytes) are allocated to registers. On the other hand, the disadvantage because of this constraint for software schemes can exact a severe performance loss. In the case of compiler schemes for scratch-pad management such a restriction means that if a program variable cannot fit in the scratch-pad, then it cannot be allocated to the scratch-pad. This is a serious drawback especially for applications that process large variables.

Some solutions have been proposed to alleviate this drawback for array variables. These solutions use compiler analysis to generate new array variables that are part of existing program arrays. These partial arrays are such that they can fit into the scratch-pad. Unfortunately, the solutions that have been proposed so far for dynamic memory allocation for scratch-pad memory systems have offered only one of the two following features. First feature is being optimized for programs that have tightly nested loops and a special category of array references known as affine references. Affine references are array references that are linear functions of the enclosing induction variables. For example A[2i+j] is affine while A[2*i*j] is not. For simplicity, we term programs with only such references as affine programs. The second feature is being applicable for all kinds of programs with arbitrary control structures and references in them. Although the wide variety of applications on embedded platforms calls for general solutions, the presence of large amount of media/signal processing applications that contain affine references means optimizations for such applications cannot be overlooked either. A naive solution is that the designer having access to both kinds of strategies, uses heuristics to choose the one that works best for his application. However, such an approach is not only expensive but fraught with other issues like longer design time.

In this chapter, we for the first time propose a tightly integrated framework that is optimized to handle parts of variables while retaining the features of our general framework like handling non-affine references, arbitrary control flow and scalar variables. We have extended the general framework proposed so far in this thesis to enable it to accept partial variables. We show the effectiveness of the enhanced framework by incorporating an initial data transformation pass that creates partial variables from parts of an array. For instance when the whole array does

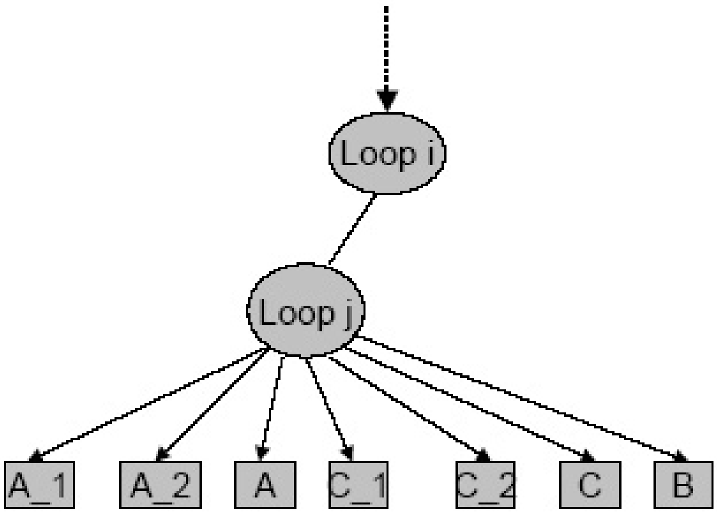


Figure 8.1: Part of a DPRG with partial variables

not fit into the scratch-pad, a row or a column can perhaps be considered as a new variable. Thus the pass essentially identifies the footprint of array references inside loop nests.[1]. In general, any such pass that generates new partial variables can be used. Towards showing this generality, we outline how another existing optimization *structure splitting* [64, 78] could be adapted to work inside our framework.

The framework so far is extended in two parts. The first part deals with the data structures. The partial variables from the initial pass are entered as *additional* variables for our framework to consider, in addition to the original program variables, for allocation into scratch-pad memory. The DPRG is modified to include the partial variables. An example of the modified DRPG with partial variables is shown in figure 8.1. The subscripted variables are the partial variables. For example, A_1, A_2 are partial variables belonging to variable A. The other property of

[1] we will formally define what a footprint is later, but for now consider it as the set of elements accessed

these additional variables is that they can be safely allocated without affecting the correctness of the program. The second part of the framework takes the modified DPRG as input. The pass involves modifications to accommodate these partial variables. The modifications include checks to avoid copies of the same array element in the the scratch-pad memory and appropriate code generation for these partial variables. In extending our framework with these two parts, we use several novel features like flexible transfer points and scatter-gather copying for non-contiguous elements that further aid in a improved allocation. The result is a general dynamic scratch-pad allocation strategy that also exploits the presence of affine access patterns in the programs. The beauty of such an architecture is that the framework can accommodate all such optimizations that can generate partial variables.

The rest of the chapter is organized as follows. Section 8.1 looks at the generation of partial variables. In section 8.2 we look at the details of how our framework can be extended to utilize the partial variables generated. Section 8.3 summarizes the chapter.

8.1 Generating partial variables

In this section, we study the problem of generating partial variables. We first discuss an affine analysis pass that can create partial variables from parts of an array. Next, we outline how an existing optimization for cache can be adapted to fit into our framework. Lastly, we discuss how existing loop nest optimizations can aid in our method.

8.1.1 Affine analysis for partial arrays

The affine analysis phase finds partial arrays that can be considered by the dynamic allocator for allocation to the scratch-pad memory. Potentially, for every reference and loop level there can be a partial array. Before a potential partial variable can be considered by the allocator, two questions have to be answered. First, what are the elements in the variable? Two, is it safe to place the partial variable in the scratch-pad memory? Two different partial variables can be intersect in multiple ways. This then can introduce correctness issues because of multiple copies of an array element in the scratch-pad. We call this issue as the *intersection problem*. We look into these issues in more detail.

Finding the elements in a partial variable

The size of a partial variable is due to the set of elements accessed by a reference at a particular loop level. The footprint of a reference r in loop x, *footprint*(r, x), is the set of data elements that can be accessed by the reference r in its enclosing loop x. For simplicity of our analysis, we over estimate this set of elements as the entire set of elements along the direction of the stride. This extended set is called the *spanned-footprint*. So when the set of elements is a subset along a row,column or diagonal, we estimate the extended set as the whole row,column or diagonal. When the set is a subset of a plane, then the extended set is the whole plane itself. In this way we extend the footprint(r,x) to spanned-footprint(r,x). The shape of the spanned-footprint unlike the footprint(r,x) is always rectangular. The benefit of this is that the index for addressing into the spanned footprint can be generated using a constant strides.

Some terminology

We represent the loop nest that we want to examine by vector $\vec{z}$= {z_1,....z_k,..z_n} where {z_1} is the outermost loop index and {z_n} is the innermost loop index. The subscript at a loop nest denotes the nesting level. The nesting level of any loop in $\vec{z}$ is equal to one more than the number of enclosing loops. A particular value of $\vec{z}$ is itself a vector made of the iteration numbers for each of the loops in order of nesting level. We call the vector an *iteration vector*. For instance, for a loop nest z={i,j}, {i=0;j=10} is an iteration vector. In other ways, z is a collection of iteration vectors where i, j take different values between their respective lower bound and upper bound.

Identifying spanned-footprint

To aid in a more precise identification of the spanned-footprint, we use the concept of data access matrix and data offset matrix. The data access function of each reference can be represented as F= Hz + K where H is the data access matrix and K is the data offset matrix.

To illustrate with an example, consider a reference A[i][2j][2k+10] in a loop nest {i, j, k} where i is the outermost and k is the innermost. The data access function F for this reference decomposed into the form Hz+K is

$$F = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} * \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 10 \end{pmatrix} = \begin{pmatrix} i \\ 2j \\ 2k+10 \end{pmatrix}$$

As a first step in identifying the spanned-footprint for loop z$_k$, our method reduces each data access function F of a reference to a reduced data access function using the following steps. First,H is reduced by setting all the elements of the rows that have any 1 in columns $k <= j <= n$ to zero. H is then further reduced by setting all elements in columns $k <= j <= n$ to 0. The reduced H is denoted H'. Then K is reduced to K' by setting the element of the rows that are all zero in H' to zero. The resulting expression H'z + K' represented by F' is our desired reduced data access function at z$_k$. The reduced access function represents the array subspace affected only by the loop variables {z$_k$,..z$_n$}. This subspace identifies the spanned-footprint(r, z$_k$). In other words, it represents the set of memory locations accessed by the set of iteration vectors {z$_1$,..z$_k$,...,z$_n$} where loop indices {z$_1$,....z$_k$$-$1} are fixed while the rest of the indices vary between their respective loop bounds. Both these methods of looking at a spanned-footprint are useful.

As an example, the reduced data function F' for our earlier example reference at loop j, which identifies spanned-footprint(A[i][2j][2k+10], loop j) is

$$F' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ 0 \\ 0 \end{pmatrix}$$

In general,

$$F' = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

represents element(s) A[x][y][z], except if any of x,y,z are zero, then it is replaced by '*' in the array expression,'*' representing the range of legal values for that subscript index. While the set of elements is called the spanned-footprint, reduced data access function in both the matrix form and the reference form is a way to represent it.

Intersection analysis

The second part of the problem is examining the intersection between different spanned-footprints. The intersection between spanned-footprints can happen in many different ways. The spanned-footprints can be exactly equal, partially intersecting or one can be the subset of the other. Finding the kind of intersection has two uses. First, such an analysis is important for the correctness of the program. If two spanned-footprints intersect and they are both allocated into the scratch-pad memory at different addresses, then multiple inconsistent copies of array elements may exist in the scratch-pad. Second, in the case of equal or subset/superset spanned-footprints, only one variable needs to be generated. This in turn eliminates redundant transfers.

We first introduce some fundamentals that would be handy in our analysis.

Lemma 1. *If the reduced data access functions for two spanned-footprints are equal, then the spanned-footprints are themselves exactly equal.*

Proof. Reduced data access functions when equal means that they define the same set of memory locations that constitute the spanned-footprints. Thus, it follows that the spanned-footprints are equal.

□

We now look at relating how our definition of spanned-footprint relates to the concept of dependence. First we review the concept of data dependence. A data dependence between two references exists when they read or write a common location [3]. Inside a loop, dependence exists between two statements S1 and S2 inside a common loop nest if and only there exist two iteration vectors i and j for the nest, such that (1) $i < j$ or $i = j$ and there exists a path from S1 and S2 in the body of the loop; and (2) Statement S1 accesses memory location M on iteration i and statement S2 accesses location M on iteration j. Dependence due to read accesses to a common memory location is not an issue and hence, we do not consider it further. If one of the references is a write then the dependence can be categorized as being a true/anti/output dependence. For a particular dependence, the reference that first accesses the common memory location is called the source reference and the other reference is called the sink reference. A useful way to characterize a data dependence is based on the number of loop iterations for each loop index separating the two references. This distance is called the dependence distance and is denoted d. Various values of d for a loop index index forms the distance vector. This vector can be further summarized based on if the value of d is greater than, less than or equal to zero and correspondingly, it is denoted as $>$, $<$, $=$. This summary vector is called the direction vector. The direction vector, in other words, indicates which among two references is accessed at an earlier iteration.

To illustrate with an example, consider the dependence in the following loop.

```
for I=1 to N
    for J = 1 to N
        A[I][J]=A[I-1][J]
```

In this case the distance vector between the reference on the right and the reference on the left is $\{1,0\}$ and the direction vector $\{<,=\}$. The number of dependence distances that can exist between two references can be as many different pairs of memory locations in the source and sink. Direction vectors can help in categorizing these distances into one of 3 types, $>$, $<$, $=$. Further, when multiple directions exists between two references, a convenient method is to merge these multiple direction vectors.

Another characteristic of a dependence is its level of the dependence. In a distance vector, the outermost loop that has a non zero distance is said to carry the loop dependence. The level of the loop also indicates the dependence level. Thus, a dependence between S1 and S2 can be denoted as S1 δ_l S2 where l is the dependence level. A loop-independent dependence is denoted as S1 δ_∞ S2.

Using definitions of dependence and spanned-footprints, we now state the following lemma that will be used to determine if two spanned-footprints do not intersect at all.

Lemma 2. *For two references S1 and S2 that are involved in a loop carried dependence such that S1 δ_l S2, the spanned footprints of the references at nesting level greater than l do not intersect at all.*

Proof. Recall that the memory location in a reference is a function of a set of loop index values. The set of loop index values is called an iteration vector. Now from the definition of dependency, a dependency S1 δ_l S2 implies that for S1 and S2, the corresponding iteration vectors that access any common elements do no differ in their inner most $l - 1$ inner loops. The valid set of iteration vectors for nesting level greater than l can only differ in their last $l - 1$ loop indices. But for this set of iteration vectors no common location is accessed between the two references. Hence, the spanned-footprint do not intersect at all. □

To illustrate how the above lemma can be useful, consider the following example.

```
for I=1 to N
    for J = 1 to N
        A[I][J]=A[I-1][J]
```

As mentioned before the direction between the source reference and the sink is vector {<,=} and hence the dependence level is the outermost loop. This means that the spanned-footprint of both these references inside the innermost loop do not intersect at all. From the definition of our spanned-footprint, the spanned-footprints of these two references are the rows A[i] and A[i-1].

To formally determine the kind of intersection between two references, we define four classes of intersection. In the first intersection class, a spanned-footprint of a reference is exactly equal to the other spanned-footprint. In this case we term them as being *common-spanned-footprint references* or in short *CSF references* and the intersection class as *common-spanned-footprint* class. The second class is when a spanned-footprint is a subset of another. The first class can be considered a sub case of the second class as two spanned-footprints that are equal also share a trivial subset relationship between them. For the sake of simplicity, we consider them separate though our algorithm would not make that distinction. The third class is when two spanned-footprints do not intersect at all. The fourth class happens when the spanned-footprints are intersecting but not subset of each other. We term these references *intersecting but not subset* or in short *IBNS references.* We now examine how to first identify and then handle these classes.

The first class we define is when two spanned-footprints are exactly equal ie. for reference r1, r2 and loop variable x, spanned-footprint(r1, x) is equal to spanned-footprint(r2, x). *In terms of the mathematical framework that we have described earlier, two references at a particular loop level which have the same reduced data function have a common spanned-footprint.* For example, references A[i][2j][2k] and A[i][j][2k] have the same spanned-footprint for only loop j, the reduced data access function for loop j being (i, 0, 0). These references from the same common-spanned-footprint class can be represented by one common partial variable.

The second class called the *subset class* occurs when one spanned-footprint is a subset of the other. This is similar to common-spanned-footprint class as the spanned-footprint of the the reference that is super set of the other reference can be used as a common spanned-footprint. The superset reference is then termed the *representative reference*. This class can also utilize the reduced data access function that we defined before.

- Create string by concatenating all the indices of the reduced data access function including their coefficients.
- For every trailing *, convert the corresponding dimension in the other reduced data access function to a *
- If both the strings are equal then they have a subset relationship.

For example, consider spanned-footprints due to A[i][i] and A[i][*], the common string we obtain is "i", and hence the references involve a subset relationship between them.

The third class of intersection is due to spanned-footprints that do not intersect. To find such references, we make use of lemma 2. Two references that are independent at a particular loop level give rise to spanned-footprints that do not intersect. Some cases of partial intersection can be further resolved as independent spanned-footprints. We will consider those cases in the next paragraph.

The fourth class we define is due to intersecting but not subset references and we call it the *IBNS class*. These can simply be identified as ones which do not belong to any of the earlier classes. Such references cause partial intersection. To handle the spanned-footprints from IBNS references in the same way as common-spanned-footprint references needs identifying the precise intersecting elements and accessing them by using conditional code. To avoid such overhead, we instead adopt one of two

```
int A[10][10], B[10][10], C[10][10]
For i=0 to 10 step 1 do
    For j=0 to 10 step 1 do
        C[i][j]=A[i][j]+A[i][j+1]+A[i+1][j-1]+A[i+1][j+1]
        B[i][j] =10
        B[j][i]++
        C[j][j]=10
    EndFor
EndFor
```

Figure 8.2: An example loop with affine references

approaches. In the first approach, references are not considered at all for scratchpad memory allocation. Alternatively if safe to do so, we revert to our default solution of considering these references separately or in other words as belonging to spanned-footprints that do not intersect at all. The choice of approach is based on what kind of dependences the two references involve. The idea of analyzing the dependence is to ensure that there are no data coherence issues. The problem of data coherence arises only when either there is a true dependence or it cannot be clearly established as to which of the two references writes the last value always. The second kind of coherence issue happens if for the loop that carries the dependence multiple dependence directions exist. Such references cause cycles inside a dependence graph. In all other cases, when there is no true dependence and writes to one reference always occur after any writes to the other reference, the references can be handled separately without impacting correctness.

Example To illustrate these ideas, lets consider the loop in figure 8.2. The

Partial	Spanned	Description	Induced by	Size
Variables	Footprint of		CSF ref.	words
A_1	loop j	Row i in A	A[i][j] A[i][j+1]	10
A_2	loop j	Row i +1 in A	A[i+1][j-1] A[i+1][j+1]	10
C_1	loop j	Diagonal in C	C[j][j]	10
C_2	loop j	Row i in C	C[i][j]	10

Figure 8.3: Example output of affine analysis phase

loop has one inner-loop and eight references or a possible set of eight partial variables. The output of the analysis is given in the figure 8.3. An example partial variable is A_1 representing an entire row A[i] of size 10 and induced by references (A[i][j], A[i][j+1]). Formally, the first column gives the four different partial variables prefixed by their parent variable name. Notice that since some spanned-footprints of some references listed in the fourth column are equal, there are only four partial variables. The second column gives the loop to which it belongs. The third column gives a description of the spanned-footprint. The fourth column gives the references which cause the partial variable to be generated. The last column gives the size of the spanned-footprint. Notice that partial variables due to references to variable B are not considered for scratch-pad memory allocation since they intersect partially and involve true dependence between them. On other hand, references to variable C can still be considered for partial handling as there is no true dependence between them and the dependence direction is unidirectional. All these partial variables can be safely considered for allocation in the scratch-pad memory. The modified DRPG part for loop j with these partial variables is shown in 8.1.

The complete algorithm is shown as pseudocode in algorithm 2. An overview of the algorithm is as follows. The algorithm works in two parts. First, all the references for an array are partitioned into multiple classes according to the affine analysis we discussed before. At the end of this, all the IBNS references are filtered out and the rest of the references are grouped into classes. The references in each class can be represented using a common spanned footprint. We call the class as a common-spanned-footprint class. Finally, a representative reference from the class is used to find the size of the common spanned footprint.

In detail, the algorithm defines functions *find-partial-set* that is the outer-level function that does the affine analysis we described before, *Partition references* that partitions the references and *Compute-sizeof-CSF* that finds the size of a common-spanned-footprint class. The return value of *find-partial-set* which is termed *Partial-set* contains all the partial variables generated with information like their sizes, the members and transfer point. At line 5 *Find-partial-set* examines unconsidered references in a nested loop. It first makes a call to *partition-references* that finds different classes of references. Partial variables for these references are then found in a while loop (lines 9- 14). In the while loop, it generates a partial variable to represent the references (line 9. It then picks a representative reference to make a call to *Compute-sizeof-CSF*. A representative reference is the reference whose spanned-footprint is a superset of spanned-footprints of all other references. For references that do not intersect with other references due to which the corresponding CSF-class

Algorithm 2 Algorithm to generate partial variables for affine references

```
procedure Partial-variable-generator
    Define Set Partial-set    ▷ Set of partial variables
    Define Set CSF-Classes    ▷ Set of class CSF-Class. Each CSF-Class stores all the references that make up the class.
    for all loop nests with outer loop o_loop and i_loop as the innermost loop do
        for all loop x_loop from i_loop to o_loop do
            while there are references in Unconsidered-references do
                partition-candidates(r, x_loop) ← Find set of references in unconsidered-references that have same base as r
                ▷ The following sets of classes contain the output of the intersection analysis. Each class in a set thus is in turn a set of references that belong to the class.
                CSF-Classes ← partition-references(partition-candidates((r, x_loop),x_loop))
                ▷ partial variables can be safely generated for equal and non intersecting references.
                while there is an unconsidered class CSF-class in CSF-classes do
                    Generate new variable V(r, x_loop) to represent CSF-class(r, x_loop)
                    Partial_set = Partial_set ⋃ V(r, x_loop)
                    r ← representative reference from CSF-class
                    ▷ representative reference is a reference that is a super-set of all other references.
                    Size(r,x_loop) = Compute-sizeof-CSF(r, x_loop)
                end while
            end while
        end for
        Unconsidered-references = Unconsidered-references - partition-candidates((r, x_loop),x)
    end for
    return (Partial-set)
end procedure
```

```
procedure Partition-references(partition-candidates,x)
    CSF-classes ← NULL
    for all references r in partition-candidates do
        for all references s in partition-candidates do
            if determine-intersection(r,s,x) == IBNS then
                Remove reference s from partition-candidates
                Remove reference r from partition-candidates, if not already removed.
            end if
        end for
    end for
      ▷ All references in partition-candidates can be considered for partial-variable handling.
    for all references r in partition-candidates do
        Create new CSF-Class(r)
        for all references s in partition-candidates do
            if (determine-intersection(r,s,x) == EQUAL ) or (determine-intersection(r,s,x) == SUBSET) then
                Add s to CSF-Class(r)
                Remove reference s from partition-candidates
                Remove reference r from partition-candidates, if not already removed.
            end if
        end for
        Add CSF-Class(r) to CSF-Classes
    end for
end procedure
```

```
1: procedure Compute-sizeof-CSF(r, x_loop)
2:     Set index_set = Index expressions with loop variable x_loop present in them
3:     if size of index_set == 1 then
4:         size_at_current_loop_level = size of dimension
5:     else if size of index_set == 0 then
6:         size_at_current_loop_level = 1
7:     else                         ▷ direction of iteration not parallel to dimension
8:         size_at_current_loop_level=size of dimension
9:     end if
10:    size=size_at_current_loop_level*size(r, x_loop -1)
11:    return (size)
12: end procedure
```

has only one reference, the reference itself is the representative of its class. The function *Compute-sizeof-CSF* returns the size of the the partial variable generated.

Function *Partition-references* partitions references by making the call to function *determine-intersection*. Function *determine-intersection* returns the intersection type between two references at particular loop level based on different rules we described in our intersection analysis. It first, in lines 2 - 10, filters out IBNS references that intersect with other references and cannot be safely allocated to the scratch-pad memory. Next, in lines 10 - 21, it partitions the references into various CSF-classes. A reference that does not have an equal or subset relationship with other references forms a CSF-class by itself.

The Compute-sizeof-CSF function returns the size based on how the particular loop level affects the reference. It first finds the size at the current loop level (lines

3- 4). If the current loop level does not affect the reference (lines 5- 6) , like for reference b[i][i] in a loop with induction variable j,only one element is touched by the reference, then the size at the level is 1. Otherwise (lines 7- 8) if the iteration progress along the row or column or diagonal, then the size of the particular dimension is returned. Line 10 uses the idea that the sizes in different loops inside a nested structure form a hierarchy. Hence the size of a spanned-footprint in a outer loop is product of the size at the current level and size of the spanned-footprint from one level lower.

8.1.2 Adapting structure splitting

In this section, we outline how an existing optimization – *Structure splitting* [64, 78]– can be adapted to work with our framework. Recall that the necessary requirement for our framework is that the optimization generate partial variables. Using this optimizations as an example, we speculate on how the particular optimization can be useful for scratch-pad's as well. The objective of this section is to only illustrate the integration aspects of our framework; the details of how to adapt the optimization is subject of future research.

Structure splitting [64, 78] as shown in the figure involves splitting a structure into hot and cold parts. Hot part comprises of fields that are used frequently and cold part of the structure is made of infrequently used fields. Finally, a pointer field to the cold part is included in the hot part. Figure 8.4 illustrates this optimization. The figure shows the structure S before and after the optimization. S is transformed

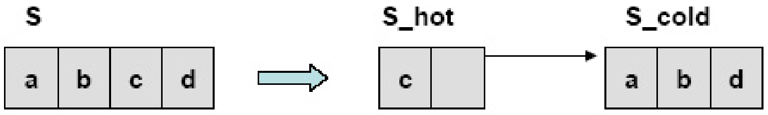


Figure 8.4: Structure splitting optimization

to *S_hot* with only field *c*, and a pointer. The pointer points to the cold part of the structure S_cold with fields *a*,*b* and *d*. For the optimization to work, there are two requirements. First, all accesses to the cold part of the structure elements now have to be accessed through the pointer in *S_hot*. This represents an overhead. Second, the cold pointer has to be initialized to point to the cold part of the structure. For structures that are objects, this is easily done in the constructor of the object [78]. For structures that are statically allocated, such initialization is done at the load-time [30]. For such structures, compiler only needs to annotate the pointers as link-time constants, constant values being the addresses of the cold part of the structure.

Structure splitting for statically allocated structures can be integrated with our framework. Apart from the other exisiting requirements for the optimization to work, two more tasks need to be done. First, partial variables representing the hot part of the structures would need to be generated. Second, the cost of using the partial variables has to be accounted in the cost model. This would include the cost of indirection to access the cold part. The details of the heuristics for constructing the hot and cold is a subject of future research. In our section on future work, section 11, we discuss a general approach to generating partial variables. However, regardless of the heuristics, the frame work with the help of the cost model would ensure that the allocation selected does not degrade the performance compared to when no optimization is used.

8.1.3 Impact of loop transformations

A significant body of research exists on transforming loops to improve both parallelism and locality in the cache. In this section, we study if such transformations can be useful in our framework. We look at cache locality improving transformations, followed by transformations for improving parallelism.

Locality improving transformations

Loop transformations to improve cache locality aim to transform loop structures such that the resulting memory access pattern reduces accesses to the main memory. Loop transformations can improve both temporal and spatial locality. Some examples of such loop transformations are linear loop transformations, loop interchange and loop fusion [3, 55, 74]. The caveat here is that loop transformations are not always legal, and they can affect all arrays in a loop nest, some of them perhaps adversely.

Before we consider how these optimizations may be beneficial in our framework, it is useful to understand how concepts such as locality and reuse hold in the context of scratch-pad memory systems. A reuse occurs whenever the same data item is referenced multiple times. However, reuse by itself does not result in a cache hit if intervening references flush the data item from the cache between uses. If the reused data actually does remain in the cache, we say that the reference that enjoys the cache hit has locality. Therefore, it is important to realize that reuse does not necessarily result in locality. Instead, the references with data locality are a subset of the references with data reuse. Using the same intuition, locality in scratch-pad memory systems can be defined as references that go to the scratch-pad while the program object already exists in the scratch-pad memory. In the case of a cache, when a address is accessed, the cache line associated with it is loaded into the cache. This means that while in the case of caches, temporal reuse necessarily translates to locality, that is not the case for scratch-pad memory systems. If an object is accessed multiple times while being in DRAM, it does not result in locality.

In spite of the differences, these transformations can be generally useful for scratch-pad memory allocation. The change in the access pattern that results from a locality improving transformation can help in two ways. First, such transformations can enable partial variables to be generated and thus enable the use of the partial-variable optimization that we have discussed. Second, they can minimize the transfer cost that is a parameter in the cost model. We now look at some examples of such transformations.

Loop Interchange Loop interchange involves switching the nesting order of two loops in a perfect nest [3, 55].

Consider the following loop

```
for I=1 to N
    for J = 1 to N
        A[J][I]=5
```

Elements along a column can be composed to form a partial variable. The only drawback is that transfer code for such a partial variable cannot make use of any hardware acceleration available for transfer of blocks of adjacent addresses. This drawback can be eliminated by using the interchange transformation.

With an interchange the following loop is obtained.

```
for J=1 to N
    for I = 1 to N
        A[J][I]=5
```

Now the elements being contiguous can utilize block transfers for less overhead transfer.

Loop Fusion Fusion involves merging adjacent loops when such a merging is valid [3,46]. Fusion that improves data locality can also be beneficial for scratch-pad memory. Consider the following loop.

```
for J=1 to N
    for I = 1 to N
        A[i][j] = B[i][j]
    for I = 1 to N
        C[i][j] = A[i][j-1] + 1
```

Let us assume that three rows of any of the arrays can fit into the scratch-pad memory. In spite of that only two rows one each of A and of B can be accommodated during the first inner loop. At the end of each iteration. the rows have to be written into the DRAM. During the second inner loop, the elements of A have to be reloaded again. Such a redundancy can be avoided by using the following fused loop

```
for J=1 to N
    for I = 1 to N
        A[i][j] = B[i][j]
        C[i][j] = A[i][j-1] + 1
```

Thus, we see that various cache optimizations that affect the access pattern can benefit scratch-pad memory allocations. A modified access pattern can enable generation of partial variables and also reduce the transfer

Dependency eliminating transformations

Transformations that improve parallelism are useful as they rely on eliminating dependencies. By eliminating dependencies inside a loop, these transformations can enable more partial variables to be generated resulting in better utilization of the scratch-pad.

One such transformation is loop distribution [3,46]. Loop distributions breaks statements in a loop into statements into separate loops. This can eliminate dependencies. Consider the following loop

```
for I = 1 to N
    for J=1 to N
        A[i][j]= B[i][j] +C
        D[i][j] = A[j][j]
```

The true dependence between the references A[i][j] and A[j][j] will prevent from partial variables being made from the elements in the row of array or elements along the diagonal. The loop can be distributed to eliminate the dependence.

```
for I = 1 to N
    for J=1 to N
        A[i][j+1]= B[i][j] +C
    for J=1 to N
        D[i][j] = A[i+1][j]
```

In the new loop nest shown above, partial variable for array A can be generated inside the inner loop.

Loop transformations as scratch-pad optimizations The examples thus far shown lead to a more general question of how to use these transformations as optimizations for scratch-pad memory. Certainly, it appears that most such optimizations can be beneficial for scratch-pad memory as well. While this may be true in these cases, some examples that are contrary to these exist.

Consider the following example.

```
for I = 1 to N
    for J=1 to N
        if $A[i][j]$
            B[i][j]++
        A[i+1][j]=A[i][j]++ ;
```

Let us assume the capacity of the scratch-pad and cache to be both 2 lines. Also assume that all variables are similar and the cache line size is equal to the size of the row of any of the arrays. In the case of a cache, the cache would suffer capacity misses. A transformation like distribution can alleviate this problem by changing the code to as above.

```
for I = 1 to N
    for J=1 to N
        if $A[i][j]$
            B[i][j]++
for I = 1 to N
    for J=1 to N
        A[i+1][j]=A[i][j]++ ;
```

Now the transformed code would be helpful in the case of a scratch-pad allocation as well if the allocation desires to allocate a row each of A and B into scratch-pad. Now consider the other case when B is not used very frequently. In that case, scratch-pad allocation would desire to allocate only two rows of A into the scratch-pad. In this scenario, the extra cost of the transformed code is unnecessary.

Thus, the question of adapting these transformations for the benefit of scratch-pad allocators needs further study. Due to the large variety of existing optimizations, we have not studied this topic further. We discuss this further in our concluding chapter, chapter 11.

8.2 Framework extensions

We next describe the second part of our integrated framework – the modifications to the framework described in chapter 4. One of the key aspects of integrated framework is that for most part of the algorithm, the additional variables generated can be treated like other variables. This allows us to retain the biggest advantage of non-affine frameworks which is their general applicability. We now look at four aspects of the original framework which have to be modified.

The first modification required is due to the hierarchy of partial variables originating from different loop levels. One partial variable for a particular reference is a subset of another partial variable generated for the same reference but at a higher loop level. The first case that needs to be handled differently is when a partial variable is being considered for swap in, and a variable that includes it is already in the scratch-pad memory then the swap in can be ignored. On the other hand, if after the swap-in has been ignored, the superset variable itself gets swapped out, then the previous decision to ignore the swap-in has to be reversed and the partial variable swapped in. The opposite case that can happen is if a superset variable is to be brought in, and a subset variable already exists, then the subset variable needs to be evicted out.

The second modification is in the cost model. When considering a partial variable that needs any special handling such cost has to be included in the cost model. For instance, if the partial variable is composed out of non-contiguous elements, then the copy code may involve extra overhead. Such overhead also needs to be taken into account.

The third aspect of the original framework that needs different handling is generating the copying code for transfers between the scratch-pad and DRAM. The copying code sometimes needs to collect non-contiguous elements. This is done by adding an additional parameter to the copy procedure. The parameter describes the stride between the elements. For contiguous elements this stride is trivially one. To simplify analysis, we only handle cases where the stride is a constant.

The fourth modification is regarding code generation to access the partial variables. Code generation to access partial variables in scratch-pad memory is done as in the original framework; that is by using a temporary variable.

To illustrate the code generation, consider the following example. In the first example, we look at a simple loop.

```
for I = 1 to N
    for J=1 to N
        A[i+1][j]=B[j][i]++ ;
```

Lets say new partial variables are to be generated for both the references representing the two rows from the array A. Lets call the partial variable A_1 and B_1. Then, the loop would have to modified into the following loop.

```
for I = 1 to N
    memcpy($A_1$,A[i],10,1);
    memcpy($B_1$,B[i],10,10);
    for J=1 to N
        new-index1=j;
        new-index2=j;
        $A_1$[new-index1]=$A_2$[new-index2]++ ;
    memcpy(A[i],$A_1$,10,1)
    memcpy(B[i],$B_1$,10,1)
```

Apart from these changes, the rest of the framework including the allocation skeleton and address assignment remains the same

8.3 Summary

In this chapter we extended our algorithm to a framework that can incorporate optimizations that generates partial variables. To illustrate the workings of the framework the first part of the chapter presented an affine analysis pass that can generate partial arrays in affine code. With this add-on, besides the advantages of an non-affine algorithm like general applicability, the framework can also places parts of variables when they accessed using affine functions. Integration also allows for selecting the transfer point using a cost-model. Our results discussed in detail in chapter 10 show that the integrated algorithm combines the benefits of an affine and non affine allocator. For a wide range of sizes, the integrated algorithm does as well or better than either affine only or non-affine only allocators. In the second part of the chapter, we consider the usefulness of existing optimizations including structure splitting, loop transformations for locality improvement and dependency elimination.

Chapter 9

Related Work

The complete space of existing memory allocation methods is shown in figure 9.1. Broadly, the existing methods to allocate date to on-chip SRAM can be categorized as being either hardware, software or both. For brevity we have grouped all methods that use some hardware as non S/w only methods. Further software methods can be either static or dynamic. Dynamic methods again can be divided into runtime solutions or compile-time solutions. Compile-time solutions can be loop based or whole program based. The existing whole-program methods vary in terms of their scope in yet another dimension. They either handle only data or both data and code.

In this chapter, we look at some examples of existing methods for the different categories illustrated in figure 9.1. Chapter 10 discusses how our methods compares quantitatively with some of these methods. Here, while we survey the different methods, we discuss how they compare qualitatively with our method. The chapter is organized in terms of the broad categories shown in the figure. The chapter has two main sections. In section 9.1, we consider the different software methods. Then

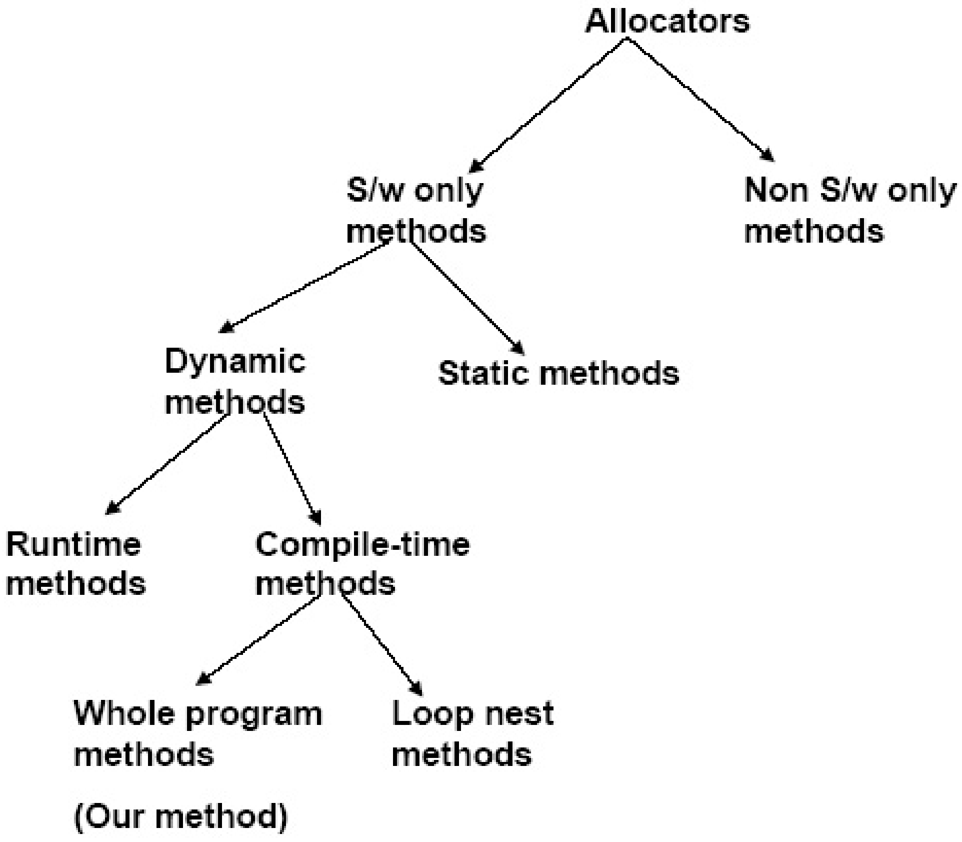


Figure 9.1: Different kinds of scratch-pad allocators

in section 9.2 we consider the other category of methods that require some hardware or are purely hardware based like caching.

9.1 Software methods

First, we study the two different categories of software methods for scratch-pad management – static methods and dynamic methods.

9.1.1 Static methods

Earlier methods by others on allocating a portion of the data to the scratch-pad include [12,13,39,65,72,73,77,88]. The principle limitation of all these methods is that all of them are static methods; hence scratch-pad allocation does not change across run-time. In contrast, our method is a dynamic method which can change the contents of the scratch-pad during the run of the application. Also, most of these methods only handle data but not code. Some methods such as [77, 88] can place both code and data in scratch-pad; however, their allocation does not handle stack variables.

Some schemes for scratch-pad allocation have had different objectives from ours; three such objectives are as follows. First, the primary goal of the method in [39] is to provide an easily re-targetable compiler method for allocation of data across many different types of memories. Second, the goal of the work in [7,77,86] is scratch-pad memory allocation with the primary objective being energy minimization. Third, the goal of the work in [6, 84] is to map the data in the scratch-pad among its different banks in multi-banked scratch-pads; and then to turn off (or send to a lower energy state) the banks that are not being actively accessed.

9.1.2 Dynamic methods

Run-time dynamic methods

One of the earliest dynamic methods proposed for memory management is the seminal work by Belady on replacement algorithms for virtual storage [16]. Although the solution is not particularly targeted for scratch-pad based embedded systems, due to the similarity of the problems the question if the approach can be adapted for scratch-pad allocation arises. The offline paging problem solved by [16] deals with deriving an optimal page replacement strategy when future page references are known in advance. Analogously, we look at finding a memory allocation strategy when program behavior is known in advance. The solution by Belady also proposes the use of ”furthest used” heuristic that we have employed as well using timestamps. Offline paging, however, cannot be used for our purposes since it makes its page transfer decisions at runtime (address translation done by virtual memory), while we need to associate memory transfers with static program points.

As discussed in chapter 9, the problem of register allocation is a special case of the problem of scracth-pad allocation. To revisit briefly, there are two fundamental differences between the problems. First, unlike in the case of memory allocation, the variables that need to be assigned to registers are of the same word size. In many register allocation algorithms, this observation of same-size registers is used to model register allocation as a graph-coloring problem. This formulation cannot directly be used for scratch-pad allocation since the variables in SPM are of varying sizes. Instead the formulation must take into account a more-complex constraint that the sum of the sizes of the variables in SPM must be less than the SPM size. This algebraic constraint cannot be expressed in terms of graph coloring. The second difference is that while in registers, variables cannot be indirectly addressed. Both these differences either completely eliminate some of the issues seen in scratch-pad allocation or reduce their complexity. While problems such as due to pointers, offset assignment are not present in the case of register allocations, problems such as determining the liveness of variables can be done lot more precisely. Due to these basic differences, register allocation solutions cannot be applied directly for scratch-pad allocation. However, two different heuristics used in our method are derived from heuristics used in register allocation methods. First, most of the register allocation solutions [18, 21, 27, 29, 35, 69] also employ some sort of cost model that finds the benefit of having a variable in register versus the cost of the transfers. Second, spill heuritics such as ”furthest used” have also been used in some register allocation methods [66].

Purely dynamic memory allocation strategies to date are mostly restricted to Software Caching [36, 59]. Software caching emulates a cache in SRAM using software. The tag, data and valid bits are all managed by compiler-inserted code at each memory access. Software overhead is incurred to manage these fields, though compiler optimizes away the overhead in some cases [59]. [59] targets the primary cache; [36] manages the secondary cache in desktops. The only software caching scheme that is a competitor is [59]; it however does not measure speedup vs. an all-DRAM allocation, and does not quantify its overheads. All software caching techniques suffer from significant overheads in runtime, code size, data size, energy consumption, and result in unpredictable runtimes. Our dynamic allocation method overcomes all of these drawbacks.

Some software caching schemes use *dynamic compilation* [40] which changes the program at runtime in RAM. These schemes have all the disadvantages of software caching mentioned in the previous paragraph in terms of increased overheads versus our method. Moreover, most embedded systems store the program in unchangeable ROM and dynamic compilation cannot be used. Even if RAM is available to store dynamically compiled code, the run-time and energy cost of translation and the dollar cost of the additional RAM needed for storing the translated code usually make dynamic compilation infeasible for embedded systems. There are other software caching schemes that have been proposed with different goals and/or non-applicable platforms [17,19,26,43,68,85]. None of these apply to our objectives; we do not discuss these further.

Compile-time dynamic methods

Many studies [5,23,52,58,67,76,87] have been proposed that consider dynamic management of the scratch-pad. However, only the strategy by [87] is a generally applicable solution like ours. To better understand the merits and demerits of these various approaches, these can be divided into two categories based on whether scope of the method is at the level of a loop or the whole program. The approaches in [76] and [87] are whole program solutions and are based on ILP formulations with the primary objective of minimizing energy consumption. While Steinke et al in [76] only consider program objects, Verma et al's scheme in [87] is a comprehensive ILP formulation and considers both data and program objects. So for a detailed comparison we do not discuss the approach in [76] further. On the other hand, the solution by Li etal [52] is also a whole program solution for global and stack objects only. The second category of approaches [5, 23, 58, 67] consider each loop nest separately. One common drawback of all these methods is that none of them are one integrated solution that can handle non-affine code while being optimized for presence of affine references. The next few paragraphs will make a detailed comparison of our approach with these methods.

Methods optimizing loop nests

The methods in [5, 23, 58, 67] only allocate global arrays unlike our method which is comprehensive and can allocate global, stack and program code. To compare the global data allocation part of our method with theirs, like our method, these schemes also move data (only global) between DRAM and the scratch pad under compiler control. The primary distinguishing feature of these dynamic methods is that they are focussed on optimizing perfectly nested loop nests for scratch-pad accesses. So these methods while considering each loop independently allocate parts or whole variables accessed in the loop nest into the scratch-pad. The local analysis makes available the entire scratch pad for each loop nest. In contrast, our method is globally optimized for the entire program. Our method is also not constrained to make the entire scratch-pad available for each loop nest; instead it may choose to retain data in the scratch-pad between successive regions thus saving on transfer time to DRAM.

Another limitation of these methods is their limited applicability which stems from these method only targeting arrays accessed through affine(linear) functions of enclosing loop induction variable. Thus, these cannot handle non-affine accesses such as accesses using pointers or indexed array expressions. These methods are also restricted in their use because the required affine analysis for these methods can be only done for only well structured loop without any other control flow such as **if-else**, **break** and **continue** statements. In contrast, our method is completely general and is able to exploit locality for all codes, including unstructured code, code with irregular accesses patterns, variables other than arrays and code with pointers.

We now compare how these methods differ in their handling of affine references. Among these strategies handling affine programs, there are two categories. The first category similar to ours, identifies the footprint of an affine reference. The second category of papers rely on data or code transformations to copy a part of the working set. Some aspects of these techniques such as precise footprint analysis are complementary to our method. One of the strengths of our integrated framework is that if so desired it can easily accommodate any of these optimizations.

We now look into these methods in more detail. The papers by Eisenbeis et al [23] and Schreiber et al [67] find the smallest footprint while Anantharaman et al [5] similar to us find a bounding rectangle around the footprint. These methods are similar to the general class of analysis techniques known as *array section analysis* Several array section analysis techniques have been proposed that vary based on their precision and speed. The precise techniques involve recording each reference to an array seperately without summarizing them [50, 54]. Coarser level techniques also exist that use approximations to represent the section [25, 37]. Compared to our method of aligning footprints with array boundaries, these techniques are likely to yield more precise array footprints.

Certainly, finding the smallest size may improve the utilization of the scratch-pad memory. However, in most of the cases, these section are of non-rectangular shapes and do not align with the array boundaries. The advantage of rectangular sections is that the index into the section can be generated as a function of the base of the section and an offset. This is not always possible for non-rectangular sections. This makes it difficult for use in dynamic allocation strategies for scratch-pad where explicit code has to be generated to address elements in scratch-pad. Thus, code generation becomes extremely complicated for non-rectangular sections and that further hinders optimization of the address generation code.

These solutions also differ from ours in the granularity of the loop nest at which they work. The granularity is in terms of loop iterations [23] or the whole loop nest [5, 67]. The granularity of an iteration may mean not only extra overhead but also large transfer cost due to transfers inside the loop. Now, the footprint at the granularity of an iteration or even several iterations may be unnecessarily small and handling them may even involve extra overhead. Also, the point at which the transfer is done has a large impact on the cost of transfer. So in some cases, it may be more beneficial to transfer the footprint outside as many inner loops as possible, if possible outside the whole loop nest where the transfer cost is minimum. On the other hand, doing the transfers only outside loop nests may not provide any benefit of affine handling for loops that access the entire array like in various stencil benchmarks. In contrast, our method for footprint analysis although with conservative footprints, *generates all possible footprints for different transfer points in the loop nest*. The choice of the footprint and the corresponding transfer point is left to our cost-benefit model driven integrated allocator. The cost-model of the framework is more sophisticated than any of the cost-models used by other affine schemes. The cost-model takes decisions in a more sophisticated fashion by taking note of the memory contents at that point. Finding footprints at different loop levels allows us to leverage this cost-model.

The other category of work that handle affine references is due to Kandemir et al [58]. Their solution involves tiling the loops to reduce working set and then copying it to the scratch-pad memory. Such tiling is illegal for benchmarks with imperfectly nested loops, hand optimized code, arbitrary control structures and it also means that only programs where there is no overlap between successive working sets can be targeted. So the solution flushes out the working set before the next tile can be brought in. The other part of their work determines one common layout for an array based on how it is accessed in different loop nests. We circumvent this issue by using copying code that collects non-contiguous elements and scatters them back into their original addresses when copying back. A drawback of such copying code is that it is likely to be slower especially when special hardware like DMA is available. In such a scenario, it might be more profitable to modify the layout to obtain faster transfers.

Methods using whole-program approach

Two recent papers that are similar to our dynamic approach are by Verma et al [87] and Li et al [52]. Both these methods have strategies motivated by register allocation. An advantage that our method has over both these methods is that it also provides a framework for incorporating optimizations for partial variables. Although, without the implementation it is hard to say how these methods would really perform, in the following paragraphs we attempt to qualitatively compare our method with theirs. First, we describe how these methods work and then look at how they handle various aspects of the allocation.

Method by Verma et al [87] The method by Verma et al is motivated by the ILP formulation for global register allocation [34]. Both parts of the problem – finding what variables should be in the scratch-pad at different points in the program and what addresses they should be at – are solved using ILP formulations. Then like our method, code is inserted to transfer the variables between the scratch-pad and DRAM.

Being ILP based, their solution is likely to be optimal in the solution space they have defined; however, ILP based solutions have two fundamental issues that limit their usefulness. First, ILP formulations have been known to be undecidable in the worst case and in many practical situations are NP hard. Their solution times, especially for large programs can be exponential. Second, using ILP solutions is also constrained by issues like intellectual property of source code from different vendors, maintenance of the resulting large combined code and the financial cost of ILP solvers. Due to these practical difficulties of ILP solvers, it is very rare to find ILP solvers as part of commercial compiler infrastructures despite many papers being published that use ILP techniques.

One other drawback of their approach is that like global register allocation methods, the solution though optimal is per procedure. In other words, the formulation does not attempt to exploit for reuse across procedures. This might lead to some data being needlessly swapped out even if retaining it in the scratch-pad might be more beneficial across two procedures. Also, even if a variable remains in the scratch-pad in both the procedures and can be identified thus, its offsets in both the procedures could be different. So code between two procedures would need to copy data from old offsets to new offsets for all the variables that remain in the scratch-pad. Similar costs exist for register allocation as well, but in the case of register allocation spilling some registers and reloading them again at the end of the procedure is not as expensive. Spilling contents of scratch-pad even if selectively and reloading at the end of procedure is likely to be much more expensive. One, latency to access the next level of storage the DRAM is many orders higher (10-100 times). Two, the size of the scratch-pad is likely to be larger. And finally, selective spilling of only used offsets is not easy because of aggregate variables like arrays. Hence, it is much more important that the copying cost be minimized in the case of scratch-pad memory allocation. In other words, for scratch-pad memory allocation interprocedural approaches are important.

One possible remedy to the problem might be to consider extending the formulation to work with an interprocedural interference graph. This is not practically feasible for the same reasons as why interprocedural register allocation schemes based on interprocedural interference graphs would not work. One, the interference graph is likely to be very large and two, the ILP solution which grows linearly with the number of edges×variables would only become more difficult to solve. Hence interprocedural approaches have to be different from per procedure approaches.

In contrast, our scheme also considers interactions across different regions including procedures. This becomes possible because of our interprocedural approach. Our method's algorithmic complexity is polynomial even in the worst case and hence can find an interprocedural solution efficiently. Consequently, our approach may choose to retain some data between procedures and thus minimize the copying overhead.

An issue that is not addressed in [87] is the issue of correctness in the presence of pointers. To address this, the ILP formulation would need to be extended to include one more extra constraint that the offsets be the same in both the locations namely when the address is taken and when the pointer is dereferenced. The drawback of this is that it would make the formulation still harder to solve. In our scheme the constraining of the offset is included in the greedy layout pass itself. The details can be found in chapter 7. A final drawback of the scheme in [87] is that it does not discuss how the compiler generates code to make use of the allocation decisions.

Method by Li et al [52] Li et al also propose a dynamic strategy motivated by register allocation. Theirs is a graph-coloring approach that works on pseudo registers created from the scratch-pad memory. The method works in 3 parts. In the first part arrays with similar sizes are grouped into a class and their sizes rounded up to a common size. The common size is based on a tunable parameter. Using these sizes, the available memory is partitioned into pseudo registers. Next, a cost model similar to ours is used to identify profitable live ranges. Finally, the graph coloring algorithm is invoked to map the live range and pseudo registers.

The method is an interesting application of graph coloring. But while graph coloring fits the problem of register allocation naturally, in this case a lot of approximations are needed. One such approximation is the use of a user-defined parameter to round of the array sizes to a common size. For example, if the parameter is 64, a variable of 8 bytes is considered to be of size 64. This leads to under utilization of the scratch-pad. Another aspect of the memory allocation problem that makes the graph coloring unsuitable is the scope of the coloring algorithm. While this method propagates live ranges into functions, it does two things that restrict the solution. First, it does not allow the live ranges to be broken at the boundaries of function. What that means is that, a variable allocated to the scratch-pad will have to be retained in the scratch-pad until the call returns. Second, like in the method by Verma et al, the scope of the allocation is a procedure. Thus, an array variable can be allocated to two different pseudo registers in the caller and the callee; thereby requiring code to copy between the registers.

The method unlike ours and Verma et al's does not handle program objects. Lastly, unlike the method by Verma at al, the method by Li addresses the issue of pointers but in a limited way. The paper restricts the breaking of live range in the presence of pointer dereference whose alias set is greater than one. This does not really solve the problem. As mentioned before, pointer also cause problems when a variable is moved since its assignment to the pointer variable.

9.2 Methods using hardware

Schemes also have been proposed that use a hardware approach for managing the scratch-pad [6]. Angiolini et al select a set of memory address ranges based on their access frequency and map them to the scratch-pad. A special decoder is then used to translate the addresses to locations in the scratchpad. The above approach is based on hardware customization instead of software customization like our method. Although hardware customization has the advantage that approaches based on it do not require application source to be available, the applicability of the approach is limited to only architectures which have the required special hardware. An extension of the above approach was proposed by Francesco et al in [33] who used a combination of hardware and software techniques to manage the scratch-pad at runtime. Special hardware needed included a DMA engine and a configurable dynamic memory management. Software support is mainly in the form of high-level API's. Apart from the limitation of special hardware the other drawback of the approach is that it being a runtime approach, it would not be well suited for real time applications with high predictability requirements. One advantage though is that it can adapt to dynamic applications better.

Comparison with caches Other researchers have repeatedly demonstrated [7,13] the power, area and run-time advantages of scratch-pad over caches, even with simple static allocation schemes such as the knapsack scheme used in [13]. Further, scratch-pads deliver better real-time guarantees than caches [89]. In addition, our method is useful regardless of caches since our goal is to more effectively use the scratch-pad memory already present in a large number of embedded systems today such as the Intel IXP network processor, ARMv6, IBM's 405 and 440 processors, Motorola's 6812 and MCORE and TI TMS 370. It is nevertheless interesting to see a quantitative comparison of our method for scratch-pad memory against a cache. Chapter 10 presents such a comparison. Overall, our method does slightly better than caches, but for some benchmarks which deal with large program object that do not fit into the scratch-pad, caches give better results.

Solutions for other scratch-pad based architectures

Some embedded systems allow both a scratch-pad and a cache to be present. Examples of such processor are Intel IXP and IBM 405 processors. For such processors our method is best applied by placing the data as dictated by our method in scratch-pad, *and placing all the remaining data, assumed to be in DRAM in our method, in cached (DRAM-backed) address space instead.* In this way, the real-time improvements from scratch-pad allocation are retained for all frequently used variables. This is not the case with previous methods for cache-aware scratch-pad placement such as [65] and [86] where frequently used variables are sometimes placed in cache; leading to poor real-time bounds for their access. Further, both these methods are static and apply to only some kind of variables. While the method in [65] is limited to global variables, the method in [86] presents an approach for placing instruction traces with the objective of energy minimization. In contrast, our method for run-time reduction is dynamic and applicable for both instruction objects (procedures) and data(global/stack) variables.

Cache designs that can deliver better real-time guarantees than ordinary caches include [28, 47, 63, 70, 79]. Some disallow eviction of parts of the cache data; others restrict parts of the cache to parts of the software; still others assign task priorities for cache usage. Most still suffer the overheads of cache listed in the introduction. Despite the possibility of such caches, the pre-dominant form of SRAM in embedded systems remains scratch-pad memory. Trends in recent embedded designs indicate that the dominance of scratch-pad will likely consolidate further in the future [13,49]. *Hence methods to allocate data to scratch-pad are useful regardless of whether real-time caches capture some share of the market.* Given these facts we do not discuss real-time cache designs further since they are not directly related to our goal of utilizing the scratch-pad better.

Chapter 10

Results

In this chapter, we discuss the performance of our dynamic method. The results are presented in four sections. In the first section, section 10.1, we provide detailed performance results for our basic dynamic method without the partial variable pass. Briefly, the methodology is as follows. We first find the performance comparison between static and dynamic methods for a variety of SRAM sizes. We then choose the size at which maximum benefit is obtained subject to some restriction as our *maximum benefit configuration.* The rest of experiments in this part are done on this size. The experiments study our methods energy and area benefits, influence of parameters like transfer cost and DRAM latency. Then in section 10.2, we present a comparison between our method and hardware caches. The SRAM size for the comparisons is again based on the size found in section 10.1, the size at which maximum benefit is obtained when compared to the static method. Our third set of results – section 10.3– study the effectiveness of our method extended to incorporate an affine analysis pass. We show our comparisons at some selected sizes. For all these experiments we use the *address constraining strategy* of pointer handling. This choice is based on some experiments. We discuss these experiments in our fourth part – section 10.4– of our results.

We first look at the experimental setup for all our results.

Experimental setup The memory characteristics for our studies are as follows. In the experimental setup, an external DRAM with 20-cycle latency and an internal SRAM (scratch-pad) with 1-cycle latency is simulated in the default configuration. To store code, the experimental setup also includes a Flash device [57]. The Flash has a seek time of 120 ns or about 24 cycles [57]. Such a configuration is typical of high end embedded systems where designers have to choose to enable the cache or not. To study the impact of the Flash and DRAM latencies, the latencies are varied later on in an experiment. The current and voltage values of the devices are also incorporated into the power simulator. For faster transfers we assume the availability of DMA hardware.

Our experimental setup for estimating the energy consumption of programs with and without our method is as follows. An M-core power simulator [14, 15], kindly donated by that group, is used to obtain energy estimates for instructions and SRAM. This is an instruction-level power simulator similar to [71,82]; its instruction power numbers were measured using an ammeter connected to an M-core hardware board. DRAM power is estimated by a DRAM power simulator we built into the M-core simulator. It uses the DRAM power model described in [44, 56] for the MICRON external DDR Synchronous DRAM chip. The DRAM chip size is set equal to the data size in the energy model.

Our methodology is as follows. We have implemented the profiling and allocation algorithm in a GCC cross-compiler targeting the Motorola M-Core embedded processor. Our experiments are based on two different versions of Gcc. All our experiments except the experiments described in section 10.3 are based on GCC v3.2. Due to lack of affine analysis support in GCC v3.2, we have used GCC 4.0 for the extending our framework for affine references. After compilation, the benchmarks are executed on the public-domain cycle-accurate simulator for the Motorola M-Core available as part of the GDB v5.3 distribution. DMA is simulated by counting the estimated costs of those mechanisms in the simulator.

10.1 Static method comparison

This section presents results comparing our dynamic method against a static method. As mentioned before, static methods developed so far have been of various kinds. Some have solved the problem when the memory systems has both cache and scratch-pad; some have looked at the problem of energy minimization and others have targeted only global or only code objects. The approaches of these methods either have been ILP formulations or heuristic-driven strategies based on the knapsack problem. Since ILP formulations are not practical in real world compilers, for the static method we choose a heuristic-driven knapsack based solution. Avissar et al. [10] shows that the heuristic to be very competitive compared to the ILP formulation . The heuristic is similar to the the profit-based greedy heuristic for the 0/1 Knapsack formulation. The approach works as follows. First, all the objects are sorted in the descending order based on their frequency/size value. Then, objects are selected until no more objects can fit. The next object that cannot fit is termed the critical object. A comparison is done between the total frequencies of all the objects selected and the frequency of the critical object. In the case that the frequency of the critical object is lesser, then the objects selected until now are retained and make up the static allocation. Otherwise, if the frequency of the critical object is higher, then the critical object is selected and set of objects selected until now are dropped. The scratch-pad size is updated by subtracting the size of the critical object. Also, the critical object is removed from the initial set of objects. The process is repeated with the new set of objects and the new scratch-pad size. The process ends when either of the following two happens. One, all the objects not selected have size greater than the remaining scratch-pad size or two, the total frequency due to all the objects selected is greater than the critical object.

The embedded applications evaluated are shown in table 10.1. The applications selected primarily use global and stack data, rather than heap data, since our method is not meant to handle heap data.

In the experiments below, the SRAM size is varied and for each size the runtime gain from the dynamic method in this thesis vs. the static method is measured. The DRAM and Flash sizes, of course, are assumed to be large enough to hold all program data and code respectively. Other experiments below perform more detailed studies, including varying different parameters, and measure the impact of doing so.

Results on run-time improvement Before presenting results, it is important

Application	**Source**	**Description**	**Lines of code**	**Object code size in KB**	**Data size in bytes**	**Data memory instructions as % total dynamic instructions**
Lpc	UTDSP	Linear predictive coding encoder	493	340	7684	29.1
Edge Detect	UTDSP	Image edge detection	368	350	196600	7.0
Gsm	MIbench	Speech compression	5473	417	20287	29.2
Spectral	UTDSP	Power spectral estimation	449	332	4356	47.5
Compress	UTDSP	Discrete cosine transform	296	324	70752	12.2
G721.Wendyfung (G721)	UTDSP	G.721 ADPCM algorithm	627	250	4148	44.0
Stringsearch	MIbench	String search	2757	242	1572	47.1
Rijndael	MIbench	AES algorithm	1142	315	22160	26.0

Table 10.1: Application programs for comparison with static method.

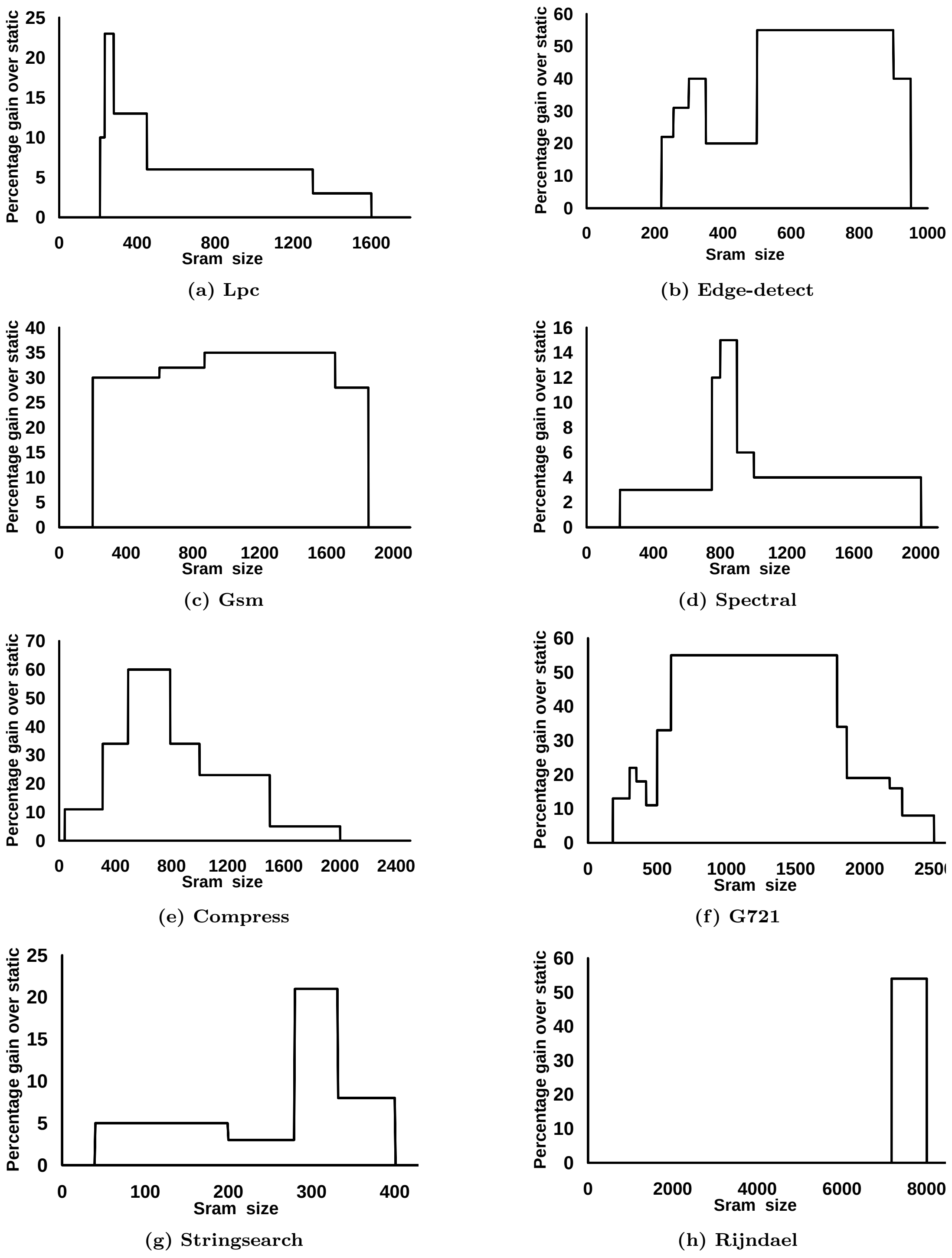
Benchmark	**Useful range of dynamic method**				**Maximum benefit vs. static**	
	Minimum SRAM size (bytes)	Maximum SRAM size (bytes)	Length of range (bytes)	% Accesses to SRAM at max size (%)	SRAM size (bytes)	Run-time gain vs. static (%)
Lpc	210	1600	1390	86	234	23.0
Edge detect	220	950	730	96.0	500	55.0
Gsm	200	1850	1650	85.1	870	35.0
Spectral	200	2000	1800	60.1	800	15.0
Compress	40	2000	1960	96.8	490	60.0
G721	180	2500	2320	98.0	600	55.0
Stringsearch	40	400	360	72.3	280	21.2
Rijndael	7170	8000	830	66.7	7170	54.0
AVERAGE				82.7		39.8

Table 10.2: Useful range of dynamic method and run-time gain vs. static allocation.

to understand that our dynamic method does not give a benefit versus a static allocation for all scratch-pad sizes. This is obvious at the extremes. For a scratch-pad size close to 0% of data+ code size (or object code size), the two methods are equal since neither can put any important data or code in the (absent) scratch-pad. Similarly, at the other extreme of the scratch-pad size when scratch-pad size is close to 100% of program size, both methods are nearly equal since they both fit all the data and code in the scratch-pad. *The benefit from any dynamic method is seen only for intermediate scratch-pad sizes which can fit some but not all of the data and code* *in scratch-pad.* Thus instead of presenting the benefit of the dynamic method for a fixed scratch-pad size, we measure a range of scratch-pad sizes for which our method shows an improvement and by how much.

Table 10.2 shows, for each benchmark, the range of scratch-pad sizes for which our method yields an improvement over the static allocation method; and the maximum improvement in that range. In particular, columns two through four show the range of scratch-pad sizes for which our method improves performance as compared to the static method by at least 1%. Columns two and three present the minimum and maximum of the useful scratch-pad sizes, respectively, for each benchmark. For example, for the *Lpc* benchmark, the dynamic method outperforms the static by at least 1% when the scratch-pad size is between 210 and 1600 bytes. Column four presents the length of the range. Columns five is discussed in the next paragraph. Finally, columns six and seven show the scratch-pad size for which the maximum improvement over the best static allocation is obtained, and the size of the improvement. The maximum improvement configuration shown is not the maximum across all sizes, but is restricted in two ways. First, only scratch-pad sizes which yield a good absolute performance, defined as sizes for which at least 60% of accesses go to the scratch-pad, are considered for finding the maximum gain. Also scratch-pad sizes greater than 20% of the total data size are not considered since they are likely to be too expensive to be used. In this way, we attempt to derive a maximum benefit across feasible scratch-pad sizes only. In the case of range of sizes for maximum benefit, we select the smallest size. The average of the maximum improvements across benchmarks in the last column is 39.8%.

From table 10.2 we can derive two salient conclusions. First, our method yields a significant run-time benefit for many of the commonly occurring small scratch-pad sizes that typically appear in embedded systems (under a kilobyte for most embedded systems; a few kilobytes for some high-end embedded systems). The maximum improvement ranges from 15.0% (*Spectral*) to 60.0% (*Compress*); averaging 39.8%. Second, the reason that larger scratch-pad sizes do not yield a benefit is that larger sizes *are not needed* for our benchmarks. Hence, larger sizes would not be used, and a lack of improvement for those sizes is not harmful. To see why, consider that in most programs a small fraction of the data accounts for a large fraction of the accesses. If the scratch-pad size is large enough to accommodate this frequently used data using the static allocation, the dynamic method yields little run-time improvement for that and larger sizes. This reasoning is verified by column five of table 10.2. Column five shows the percentage of memory accesses that go to scratch-pad for the maximum useful scratch-pad size. *The high average of 82.7% shows that the maximum useful scratch-pad size already has good performance for most benchmarks so a much larger scratch-pad is not needed.* This is further verified by our observation that even doubling the scratch-pad size compared to the maximum useful scratch-pad size in column three yields only an average 0.6% and a maximum 2.0 % improvement (details not shown). This shows that the maximum useful range is already at the point of diminishing returns for our benchmarks; and hence *cost-effective scratch-pad sizes are in the useful range.* The maximum scratch-pad size larger than which diminishing returns are seen is highly application-dependent and is likely to be larger for applications with larger data sets.



(a) Lpc

(b) Edge-detect

(c) Gsm

(d) Spectral

(e) Compress

(f) G721

(g) Stringsearch

(h) Rijndael

Figure 10.1: Runtime gain from our dynamic method vs. static method for different SRAM sizes.

Figure 10.1 shows the same data as in table 10.2 but in more detail. For each benchmark, the X-axis varies the scratch-pad size and the Y-axis shows the run-time gain of the dynamic method in this paper over the static allocation method. From the figure we see two trends. First, for any given scratch-pad size the dynamic method does at least as well as the static method and for the small sizes shown, it often does better. Second, the shapes of the curves follows steps; this is not surprising since those are the discrete points at which the allocation of individual variables in the application changes in either the static or dynamic allocations.

The shapes of the curves in figure 10.1 can be understood by why the up-steps, down-steps and valleys occur. First, an up-step is when the gain from the dynamic method suddenly increases beyond a certain scratch-pad size; an example is for *compress* at scratch-pad size=490 bytes. This happens when an increase in the scratch-pad size enables the dynamic method to accommodate an additional variable in the scratch-pad, perhaps by replacing a lower-frequency variable; while the static method has the same allocation since the additional space is not enough for another variable. Thus, the dynamic method's gain over static increases. Second, a down-step is when the gain from the dynamic method suddenly decreases beyond a certain scratch-pad size; an example is for *compress* at scratch-pad size=790 bytes. This happens when the static method can accommodate some of the additional variables in the dynamic allocation and bridge the gap with it. Third, a valley is when the gain in a certain range is lower than either before or after it; an example is for *Edge-detect* in the range 350-500 bytes in scratch-pad size. A valley is nothing more than an down-step at its start and an up-step at its end. The down-step and up-step occur because of changes in allocations of different variables.

Experiments on maximum benefit configuration The rest of the experiments vary several architectural and method parameters to measure the impact. They are conducted for the smallest scratch-pad size which yields the maximum run-time improvement for our method versus the static method. The reason for this choice is that presenting all the remaining data for all the possible scratch-pad sizes yields a volume of data that is too large to present. Thus we had to choose one scratch-pad size per benchmark to show the underlying reasons as to why our method improves performance. The point of maximum benefit is a good choice to gain such insights.

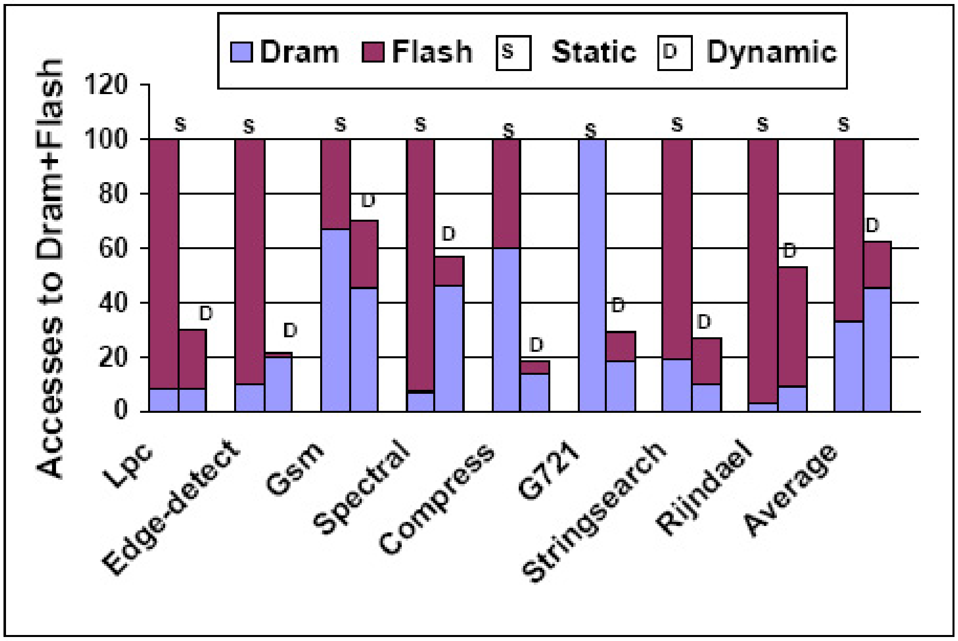


Figure 10.2: Number of memory accesses going to the DRAM and Flash for each benchmark for the maximum benefit configuration.

Figure 10.2 shows the reduction in memory accesses going to the non SRAM

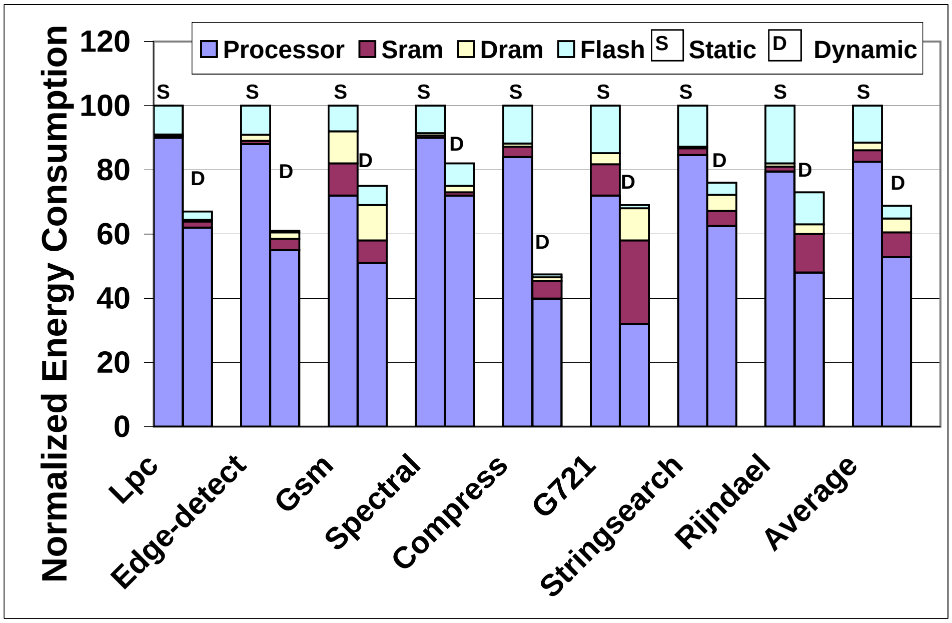


Figure 10.3: Reduction in energy consumption from dynamic method for the maximum benefit configuration.

devices, the DRAM and Flash, because of the improved locality to the scratch-pad afforded by our method. The average reduction across benchmarks is a very significant 38% reduction in Flash+DRAM accesses versus the static allocation. The total number of memory accesses actually increases(not shown) in our method because of the added transfer code, but the reduced number of accesses to Flash+DRAM more than compensates for this increase by delivering an overall reduction in runtime.

Figure 10.3 compares the total system-wide energy consumption of application programs with our method versus the static allocation. Each bar is further divided into the different energy components namely the DRAM, Flash, SRAM and processor energy consumption. *We measured an average reduction of 31.3% in total energy consumption* for our applications by using our method vs. the best static allocation. This number is noteworthy since it refers to total system energy and not just memory system energy. The savings in energy are because our method reduces the number of Flash and DRAM accesses in the program by converting them to SRAM accesses. Flash and DRAM accesses cost more energy than SRAM accesses for two reasons. First, these devices take more energy to access than SRAM banks in our platform. The ratio of DRAM bank energy to SRAM bank energy for a single access is about 5.7:1 in our energy model; but this number is highly implementation-dependent. From data sheets [56, 57], we found that the energy cost of our Flash device to be similar to the DRAM device. So for simplicity, we assume that the ratio of Flash bank energy to SRAM bank energy for a single read access is also about 5.7:1. Second, when a DRAM access occurs in an in-order processor such as our Motorola MCore, the processor is idle while waiting for the DRAM access to complete, but it still dissipates substantial amounts of energy (although slightly less than when instructions are executing). Most embedded processors are in-order; out-of-order processors are rare in embedded systems. Current-day technologies to turn down the processor to a low-energy energy-saving state typically take thousands of cycles to complete. This is infeasible during a DRAM access which typically only takes 10-100 cycles. Thus the processor burns a substantial amount of energy while waiting for a DRAM access. These reasons can be also verified from the figure. From the figure it can be seen that on average the 82.5% of the energy consumption in the static case and 52% of the energy consumption in the dynamic case is consumed by the processor. Consequently, the energy savings in the processor portion by eliminating unnecessary stalls contributes the most to the total savings. Energy reduction in smaller measure is also contributed by reduction in DRAM or Flash accesses or in some cases both. This is accompanied by an increase in the SRAM portion.

Table 10.3 shows some whole program and some region statistics for our benchmarks. Columns two and three show the number of global and stack variables per benchmark. There are a substantial number of them in our benchmarks. Column four shows the code growth (in bytes of code portion) from our method as a percentage of the original code size, primarily because of the inserted transfer code. The average code growth is a modest 1.8% versus the un-modified original code for a uniform memory abstraction; such as for a machine without scratch-pad memory. Column five shows the run time spent in the copy procedure. Column six shows the number of static regions in each benchmark. Column seven shows the average static size of regions in instructions. We see that regions contain about 57 static instructions on average. (In columns six and seven only the regions that are visited at least once during run-time are counted.) Column eight shows the average *turnover fraction* across regions, where the turnover fraction for a region is defined as the amount of *new* data allocated in the scratch-pad for that region expressed as a percentage of the SRAM size. The average turnover fraction is 11.0%; thus on average 11.0% of the scratch pad data is new per region. The relatively low turnover fraction shows that the method is careful not to unnecessarily transfer data: it does so only when beneficial and when it does bring something in scratch-pad, it retains it for several regions before eviction. The turnover is higher for benchmarks such as Rijndael where transfers are carried out inside loops.

Benchmark	**Program statistics**				**Region statistics**		
	# of global variables	# of stack variables	Code growth vs original (%)	% Runtime in copy block	# of regions (instr)	Ave. static size	Turnover fraction (%)
Lpc	17	36	1.0	0.1	44	57.3	9.1
Edge Detect	10	9.0	4.1	0.1	24	64.0	0.6
Gsm	39	382	0.2	4.4	156	58.2	6.3
Spectral	12	33	2.4	1.2	28	45.2	5.6
Compress	8	11	3.2	1.9	39	18.5	1.1
G721.Wendyfung	25	57	2.6	2.0	21	49.1	0.5
Stringsearch	6	9	0.5	16.8	13	13.4	16.0
Rijndael	11	58	0.1	4.9	26	134.8	49.0
AVERAGE	15.5	83.3	1.8	3.9	43.9	57.0	11.0

Table 10.3: Program and region statistics.

Figure 10.4 shows the run-time gain for different data-transfer strategies between the scratch-pad and DRAM. Note that a data transfer involves a call to a copy procedure(13 lines). Also at a program point there may be multiple calls to the copy procedure for different address ranges. The data transfer strategies that are shown for each benchmark are (i) all-software transfer (used in all experiments so far in this paper); (ii) transfers accelerated by DMA; and (iii) a hypothetical zero-run-time (free!) data transfer mechanism. DMA is a hardware mechanism available in some embedded processors and is discussed in chapter 6. Faster transfers can provide additional benefit in two ways. One, due to lower cost of transfers the allo-

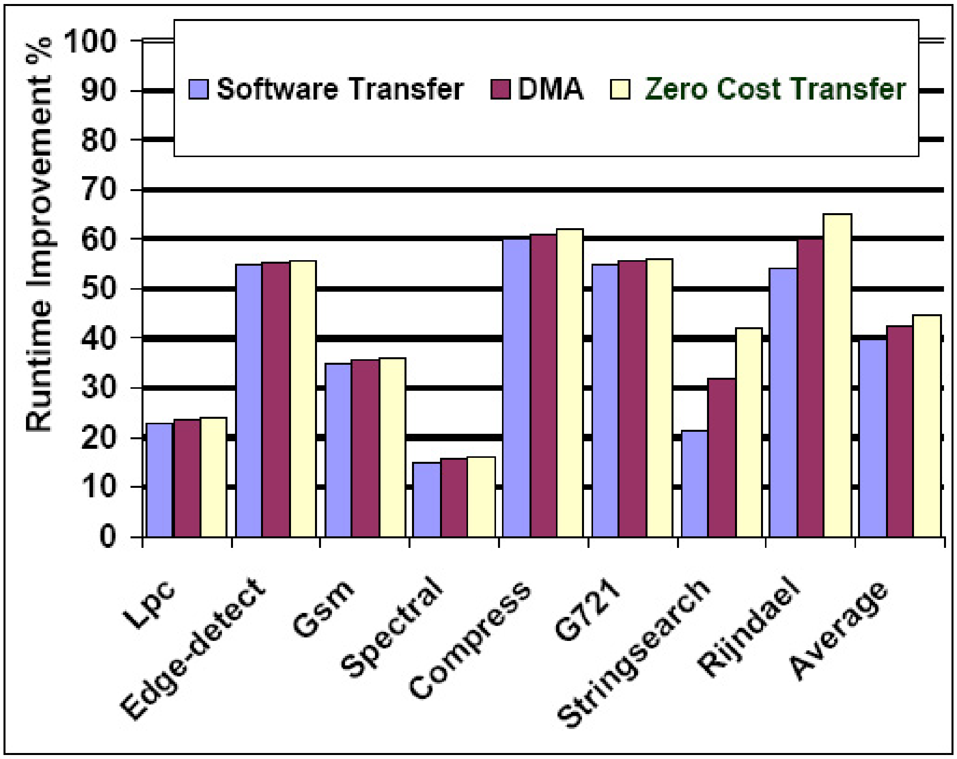


Figure 10.4: Run-time gain for different data-transfer methods for the maximum benefit configuration.

cation method might be able to bring in more variables and in some cases choose a totally different allocation. Two, faster transfers also means that the cost of transfers is lower. But comparing the first and third bar, we see that the run-time gain suffers only by 44.6%-39.8%=4.8% because of transfers. This shows that in general software transfers can deliver good performance. This is because (as we observed) allocations do not change much with faster transfers for our benchmarks. Also, our method is largely successful in transferring data only when doing so yields a benefit; at the same time unnecessary transfers of dead data and non-dirty scratch-pad data are not done. Given these reasons, not surprisingly, the additional run-time gain

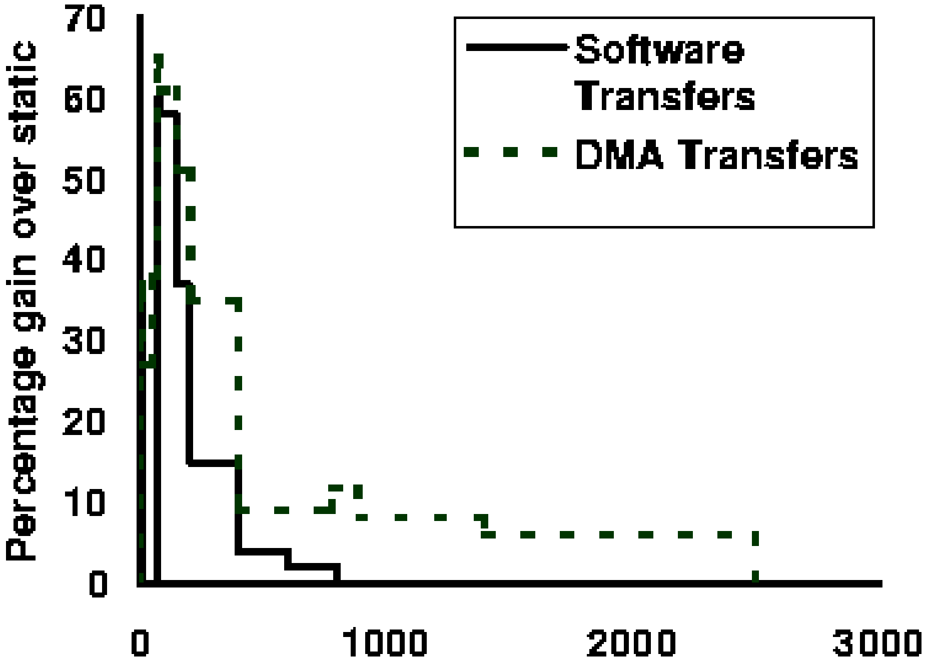


Figure 10.5: Run-time gain for different data transfer methods (with allocations re-computed).

from using faster transfer mechanisms is small: only an additional 2.5% with DMA transfers.

While the runtime increase is modest, another benefit results because of faster transfers. We observed that faster transfers can result in increase in the useful range of SRAM sizes for a benchmark. Figure 10.5 shows the run-time gain for software vs. DMA data-transfer strategies between SRAM and DRAM for the *G721.Wendyfung* benchmark for different SRAM sizes. The figure verifies our claim that the useful range can increase with faster transfers: for *G721.Wendyfung* benchmark the useful range of the dynamic method is increased from 180 to 2500 bytes (all-software) to 180 to 2700 bytes (DMA).

Figure 10.6 shows the effect of increasing Flash and DRAM latency on the run-

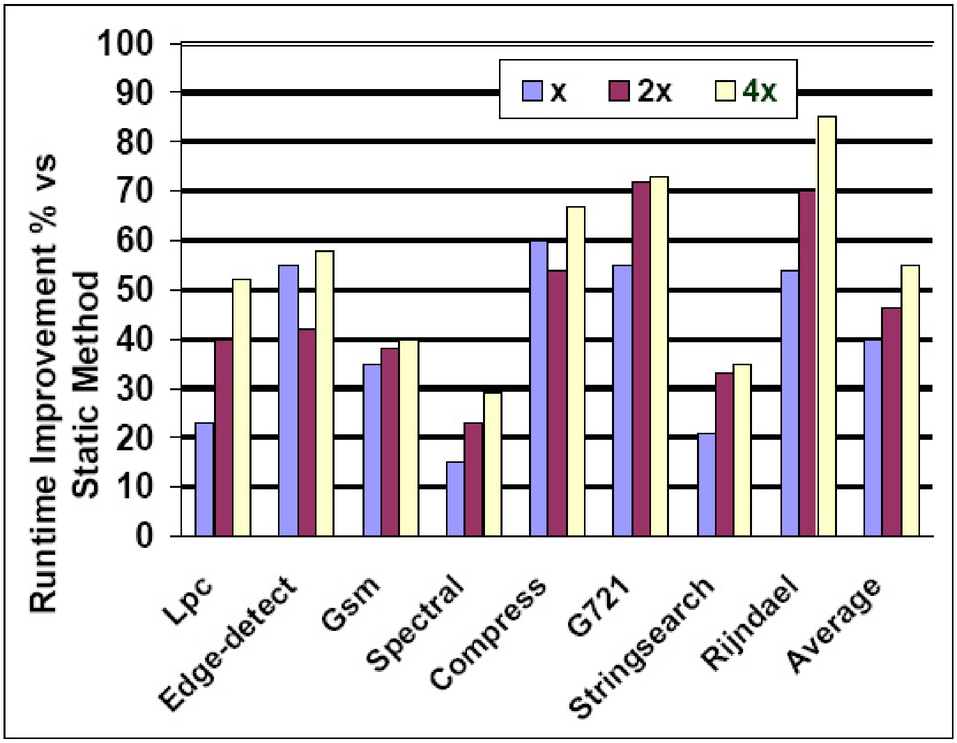


Figure 10.6: Effect of varying DRAM and Flash latencies on run-time gain from our method for the maximum benefit configuration.

time gain from our dynamic method versus the best static allocation. Apart from the original latencies labeled as *x*, the gain is shown for two other latencies namely twice the original termed *2x* and latency four times the original termed *4x*. Recall that the original DRAM and Flash latency is assumed 20 and 24 cycles respectively. Since our method reduces the number of accesses to Flash and DRAM, the gain from our method is greater with higher latencies for most benchmarks. The figure shows that the run-time gain from our method increases from 39.8% with the original DRAM and Flash latencies to 54.8% with latency 4 times the original latencies.

Area benefits A different perspective on the impact of our method can be seen by considering the reduction in area that it can offer to a embedded system designer, who has a desired performance requirement in mind. To measure the reduction in area, we performed a study on the area benefits. For lack of a better heuristic, we measured the area benefits at three different SRAM sizes specified as a fraction of the useful range. These sizes are 25%, 50%, 75% of the useful range. Now we ask, how much additional SRAM size would be needed by the static method to obtain the same runtime as the dynamic method at these sizes. Table 10.4 shows the area benefits of our method over the static method. The table shows the additional SRAM size that would be needed by the static method to match the runtime of the dynamic method at these three different sizes. The additional memory is expressed as a percentage of the SRAM size used by the dynamic method. For some of these benchmarks like Rijndael, Spectral the static method already does as well as the dynamic method at these sizes. This can also be seen in figure 10.1 where for these benchmarks the runtime gain at these points is small. But for the other benchmarks, we measured a decent reduction in area ranging from 5% to a high of 220%. On an average, at these three different sizes of 25%, 50% and 75% we obtain 75%, 43% and 33% reduction in area respectively. All of these are significant when considering that on chip memory uses upto 50% of the total chip area [45]

Efficacy of offset assignment As discussed before, our offset assignment pass is made up of two parts – a best fit memory management along with a limited compaction when an appropriate size hole cannot be found. To study how well this simple strategy performs, we compare it with a perfect address assignment method. The perfect address assignment method is assumed to magically fit all the variables

Benchmark	**SRAM size**		
	25 % of useful range	50% of useful range	75 % of useful range
Lpc	220.0	74.0	28.5
Edge Detect	28.5	57.0	33.0
Gsm	70.0	52.0	23.5
Spectral	5.0	5.5	0.0
Compress	80.0	51.0	30.1
G721.Wendyfung	70.4	37.5	25.0
Stringsearch	60.0	60.7	100.0
Rijndael	66.6	6.7	3.9
AVERAGE	75.0	43.0	33.3

Table 10.4: Additional scratch-pad memory area required by static allocation to match runtime of dynamic method.

at different program points without requiring to sacrifice any variable to fit another variable or using compaction. Further, at every program point it needs only one copy procedure and hence the minimum call overhead. Figure 10.7 shows the degradation of our method compared to a hypothetical method with perfect address assignment pass. Column two of the table shows the run time degradation when compared to the perfect assignment and column three shows the runtime spent in compaction as a percentage of the total runtime. The results show that that our address assignment pass suffers negligible degradation for almost all the benchmarks and on an average suffers a 1.1% degradation. This happens primarily because, program objects do not live in the scratch-pad for too long and hence free holes get easily created. Secondly, fragmentation is almost totally overcome with the help of compaction which at a negligible cost delivers a huge benefit.

Benchmark	**% Runtime Degradation Vs Perfect Assignment**	**% Runtime in Compaction**
Lpc	0.0	0
Edge Detect	8.1	0
Gsm	0.01	0.00002
Spectral	0.04	0.00001
Compress	0.02	0.00002
G721	0	0
Stringsearch	0.5	0
Rijndael	0.0	0
AVERAGE	1.1	

Figure 10.7: Comparison of our address assignment with perfect address assignment.

Profile sensitivity We next measure the profile-sensitivity of our method. Recall that our allocation (like most scratch-pad methods) is based on profile data input. The objective of the experiment is to find out how portable our allocation is across other data inputs.

In figure 10.8 we show the how our method fares in this measure. The experiment uses two data sets. For most benchmarks, these data sets are part of the benchmark suite. In cases when a second data set does not exist, we have used similar data from other benchmark suites. The experimental methodology is as fol-

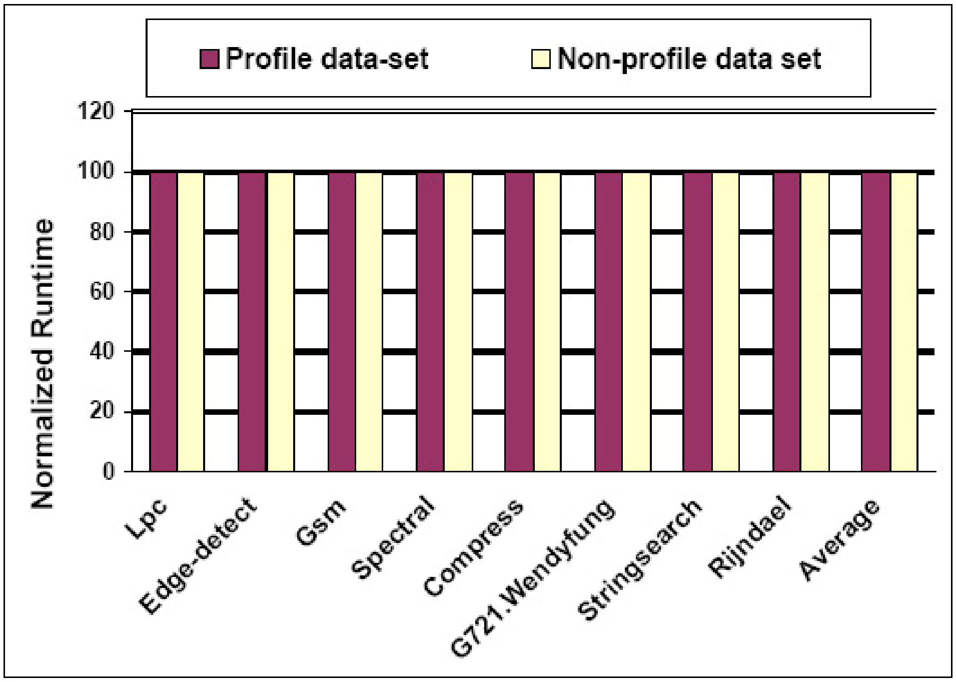


Figure 10.8: Runtime comparison of profiled data-set and non-profile data-sets

lows. First, the benchmark's runtime on profiled data set is obtained by training it on one data input which we call *profile data set*, thus obtaining one profile output. The profile output is used to derive the allocation for the benchmark and then the runtime for the benchmark is measured with the same input data set. This represents the first bar in the figure. Next, the benchmark is trained on a second data set. Using the allocation obtained, the benchmark runtime is obtained by running it on the first data set. This represents the second bar in the figure, which we term *non-profile data set*. The difference between the bars measures the profile sensitivity.

The figure shows that no variation in profile sensitivity happens for any of our benchmarks. While to some extent this might be characteristic of the benchmarks that we have used, we believe it represents a large class of embedded benchmarks whose control characteristics largely depend on the the length of the input and not

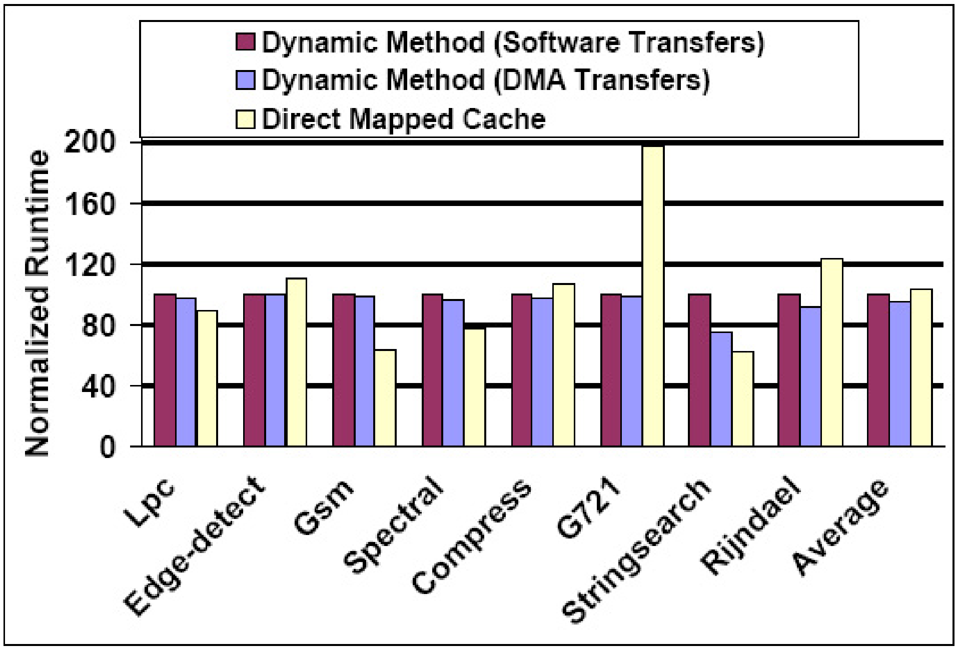


Figure 10.9: Normalized run time for a cache only and scratch-pad only architecture measured for maximum benefit configuration

the actual content. Examples of such benchmarks are various encoding/decoding benchmarks that compress/decompress, encrypt/decrypt. Such benchmarks do such encoding/decoding on streaming characters or small buffers of such characters. The outermost loop receives the input. With a different data set, apart from the trip count of the outer loop, nothing else changes much, thus making no difference to the relative weights between the different variables. Hence, the allocation desired by the non profiled data set is the same as the one for the profiled data set.

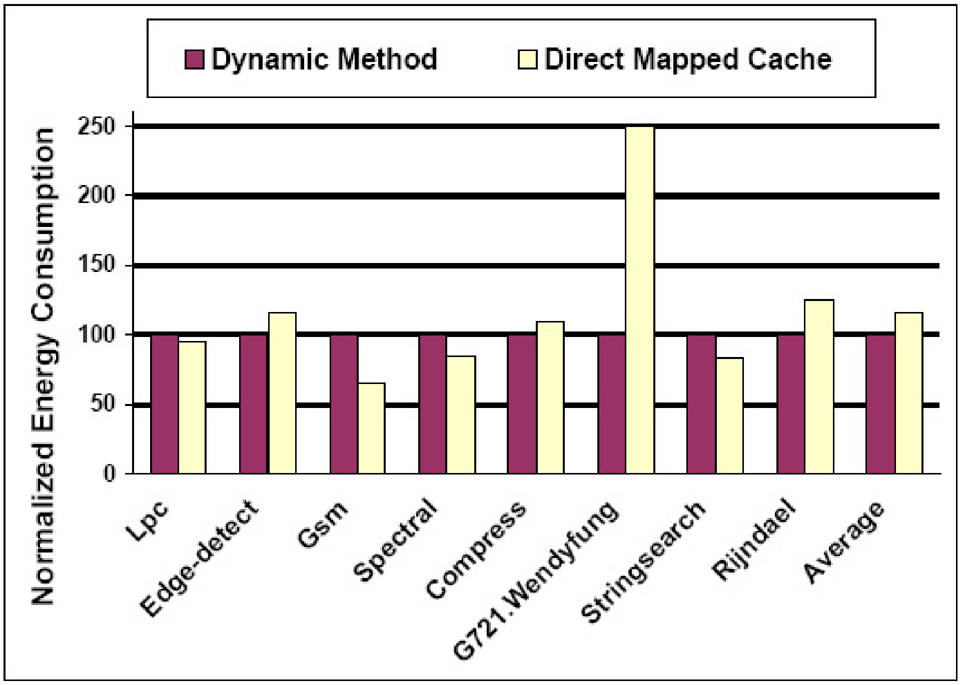


Figure 10.10: Normalized energy consumption for a cache only and scratch-pad memory only architecture measured for maximum benefit configuration

10.2 Comparison with caches

Here we compare our method with an architecture that uses a cache. It is important to note that our method is useful regardless of the results of a comparison with caches because there are a great number of embedded architectures which have a scratch-pad and DRAM directly accessed by the CPU, but have no data cache or I-cache. Examples of such architectures include low-end chips such as the Motorola MPC500, Analog Devices ADSP-21XX, Motorola Coldfire 5206E; mid-grade chips such as the Analog Devices ADSP-21160m, Atmel AT91-C140, ARM 968E-S, Hitachi M32R-32192, Infineon XC166 and high-end chips such as Analog Devices ADSP-TS201S, Hitachi SuperH-SH7050, and Motorola Dragonball. We found at least 80 such embedded processors with no D-cache but with SRAM and external memory (usually DRAM) in our search but have listed only the above eleven. These architectures are popular because scratch-pad is simple to design and verify, and provide better real-time guarantees for global and stack data [89], power consumption, and cost [7, 13, 77, 86] compared to caches.

Nevertheless, it is interesting to see how our method compares against processors containing caches. We choose our desired data capacity as the SRAM size at which the dynamic method obtains maximum benefit compared to the static method. Note that this SRAM size is only an indication of relative gain versus static method. So for purposes of comparing with caches, it is a fairly unbiased choice. To ensure a fair comparison the total silicon area of fast memory (scratch-pad or cache) is equal in both the architectures and roughly equal to the silicon area of the scratch-pad. For our experiments we choose an cache architecture similar to the intel IXP network processor which has an Icache and a Dcache of equal sizes. The goal of our experiment is to compare the performance of cache and SPM of equal silicon area.[1] So the desired data capacity is divided equally between the Icache and Dcache. For a scratch-pad and cache of equal area the cache has lower data capacity because of the area overhead of tags and other control circuitry. Area estimates for cache and scratch-pad are obtained from Cacti [90]. The cache simu-

[1] Actually since cache must be a power of two in size and Cacti has a minimum line size of 8 bytes, the sizes of caches are not infinitely adjustable. To overcome this difficulty we first fix the sizes of the Icache and Dcache to the nearest possible SRAM size. Then a scratch-pad of the same total area is chosen; this is easier since scratch-pad sizes are less constrained.

lated is direct-mapped[2], has a line size of 8 bytes, and is in 0.5 micron technology. On a cache miss, we assume the first word incurs the full DRAM latency of 20 cycles and 1 cycle for each byte thereafter. The scratch-pad is of the same technology but we remove the decoder, tag memory array, tag column multiplexers, tag sense amplifiers and tag output drivers in Cacti that are not needed for the scratch-pad. The Dinero cache simulator [31] is used to obtain run-time results; it is combined with Cacti's energy estimates per access to yield the energy results.

Figure 10.9 shows the normalized run time for cached and non-cached architectures. The first bar represents our method with software transfers. The third bar represents the runtime with cached architecture. As caches have the advantage of hardware mechanisms for fast transfer, we thought it would be interesting to compare when our method also can use some faster transfer mechanism. So the second bar represents our method with faster transfers using hardware like DMA. Our method does better than cached architecture for four of the benchmarks. On the average our method on scratch-pad has 4.1% less runtime when compared to cached architecture. This increases to 5.2% savings with the help of faster transfers. Of course, the improvement in real-time guarantees from the scratch-pad is much larger.

Figure 10.10 shows the normalized energy consumption for cache and our dynamic method with software transfers. For lack of a energy model for DMA we do not show energy consumption for our dynamic method with DMA transfers. On

[2] due to the small sizes involved and Cacti's inability to generate parameters for such sizes, higher associativities were not included

the average our method has 16% less energy consumption compared to a direct mapped cache.

The reason we believe our method does better for four benchmarks is that it can correctly identify the more reused program object and retains it in the scratch-pad whereas cache is likely to evict it even when fetching less-used program object. This is especially significant in case of programs with large loops and loops with procedure calls in them. In such cases a cached architecture might do worse because of large transfers. For Lpc the performance of the dynamic method is very close to the cached architecture. But for rest of the three benchmarks namely Gsm, Spectral and Stringsearch, cached architectures do better than our method. These were benchmarks which had some very large variables which do not fit in the scratch-pad. So while our method cannot fit these variables into the scratch-pad, the cached architecture which only deals with cache lines can bring in data belonging to these variables. This is a general advantage cached architectures would have over any compile time scratch-pad method. In general, to some extent at least for regular programs, this gap can be bridged with the help of more aggressive outlining and advanced array optimization techniques like array blocking [2, 51], structure splitting [78]. Nevertheless, we believe it is remarkable for a compile-time method to do so well compared to a hardware cache.

For real time tasks with hard deadlines, the use of our method offers an improvement in the worst-case runtime. For such tasks, all the improvements of our method translate to improvement in the worst case estimate since the worst-case access time for an SPM is the same as the average case. As compared to this,

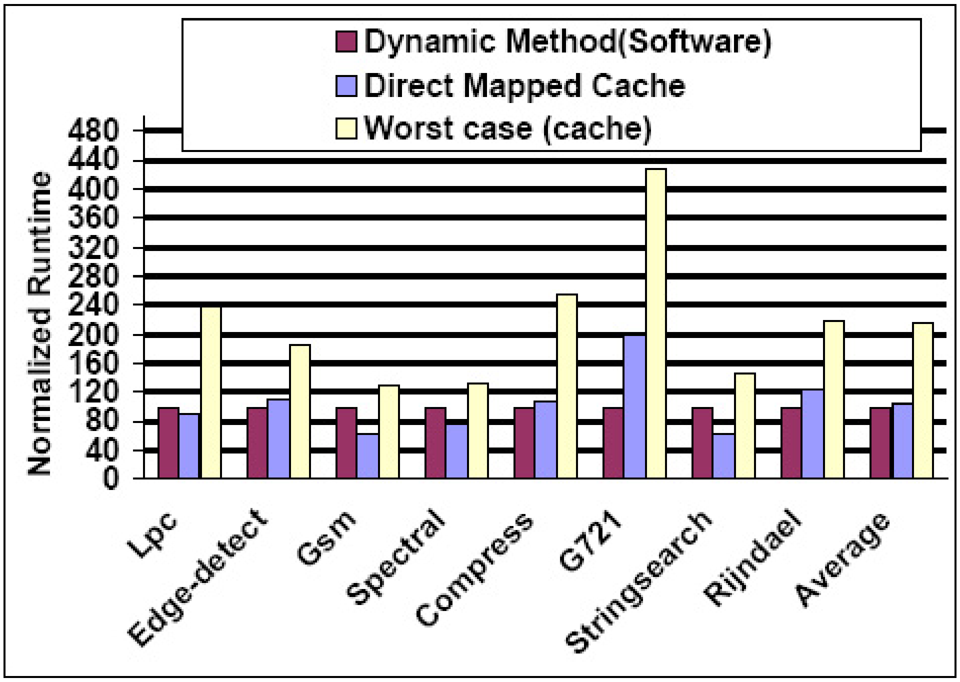


Figure 10.11: Normalized runtime and worst-case runtime for cached architecture

the worst case estimate for a cache has to be conservative regardless of the observed average-case performance, since for the worst case accesses have to be assumed to be cache misses. A practical methodology used by a designer to check if his task meets the worst case scenario is to test his system when the cache is disabled. We use this methodology to approximately quantify our real time gains[3]. In figure 10.11, we show the comparison between the measured and worst case estimated runtimes of a cache. For the worst case, we assume all the memory accesses are cache misses. Two

[3] In this experiment we compare the worst-case run-time with the *one* data set used in our experiments. This is not really the worst-case run-time of the program, since that is measured across *all possible* data sets. Since the true worst-case data set is hard to find (or prove that it is the worst case), we use this admittedly inaccurate, approximate method for worst-case time comparison for the lack of a better method.

possible worst cases are shown with faster transfer rates(12 cycles for a read/write) and normal transfer rates (20 cyles for read/write). For our method with dynamic transfers, the worst-case runtime is same as the measured runtime. In the case of a cache, the ratio between the average worst case and average observed run-time is 2.07:1. Thus, the big difference between the real and the worst cases in the figure shows the importance of using a scratch-pad memory with dynamic method that would offer good performance while ensuring a tighter real time bound.

10.3 Results on dynamic method integrated with partial array handling

This section presents results comparing our method integrated with a partial array pass using affine analysis discussed in chapter 8. We compare against several other allocation alternatives. These include the static allocation method and the basic dynamic method without the partial variable pass. We also consider three variations of an affine-only method that only handles affine loops, flushing the scratch-pad memory after each loop. The objective is to approximately illustrate how some of the other methods cited in chapter 9 can benefit from being a part of a integrated framework. The first variation called "Inflexible-Affine1" performs transfers just before the loop where the affine reference is located. In the second variation called "Inflexible-Affine2" the transfer point is located outside the whole loop-nest. This approach is not likely to be able to move partial arrays for the kind of bench-

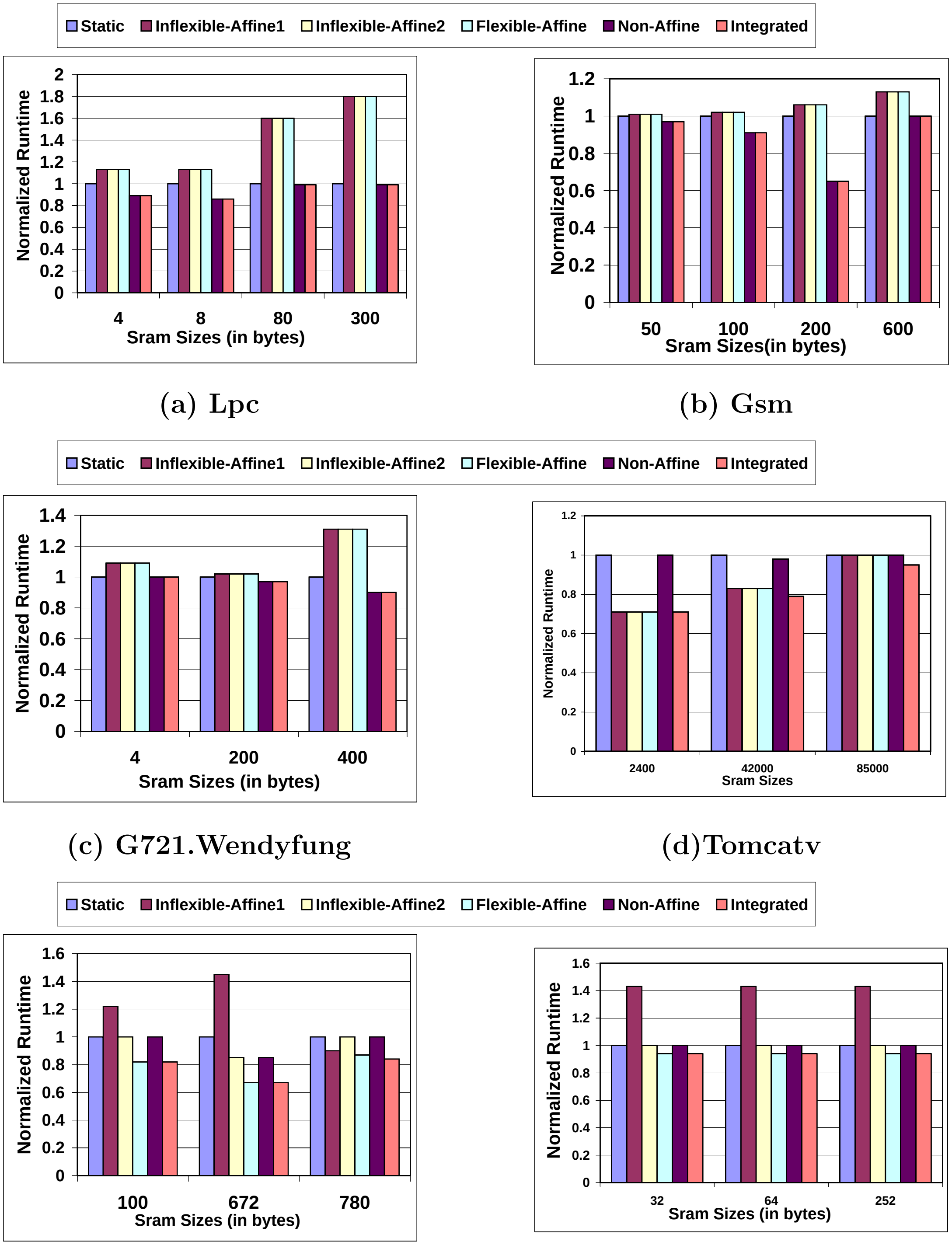
Benchmark	Source	Lines of code	Data size(in KB)
Wss	Perfect club	68	2.5
Compress	UTDSP	309	70.7
Tomcatv	Spec 95	104	28.1
G721.Wendyfung	MiBench	673	2.9
lpc	MiBench	522	7.5
Gsm	MiBench	6035	20.2

Table 10.5: Benchmark programs for our experiments on integrated algorithm.˙

marks that we have chosen, since the loop nest accesses the whole array. The only objective of including it in our experiments here is to illustrate this drawback. Both of these variations employ a simple knapsack allocation that is based on a list of spanned-footprints sorted by their frequency per byte. The only constraint is that a check is made if the cost of transfer can be recovered. The third affine-only method, called the Flexible-Affine method, allows transfers at any of loop levels. The final choice of the transfer point is left to the framework since it uses a sophisticated cost model to find the best point of transfer. The Flexible-Affine method differs from our integrated method in that it empties the scratch-pad memory after each loop nest and moreover, the integrated method can also handle non-affine loops.

The embedded applications evaluated are shown in table 10.5. Only some of the benchmarks from our original table of 10.1 have been selected. This is since the three that we have selected are similar to the rest of the benchmarks in the kind of opportunities they offer for partial-variable allocation.

Figure 10.12 shows the runtime at some key scratch-pad memory sizes using



(a) Lpc

(b) Gsm

(c) G721.Wendyfung

(d)Tomcatv

(e) Wss

(f) Compress

Figure 10.12: Normalized runtime for our integrated method, different affine-only methods, non-affine method and static method .

six different allocation methods. Sizes are chosen to show overall trends. Typically the sizes include the first size when the runtime of the integrated dynamic methods changes significantly (by more than 1%), the final size beyond which the runtime of the integrated method is very close to static allocation and sizes in between this range where any of the dynamic methods perform significantly differently than the other methods. Outside of this scratch-pad memory size range dynamic methods generally perform no better than static methods because for smaller sizes no variable fit, while for larger sizes all variables fit in the scratch-pad memory with no need for dynamic swapping. The figure also shows two important observations in favor of our integrated method.

The first observation is that the integrated method always does as good or better than all the other other methods. In particular, the method does as well or better than both affine and non-affine method. This can be seen in affine benchmarks Tomcatv, Compress and Wss where the integrated method does as well as the Flexible-Affine method while for benchmarks with no affine loops, G721, Gsm and Lpc, the integrated and non-affine methods do better.

It is instructive to see why different benchmarks have different best-allocation strategies. In the case of benchmarks with affine loops, the affine and the integrated method can allocate partial variables to scratch-pad memory whereas neither the static nor the non-affine method can do that. On the other hand, for the same benchmarks but larger scratch-pad memory sizes like for example Tomcatv at size 85,000, the integrated method does better than the affine methods. This happens due to the simple cost model of the affine methods and higher transfer cost arising from the flushing of the scratch-pad memory at the end of the loop nest. The other three non-affine benchmarks, namely Gsm, G721 and Lpc, do not offer any opportunities for partial variable allocation. Hence, the integrated and the non-affine method do equally well and better than the affine alternatives. In conclusion, the best-allocation strategy is a function of both the benchmark characteristic and scratch-pad memory size, with the integrated method uniformly being the best.

The second salient observation is that even among the affine alternatives, flexible transfers makes a difference. This is supported by the observation that flexible-affine either outperforms or equals the other affine alternatives. For two benchmarks Wss and Compress, the flexible-Affine does better than the other affine alternatives. These benchmarks contain triply nested loops and three points where transfer can be done. Transfers outside the outer loop mean that the whole variable has to be placed while transfer just outside the innermost loop causes the transfer cost to be high. Transfers outside the whole loop can exploit partial-variable opportunities only if the complete loop nest accesses only part of the array. The flexible-Affine method on the other hand compares between all the three transfer points and chooses the most beneficial one, which in this case is the middle loop. In case of Tomcatv that contains only doubly nested loops, the runtime of Flexible-Affine is the same as one of the other affine methods.

The results presented here we believe show the importance of integrating partial-variable optimizations with a general framework. The different affine methods we have shown represent important characteristics of some of the existing methods. The better results of our integrated method over affine methods for non-affine benchmarks illustrate how existing methods [5, 23, 58, 67] are limited to affine programs. Further, as discussed in the last paragraph, integration is also important so that the memory transfers can be inserted at the optimal point in the loop nest. Methods similar to the affine alternatives that we have presented such as ones by Eisenbeis [23] et al's method and Schrieber et al [67], do not offer the flexibility to choose the transfer point. However, these methods propose several sophisticated strategies to determine the footprint in a more precise fashion. These are complementary to our scheme and in combination with our approach to integrate such optimizations can yield further benefits.

10.4 Results on pointer handling

In this section, we present some results on the effectiveness of our pointer handling strategies discussed earlier. These results from these experiments also form our basis for the choice of our default pointer handling strategy.

Our experimental methodology involves comparing three different dynamic schemes. The first one is where the dynamic scheme uses address constraining. The second dynamic scheme is where it uses pointer translation implemented in software; we call this software pointer translation. The third also involves pointer translation but at zero overhead; this would represent a perfect but hypothetical dynamic scheme.

An important objective in these experiments is to show the promising applicability of our schemes. Towards this, we base our schemes on a simple address-taken

Benchmark	**Source**	**SPM Size in bytes**	**Additional Code Growth from Translation**
Compress	UTDSP	700	1.4 %
Edge-detect	UTDSP	500	0.0 %
Spectral	UTDSP	800	0.0%
Gsm	MIbench	400	0.2%
Fft	MIbench	300	0.8 %
Dijkstra	MIbench	180	1.2 %

Table 10.6: Benchmarks and Characteristics.

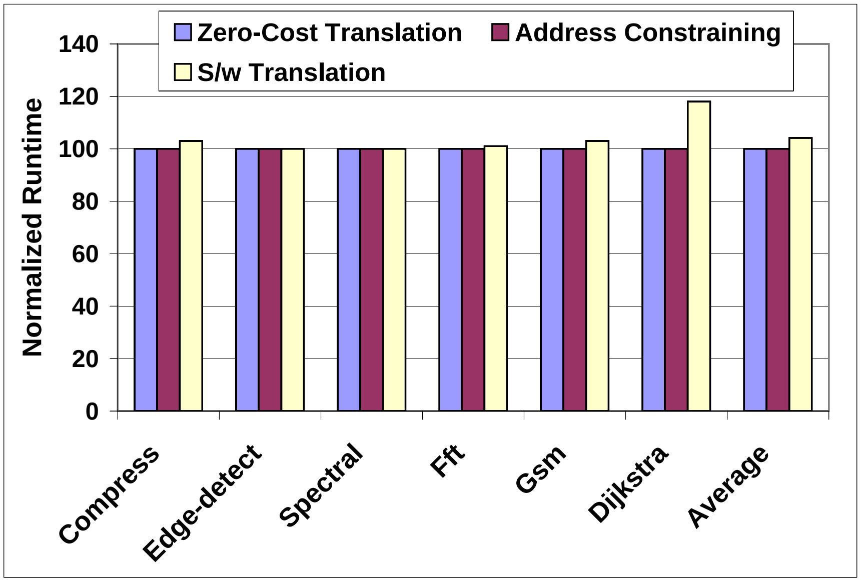


Figure 10.13: Runtime overhead of pointer handling strategies.

pointer analysis strategy. Further, we design our application set and parameters to illustrate various scenarios likely to be seen in real programs. Table 10.6 shows our benchmark set used in these experiments and its source. The benchmarks involve the different kinds of pointer usages such as reference parameters (eg. Compress, Spectral), pointers to global/stack (eg. Compress, Spectral), function pointers (eg. Gsm) and pointers to heap (eg. Fft, Dijkstra). Only four of the benchmarks from our original table of 10.1 have been selected. This is since these the rest are similar to one of these benchmarks in the kind of pointers they use. However, all the results presented in section 10.1 already have these benchmarks using the address-constraining strategy. One other parameter in our experiments is the scratch-pad memory size. The scratch-pad memory size chosen for each benchmark is shown in the table. The choice of our size is based purely on illustrating various scenario's that occur involving pointers and their points-to sets. At the same time, we make sure that the size that we choose is a reasonable size from a designers perspective.

To introduce our results in this section– the overhead of our schemes has two dimensions. Both the schemes can cause some runtime loss. This is illustrated in the figure 10.13. Additionally, pointer translation can cause some code growth. This is in addition to code growth due to memory transfer code needed by dynamic methods. The code growth is shown in the benchmarks figure in column 4. The results show that constraining the layout does as well as the zero-overhead dynamic schemes. On the other hand, the overhead from pointer translation both in code growth and runtime varies from benchmark to benchmark. The runtime overhead ranges from 0 % to 18% and averages 4% over the benchmarks. The code growth is between 0% and 1.4% and averages 0.6%.

The spectrum of results can be understood based on various allocation scenarios involving global and stack variables pointed by pointers. We first try to understand pointer translation. In Edge-detect and Spectral all the variables pointed to by pointers are never allocated to scratch-pad memory, hence the strategies do not cause any overhead. This is important since this means the strategies preserve the runtime totally when not applicable. Benchmarks Compress, Gsm and Fft involve some pointer translation but translation is avoided inside the inner loop and hence does not cause much overhead. The benchmark Gsm involves function pointers and hence causes some additional code growth due to the need for compensation code inside all functions whose address has been taken. Again, this is negligible. The overhead of translation is maximum in the case of Dijkstra. The benchmark involves a dynamic queue traversal and requires a check for heap pointer inside the loop. Further using simple address-taken pointer analysis this pointer cannot be disambiguated as being a heap pointer. This leads to the large runtime loss of 18%.

The results also show that address constraining does very well. This is because these benchmarks do not exercise constraining code at all. This happens due to two main reasons. One, when the variables pointed to by pointers are in DRAM, no constraining is required. Two, even when the variables are in scratch-pad memory and are pointed to by reference pointers, constraining is not required because the variable once swapped out from scratch-pad memory is never brought in again. This we believe stems from several programming practices that have evolved for coding a procedure. Generally procedures are kept short and so sometimes involve just one region. Also even when they are long they process one aggregate variable at a time and the output is passed on to the procedure's next part. So inside a function variables are required to be in the scratch-pad memory only for a few consecutive regions, eliminating the need for constraining.

Chapter 11

Conclusion and Future work

This thesis presents compiler-driven memory allocation scheme for embedded systems that have SRAM organized as a scratch-pad memory instead of a hardware cache. Most existing schemes for scratch-pad rely on static data assignments that never change at runtime, and thus fail to follow changing working sets; or use software caching schemes which follow changing working sets but have high overheads in runtime, code size memory consumption and real-time guarantees. We present a scheme that follows changing working sets by moving data from scratch-pad to DRAM, but under compiler control, unlike in a software cache, where the data movement is not predictable. Predictable movement implies that with our method the location of each variable is known to the compiler at each point in the program, and hence the translation code before each load/store needed by software caching is not needed. The benefit of our method depend on the scratch size used. When compared to an existing static allocation scheme, results show that our scheme reduces runtime by up to 39.8% and overall energy consumption by up to 31.3% on average for our benchmarks, depending on the scratch-pad size used.

We identify two directions of future work. We outline these directions below.

Optimizations One direction for further research is developing loop transformations and data transformations that would benefit scratch-pad memory allocators. In this thesis we have addressed one issue associated with some of these optimizations namely integration of such optimizations with an existing allocation framework. Another issue in the developement of such optimizations is the question of cost model. Such optimizations currently are based on cost-models that are cache specific. Adapting them for optimizing scratch-pad allocation would require adapting the cost-model as well. One alternative to a precise cost-model is to generate several versions of the output of the optimization. An example is to generate multiple partial variables from a structure variable, composed of different fields in the structure. The exact choice of partial variable is left to the algorithm. In the extension that we described, the framework is already extended be able to utilize different partial variables belonging to the same variable. A similar approach can be adopted in the case of loop transformations; multiple versions of the same loop that are transformed accroding to different heuristics can be generated. To handle such cases, the framework would need additional extensions.

Memory allocation for multitasking environment Another direction for future research can be to extend the scheme to multithreading environments. Many such systems use preemptive multitasking, especially those with an underlying real-time operating system (RTOS). Priorities are assigned to tasks, and the RTOS always executes the ready task with highest priority. We believe the key aspect of the problem would be the trade of between predictability of memory accesses and utilization of scratch-pad memory. If total predictability is still desired, then statically partitioning the scratch-pad memory among the different task according to their priority and then using our method for each task separately might be the only solution. If instead utilization were more important, slightly dynamic extensions to our scheme might be required.

BIBLIOGRAPHY

[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3), August 2002. http://developer.intel.com/technology/itj/2002/volume06issue03/.

[2] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 31. IEEE Computer Society, 2000.

[3] Randy Allen and Ken kennedy. *Optimizing Compilers for Modern Architecture.* Morgan Kaufmann Publishers Inc, 2001.

[4] *ADSP-21xx 16-bit DSP Family.* Analog Devices, 1996. http://-www.analog.com/processors/processors/ADSP/index.html.

[5] S. Anantharaman and S. Pande. Compiler optimization for real time execution of loops on limited memory embedded systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, 1998.

[6] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 2003*

international conference on Compilers, architectures and synthesis for embedded systems, pages 318–326. ACM Press, 2003.

[7] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267. ACM Press, 2004.

[8] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, January 1998.

[9] *ARM968E-S 32-bit Embedded Core*. Arm, Revised March 2004. http://-www.arm.com/products/CPUs/ARM968E-S.html.

[10] Oren Avissar. Heterogeneous Memory Management for Embedded Systems. Master's thesis, University of Maryland, College Park, 2002.

[11] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proceedings of the ACM 2nd International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, November 2001. Also at http://www.ece.umd.edu/∼barua.

[12] Oren Avissar, Rajeev Barua, and Dave Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Systems (TECS)*, 1(1), September 2002.

[13] R. Banakar, S. Steinke, B-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded

Systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, Colorado, May 6-8 2002. ACM.

[14] Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, Tiebing Zhang, and Bruce Jacob. The performance and energy consumption of three embedded real-time operating systems. In *Proc. Fourth Workshop on Compiler and Architecture Support for Embedded Systems (CASES'01)*, pages 203–210, Atlanta GA, November 2001.

[15] Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, Tiebing Zhang, and Bruce Jacob. The performance and energy consumption of embedded real-time operating systems. *IEEE Transactions on Computers*, 52(11):1454–1469, November 2003.

[16] L.A. Belady. A study of replacement algorithms for virtual storage. In *IBM Systems Journal*, pages 5:78–101, 1966.

[17] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, 1990.

[18] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. 24(7):258–263, July 1989.

[19] Azer Bestavros, Robert L. Carter, Mark E. Crovella, Carlos R. Cunha, Abddsalam Beddaya, and Sulaiman A.Mirdad. Application-level document

caching in the internet. In *Proceedings of the Second Intl. Workshop on Services in Distributed and Networked Environments (SDNE)'95*, pages 125–135, 1990.

[20] David Brash. *The ARM architecture Version 6 (ARMv6).* ARM Ltd., January 2002. White Paper.

[21] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. 24(7):275–284, July 1989.

[22] R. A. Bringmann. *Compiler-Controlled Speculation.* PhD thesis, University of Illinois, Urbana, IL, Department of Computer Science, 1995.

[23] D.Windheiser C. Eisenbeis, W. Jalby and C.B. Fran. A strategy for array management in local memory. In *Technical Report 1262, INRIA, Domaine de Voluceau, France*, 1990.

[24] *Cacti 3.2.* P. Shivaumar and N.P. Jouppi, Revised 2004. http://-research.compaq.com/wrl/people/jouppi/CACTI.html.

[25] D Callahan and K Kennedy. Analysis of interprocedural side-effects in a parallel programming environment. *In Journal of Parallel Distributed Computing*, 1988.

[26] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. *Journal of Parallel and Distributed Computing*, 38(2):248–255, 1996.

[27] G. J. Chaitin. Register allocation and spilling via graph coloring. 17(6):98–105, June 1982.

[28] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. Application-Specific Memory Management in Embedded Systems Using Software-Controlled Caches. In *Proceedings of the 37th Design Automation Conference*, June 2000.

[29] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. 19(6):222–232, June 1984.

[30] Intel Fortran compiler. *http://www.intel.com/cd/software/products/asmo-na/eng/compilers/index.htm.*

[31] *DineroIV Cache simulator.* J. Edler and M.D. Hill, Revised 2004. http://www.cs.wisc.edu/ markhill/DineroIV/.

[32] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. *In Journal of Embedded Computing(JEC), Issue 4*, 2005. IOS Press, Amsterdam, Netherlands.

[33] Poletti Francesco, Paul Marchal, David Atienza, Francky Catthoor Luca Benini, and Jose M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *In Proceedings of the Design Automation Conference*, pages 238–243. ACM Press, June,2004.

[34] D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. In *Software-Practice and Experience*, pages 929–965, 1996.

[35] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. 24(7):264–274, July 1989.

[36] G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proc. of the 27th Int'l Symp. on Computer Architecture (ISCA)*, Vancouver, British Columbia, Canada, June 2000.

[37] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *In IEEE Transactions on Parallel and Distributed Systems*, July 1991.

[38] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, page 113, 2000.

[39] Jason D. Hiser and Jack W. Davidson. Embarc: an efficient memory bank assignment algorithm for retargetable compilers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 182–191. ACM Press, 2004.

[40] C. Huneycutt and K. Mackenzie. Software caching using dynamic binary rewriting for embedded devices. In *Proceedings of the International Conference on Parallel Processing*, pages 621–630, 2002.

[41] *The PowerPC 405 Embedded Processor Family.* IBM Inc. Microelectronics, 2002. http://www-306.ibm.com/chips/products/powerpc/processors/.

[42] *The PowerPC 440 Embedded Processor Family.* IBM Inc. Microelectronics, 2002. http://www-306.ibm.com/chips/products/powerpc/processors/.

[43] Arun Iyengar. Design and performance of a general-purpose software cache. *Journal of Parallel and Distributed Computing*, 38(2):248–255, 1996.

[44] Jeff Janzen. Calculating Memory System Power for DDR SDRAM. In *DesignLine Journal*, volume 10(2). Micron Technology Inc., 2001. http://www.micron.com/publications/designline.html.

[45] D. Keitel-Sculz and N. Wehn. Embedded dram development technology, physical design, and applicationissues. In *IEEE Design and Test of Computers*, June 2001.

[46] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320, Portland, Ore., 1993. Berlin: Springer Verlag.

[47] David B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the IEEE Symposium on Real-Time Systems*, pages 229–237, December 1989.

[48] Eric Larson and Todd Austin. Compiler controlled value prediction using branch predictor based confidence. In *Proceedings of the 33th Annual International Symposium on Microarchitecture (MICRO-33).* IEEE Computer Society, December 2000.

[49] Compilation Challenges for Network Processors. *Industrial Panel, ACM Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, June 2003. Slides at http://www.cs.purdue.edu/s3/LCTES03/.

[50] Zhiyuan Li and Pen-Chung Yew. Efficient interprocedural analysis for program parallelization and restructuring. *In ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, June 1988.

[51] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112. ACM Press, 2001.

[52] L.Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *International conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS)*. ACM, 2004.

[53] Chi-Keung Luk and Todd C. Mowry. Cooperative instruction prefetching in modern processors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 182–194, November 30-December 2 1998.

[54] Burke M and Cytron R. Interprocedural Dependence Analysis and Parallelization. *In SIGPLAN Symposium on Compiler Construction*, July 1986.

[55] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[56] *128Mb DDR SDRAM data sheet.* (Dual data-rate synchronous DRAM) Micron Technology Inc., 2003. http://www.micron.com/products/dram/ddrsdram/.

[57] *128Mb Q-Flash memory.* Micron technology Inc. http://www.micron.com/products/nor/qflash/partlist.aspx.

[58] M.Kandemir, J.Ramanujam, M.J.Irwin, N.Vijaykrishnan, I.Kadayif, and A.Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Design Automation Conference*, pages 690–695, 2001.

[59] Csaba Andras Moritz, Matthew Frank, and Saman Amarasinghe. FlexCache: A Framework for Flexible Compiler Generated Data Caching. In *The 2nd Workshop on Intelligent Memory Systems*, Boston, MA, November 12 2000.

[60] *CPU12 Reference Manual.* Motorola Corporation, 2000. (A 16-bit processor). http://e-www.motorola.com/brdata/PDFDB/MICROCONTROLLERS/-16_BIT/68HC12_FAMILY/REF_MAT/CPU12RM.pdf.

[61] *M-CORE - MMC2001 Reference Manual.* Motorola Corporation, 1998. (A 32-bit processor). http://www.motorola.com/SPS/MCORE/-info_documentation.htm.

[62] *MPC500 32-bit MCU Family.* Motorola/Freescale, Revised July 2002. http://-www.freescale.com/files/microcontrollers/doc/fact_sheet/MPC500FACT.pdf.

[63] Frank Mueller. Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 125–133. ACM Press, 1995.

[64] K. Palem, R. Rabbah, V. Pinar, and K. Kiran. Design space optimization of embedded memory systems via data remapping, 2002.

[65] P. R. 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*, 5(3), July 2000.

[66] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[67] D. C.Cronquist R. Schreiber. Near-optimal allocation of local memory arrays. In *HPL-2004-24*, 2004.

[68] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R.Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[69] R. Sethi. Complete register allocation problems. In *SIAM J. of Computing,*.

[70] S. M. Shahrier and J. C. Liu. On the Design of Multiprogrammed Caches for Hard Real-Time systems. In *Proceedings of the IEEE International Perfor-*

mance, Computing and Communications Conference (IPCCC'97), pages 17–25, February 1997.

[71] Amit Sinha and Anantha Chandrakasan. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Design Automation Conference*, pages 220–225, 2001.

[72] Jan Sjodin, Bo Froderberg, and Thomas Lindgren. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*, December 1998.

[73] Jan Sjodin and Carl Von Platen. Storage Allocation for Embedded Processors. *Compiler and Architecture Support for Embedded Computing Systems*, November 2001.

[74] Monica S.Lam and Michael E.Wolf. A data locality optimizing algorithm. In *in Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[75] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, St. Petersburg Beach, FL, January 1996.

[76] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*. ACM, 2002.

[77] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, page 409. IEEE Computer Society, 2002.

[78] B. Davidson T. M. Chilimbi and J. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, pages 13–24. ACM, 1999.

[79] Yudong Tan and Vincent Mooney. A Prioritized Cache for Multi-tasking Real-Time Systems. In *Proceedings of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 168–175, April 2003.

[80] Andrew S. Tanenbaum. *Structured Computer Organization (4th Edition)*. Prentice Hall, October 1998.

[81] *TMS370Cx7x 8-bit microcontroller.* Texas Instruments, Revised Feb. 1997. http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf.

[82] V. Tiwari and M. T.-C. Lee. Power Analysis of a 32-bit embedded microcontroller. *VLSI Design Journal*, 7(3), 1998.

[83] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems (CASES)*, pages 276–286. ACM Press, 2003.

[84] Sumesh Udayakumaran, Bhagi Narahari, and Rahul Simha. Application specific memory partitioning for low power. In *Proceedings of ACM COLP 2002 (Compiler and Operating Systems for Low Power*. ACM Press, 2002.

[85] Osman S. Unsal, Rakshit Ashok, Israel Koren, C. Manik Krishna, and Csaba Andras Moritz. Cool-cache for hot multimedia. In *Proceedings of the International Symposium on Microarchitecture*, pages 274–283, 1990.

[86] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the conference on Design, Automation and Test in Europe*. IEEE Computer Society, 2004.

[87] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2005.

[88] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI2004)*, 2004.

[89] L. Wehmeyer and P. Marwedel. Influence of onchip scratchpad memories on wcet prediction. In *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2004.

[90] S.J.E. Wilton and N.P. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*, 1996.