Improving Code Overlay Performance by Pre-fetching in Scratch Pad Memory Systems

by

Nikhil Ghadge

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved March 2011 by the
Graduate Supervisory Committee:

Karamvir Chatha, Chair
Aviral Shrivastava
Yann-Hang Lee

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

Advances in electronics technology and innovative manufacturing processes have driven the semiconductor industry towards extensive miniaturization & ever greater integration of chip design. One consequence of this sustained evolution has been the growing relative cost of accessing off-chip components with external memory being one of the dominant contributors. In embedded systems and applications, where power consumption and cost are extremely crucial factors, the use of on chip Scratch Pad Memories (SPMs) has proven to be a good alternative to caches. SPMs are more efficient than on-chip caches in a wide variety of aspects including energy consumption, power dissipation, speed performance, area, and timing predictability. However, at the same time, they entail explicit software-level management. Specifically, the system performance depends upon overlay scheme for mapping code and data onto the size-limited SPMs. It has been found that for applications with large code sizes, the overlay overhead cost becomes significant. This work aims to evaluate and implement pre-fetching as a performance improvement technique for SPMs. It is implemented in code overlay manager, provided with the Cell Broadband Engine (CBE) Synergistic Processing Unit (SPU) compiler from IBM, *spu-gcc*. Four different approaches proposed in this work use profiling information to predict pre-fetch calls. The pre-fetching technique achieves considerable performance improvement by hiding some of the code overlay cost behind active computations by fetching the required code segment in advance into SPM. Experimental results supporting this claim are obtained using the IBM Cell architecture platform with substantial gain of more than 30%.

To my parents ...

... for their unwavering support and encouragement

and for giving me the courage to soar ...

To my sister ...

... for being my strength through the entire journey so far...

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor, Dr Karamvir Chatha for his direction and kind encouragement throughout the course of my thesis. His patience and tolerance towards my endless questions is really appreciated. His keen interest and support were pivotal to the successful completion of my research.

My sincere thanks are due to my committee members Dr. Yann-Hang Lee and Dr. Aviral Shrivastava for their constant interaction and mentoring.

I owe my gratitude to my mentor, Mike Baker, who is my inspiration, for all the fruitful discussions and suggestions during the various stages of the work carried out. This thesis would not have been possible without his guidance.

I am equally grateful to Amrit Panda for his constant assistance and motivation through the course of the project. I am also extremely grateful to Weijia Che for enlightening me on the conceptual aspects and nuances of Cell IBM architecture and code overlay, which greatly simplified my thought process.

I am extremely thankful to all the reviewers of this document, for making it what it is.

I am also grateful to the faculty members and administrative staff for their unstinted support throughout the duration of my master's degree, here at Arizona State University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

INTRODUCTION

1.1 Overview

The first generation of embedded systems were limited to fixed, relatively simple, single functionality devices like digital watches, calculators, coffee makers etc. Contemporary embedded systems have, however, evolved over a period of time into highly complex, programmable, intelligent, multi-functionality devices which include PDAs, GPS, cellular phones, and music players. Such systems have to satisfy stringent power, performance, and cost constraints. Many of these devices are on portable platforms and hence operate on battery, making power consumption a crucial design factor. For such mobile embedded devices, reduced energy consumption translates to increased battery life or reduced product dimensions, weight and cost or both. Consequently, the overall competitiveness of the product is improved. It has been observed that memory subsystems contribute to the consumption of 50% to 75% of the power budget of the entire system[14, 25].

Modern embedded systems improve their performance by using memory hierarchies consisting of caches or scratchpads or both. The memory subsystem, especially on-chip caches using SRAM, consume a large portion of the total chip power. The advantage of caches is the easy integration with the software of the system. The detection of a cache hit or miss is done in hardware. If the accessed data is currently not available in the cache, the hardware control automatically copies the data into the cache. This mechanism allows the use of software without any adaption to the changed memory hierarchy. A disadvantage of caches occurs in realtime embedded systems where a certain response time has to be guaranteed. For the Worst Case Execution Time (WCET), a cache miss has to be assumed [24].

Recently, Scratch Pad Memory (SPM) has been proposed as an alternative to caches in order to reduce power and improve performance. While cache behavior is unpredictable to programmers and transparent to applications, programmers can explicitly map the addresses of external memory to the addresses of the SPM because the SPM is addressed using an independent address space.

A SPM is a fast on-chip SRAM managed by software. Compared to hardware-managed caches, SPMs offer a number of advantages. First, SPMs are more energy efficient and cost-effective than caches since they do not need complex tag-decoding logic. Second, in embedded applications with regular data access patterns, an SPM can outperform a cache memory since software can better choreograph the data movements between the SPM and off-chip memory. Finally, such a software managed data movement guarantees better timing predictability, which is critical in hard real-time embedded systems. Given these advantages, SPMs are increasingly used as an alternative to caches in modern embedded processors, such as the Motorola M-core MMC221, the TI TMS370Cx7x, and the IBM Cell processor. In other embedded processors such as ARM10E and ColdFire MCF5, both caches and SPMs are incorporated in order to obtain the best of both memory architectures.

1.2 Statement of the Problem

For SPM-based systems, the programmer or compiler must schedule explicit data transfers between the SPM and off-chip memory. The efficacy of SPM management critically dictates the performance and energy cost of an application. Presently, this task is largely accomplished manually. The programmer often spends a lot of time on partitioning data and inserting explicit data transfers between the SPM and off-chip memory. Such a manual approach is time-consuming and error-prone. In addition, data aggregates such as arrays in large programs often exhibit cross-function data reuse. Obtaining satisfactory solutions for large applications by hand can be challenging. Finally, hand-crafted code is not portable since it is usually customized for one particular architecture.

To overcome these limitations, researchers have investigated a number of compiler strategies for allocating data to a SPM automatically. SPM management techniques using Memory Management Unit (MMU), cache aware techniques, code overlay schemes, and many more have been extensively researched. All these techniques involve better management of available memory in the embedded system based on the application which runs on it.

Although, the key problem of SPM management is addressd by other techniques, the cost of management scheme can not be overlooked when it comes to large code sizes. Overhead of using management scheme for large code bases is considerably high. This work addresses this issue by reducing the overhead cost of one of the management scheme.

1.3 Contribution

This work optimizes the code overlay scheme to get performance improvement for multi-core embedded architectures. The proposed technique is pre-fetching where the required set of code segment is brought to the SPM via Direct Memory Access (DMA) in advance by the overlay manager. This technique is implemented in *spu-gcc* to reduce the code overlay overhead cost. In order to evaluate the pre-fetch technique, this work proposes four different approaches to predict code pre-fetch calls within large code bases. Out of four, three approaches are fully automated such that they predict the pre-fetch calls, insert them into the code, and execute them to check the reduced code overlay overhead. Only one prediction technique involves human intervention for making pre-fetch calls prediction. This work also evaluates the performance of the proposed stratergies on large code bases when executing on commercial processors.

1.4 Organization

Chapter 2 describes the background study and literature. Chapter 3 discusses related work. SPU code overlay is explained in Chapter 4. Actual implemetation is presented in Chapter 5, which explains different approaches followed by experimentation results of pre-fetching compared to the normal execution without pre-fetching in Chapter 6. Chapter 7 summarizes the research work.

Chapter 2

BACKGROUND

2.1 Overview

Code mapping for SPM can be broadly classified into static and dynamic techniques, as shown in Figure 2.1. Since it is managed by software, placing the correct code/data segments in SPM requires a meticulous and careful analysis of the memory access patterns of the application. Allocating frequently accessed code or data to the SPM may benefit both performance and energy consumption. SPM mapping can be done in either of the following ways:

Static: Static allocation techniques select a fixed set of segments to reside in the SPM during the entire execution of the application. In this case, SPM is loaded once during program initialization and this content does not get altered further during program execution.

Dynamic: Dynamic allocation schemes modify the contents of the SPM during execution. These techniques replenish the contents of the SPM with different code segments during program execution by overlaying multiple code segments.

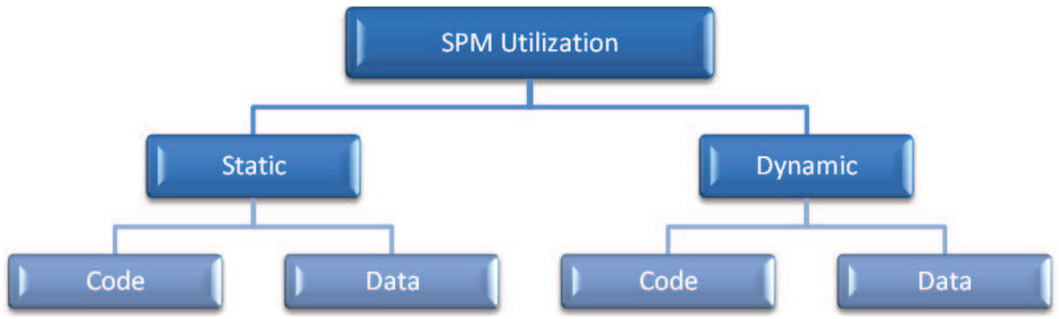


Figure 2.1: SPM utilization techniques

These traditional methods for SPM utilization break down the SPM mapping problem into two small challenges [18]:

(i) “what to map”: This involves partitioning the application code into SPM and main memory. This segregation helps eliminate segments whose cost of transfer from memory to SPM is greater than that of executing from SPM. This memory management problem is insignificant for architectures having DMA controller. With DMA, it is always beneficial to execute code from the SPM rather than main memory.

(ii) "where to map" : This challenge, termed as address assignment, involves determining the addresses on the SPM where the code will be mapped. Appropriate address assignment becomes essential for efficient utilization of available memory resource.

Static techniques do not always address issue `(i)` , but they do solve issue `(ii)`. The most efficient SPM management results from dynamic techniques. This comprises of partitioning the SPM into different regions and mapping multiple code segments with non-overlapping live ranges to these regions. Thus a dynamic technique for code mapping can be broken down into the following two steps:

(I) Partitioning of SPM into a number of regions

(II) Overlaying the code objects/segments onto these regions

2.2 Introduction to SPM

SPM, also known as scratchpad, scratchpad RAM or local store in computer terminology, is a high-speed internal memory used for temporary storage of calculations, data, and other intermediate work. With reference to a microprocessor ("CPU"), scratchpad refers to a special high-speed memory resource used to hold chunks of data for rapid retrieval.

SPM is comparable to the *L1* cache in the system because it is the next closest memory to the processor, after the internal registers. It often uses DMA-based data transfer by explicitly issuing instructions to store and load data to and from the main memory. Although analogous to cache, they are not equivalent. In direct contrast with a system that uses caches, a system with scratchpads has Non-Uniform Memory Access (NUMA) latencies. This is because the memory access latencies to the scratchpads and main memory vary. Another important difference between the two systems is that a scratchpad normally does not contain a copy of the data that is also stored in the main memory.

Scratchpads are typically employed for simplification of caching logic. In the context of Multiprocessor System-on-Chip (MPSoC) embedded systems, they are used to guarantee that a unit can function without main memory contention, a phenomenon that is common for the multi-processor environment. They often act as a CPU stack storing temporary results, which mostly does not need committing to main memory. In case of systems with relatively slower main memory, they can also be utilized as a replacement for cache in order to mirror the state of the main memory. Similar to cache, SPM also raises the issue of locality of reference when it comes to its efficient use, although some systems do allow strided DMA to access rectangular data sets. Another major difference is that scratchpads are explicitly manipulated by applications, which is quite different to the cache function.

Scratchpads are not used in mainstream desktop processors. A major reason for this is the generality requirement necessary for legacy software to run from generation to generation, in which case the available on-chip memory size may change. They are better implemented in embedded systems, special-purpose processors and gaming consoles. In these cases, the chips are generally manufactured as MPSoC, and the software is often tuned to a single hardware configuration. e.g. Sony PlayStation 3 (PS3), XBOX 360.

Code Overlay

Code overlay is a technique for mapping instruction code onto available memory in which the code would not otherwise fit. In designing an overlay mapping scheme, available memory is partitioned into one or more fixed regions to which one or more code segments may be assigned. Every segment assigned to a given region is mapped to the same address in the SPM. The size of a region in memory is precisely that of its largest mapped segment. Code may be stacked into segments so as to ensure that the end address of the first element is followed by the start address of the second code element and so on.

Normally instructions may be assigned to segments at object file, function, or basic block resolution. In addition, any number of elements may be assigned to one segment. When the code overlay scheme is implemented and the code is executed, each region of memory always contains exactly one code segment at any given time during execution. When instructions are requested which are not currently present in the SPM, the appropriate segment is loaded from the main memory into its assigned region by overwriting the current segment. This is referred to as an overlay miss. Each miss has an associated overhead which is defined as the amount of time expended to load the missing segment. This overhead is proportional to the size of the code segment. The cumulative overhead in terms of time associated with execution of a program using a specific overlay mapping is the mapping's total cost. Chapter 4 talks in detail about SPU code overlay to give a more comprehensive idea.

2.3 Sony IBM Cell

The Cell architecture grew from a challenge posed by Sony Computer Entertainment, Toshiba, and IBM, an alliance commonly know as *"STI"*, to provide a cost-effective, power-efficient, and high-performance processing unit for a wide range of applications, including the most demanding consumer appliance - gaming consoles. Cell is shorthand for Cell Broadband Engine Architecture, often referred to as Cell BE. This architecture is a state-of-the-art design, which is based on the analysis of a broad range of workloads, in a variety of application areas such as cryptography, graphics transforms, image processing, video encoding/decoding, computational physics, Fast-Fourier Transforms (FFT), matrix operations, and scientific workloads. Cell combines a general-purpose Power PC Architecture core having modest performance with streamlined co-processing elements, which greatly accelerate multimedia, vector processing functions, and other dedicated computation-intensive tasks.

Cell Architecture

Cell is a heterogeneous, multiprocessor chip [1] that consists of an IBM 64-bit Power Architecture^{TM} core, often called as the Power Processing Unit (PPU). This PPU core, which acts as a master, is supplemented by eight specialized co-processors based on a novel Single-Instruction Multiple-Data (SIMD) architecture called Synergistic Processor

Unit (SPU). The SPU architecture is used for data-intensive processing, like that found in cryptography, media, and scientific applications. The entire system is integrated by a coherent on-chip bus. It actually bridges the gap between conventional desktop processors (such as the Athlon 64, and Core 2 families) and more specialized high-performance processors, such as NVIDIA and ATI graphics processors.

The performance of single core processors steadily improves as the frequency is increased. They also fulfill the current industry trends of small die area, extensive integration of large function sets, and cost-effectiveness. However, this increase in frequency and decreasing die area has raised the issue of increased power dissipation. This is because the power densities across the chip escalate due to increased technology scaling and switching activity. Consequently, the single core processor has hit the *power wall*, where frequency can no longer be increased without additional power handling costs. As a result, the best approach to achieving high performance and reduced power targets is to exploit parallelism through a large number of cores working in tandem on a single chip. A heterogeneous configuration with a SIMD-centered architecture further achieves reduction in power. This configuration combines the flexibility of the PPU with the functionality and performance optimizations of the SPU SIMD cores.

Synergistic Processing Elements (SPE) is comprised of SPU along with Memory Flow Controller (MFC) and some small memory. Each SPU consists of a 256 KB embedded SRAM for instruction and data, called as "Local Storage" (LS). This LS is referred to as SPM and it is the only memory directly addressable by the SPU.
The SPU architecture has the following characteristics:

- provides a large register file
- simplifies code generation
- reduces the system size and power consumption by unifying resources, and
- simplifies decode and dispatch.

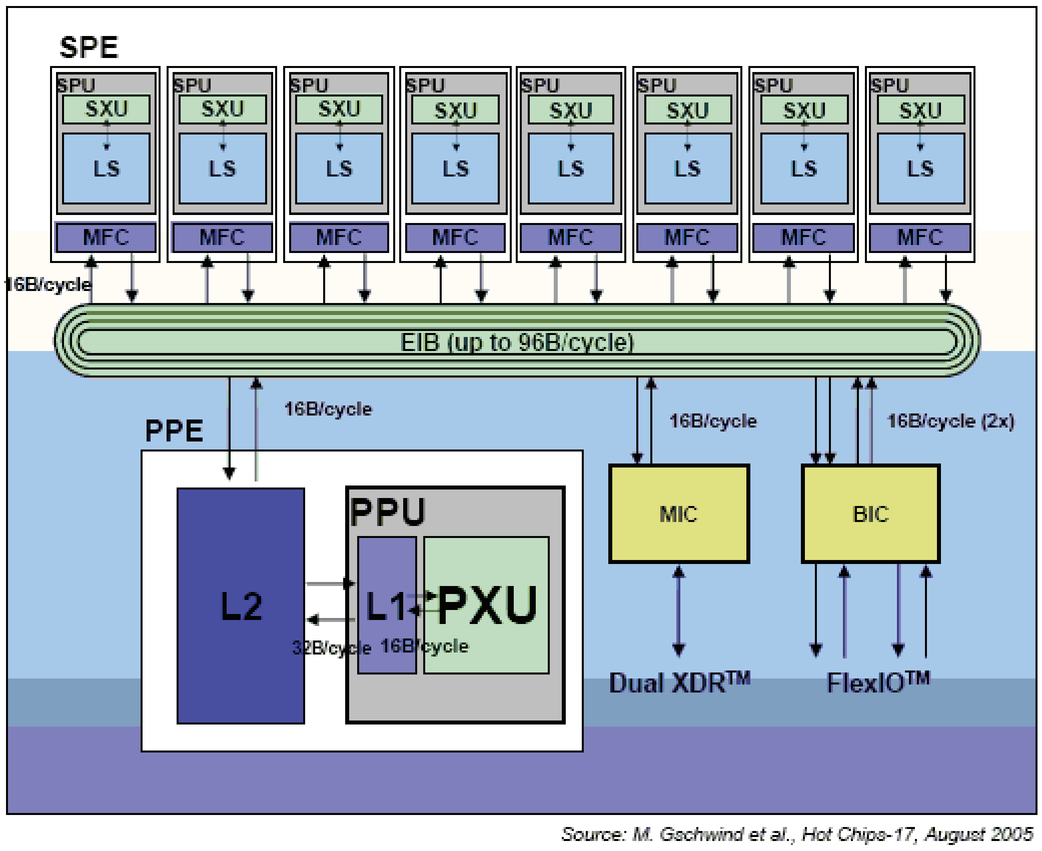


Figure 2.2: IBM CELL Architecture

While the SPU Instruction Set Architecture (ISA) is a novel architecture, the SPU operations are closely aligned with the functionality of the Power^{TM} VMX unit. This form of closely-associated functionality enables code portability between the PPU processor and the SPU SIMD cores. However, for most computation formats, the supported data type range has been reduced. VMX supports a number of densely packed saturating integer data types. However, such data types lead to loss of dynamic range, which produces degraded computation results. Hence, it is preferable to widen the integer data types and perform the intermediate computations without saturation. Memory bandwidth and footprint may be reduced and at the same time, high data integrity can be achieved by performing unpack and saturating pack operations.

Floating point data types do not require additional data conditioning since they, by default, support a wide dynamic data range and saturation. So as to ensure that the area and power constraints are met, floating point arithmetic is limited to the most basic modes in the SPU. In addition, a single rounding mode is supported. Single precision floating point is supported so as to achieve improved performance for multilmedia and three-dimensional graphics rendering applications. Real time gaming applications work extremely well with this architecture.

The PPU has the capability to run conventional operating system on it. It controls all 8 SPUs and can start, stop, interrupt, and schedule processes running on each of them. In order to control SPUs, the PPU has been designed to support additional instructions. Unlike SPUs, the PPU can read from and write to the main memory and the local stores of the SPEs through the standard load/store instructions. Despite having complete architectures, the SPUs are not fully autonomous and require the PPU to prime them before they can do any useful work. Although most of the computational work is done by the SPEs, the use of DMA as a data transfer method and the limited local memory footprint of 256KB in each SPU pose a major challenge to software developers who wish to make the most of this. This scenario necessitates meticulous hand-tuning of programs to extract maximum performance from the SPU. To achieve the high performance necessary for computation-intensive tasks such as decoding/encoding MPEG streams, generating or transforming three-dimensional data, or undertaking Fourier analysis of data, software developers are required to come up with the best overlay schemes that do justice to this heterogenous architecture.

The Element Interconnect Bus (EIB) is a communication bus internal to the Cell processor, which connects the various on-chip system components - PPU processor, Memory and I/O Controller (MIC), eight SPU co-processors, and two off-chip I/O interfaces. The EIB also includes an arbitration unit that actively determines and mediates which unit has current bus control. The data transfer between a SPU and PPU or between two SPUs is performed over EIB via the cache coherent DMA mechanism. The PPU and bus architecture includes various modes of operation giving different levels of memory protection, allowing areas of memory to be protected from access by specific processes running on the

SPUs or the PPU.

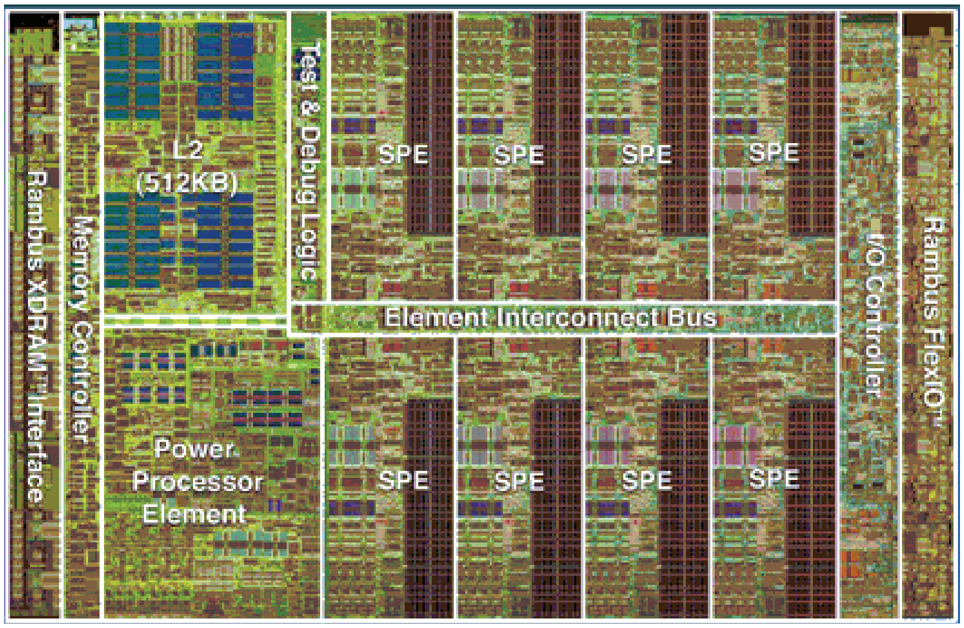


Figure 2.3: IBM CELL die

The SPU is an in-order, dual-issue, statically scheduled architecture. It can issue two SIMD instructions per cycle - one compute instruction and one memory operation. Unlike x86 architecture, the SPU architecture does not include dynamic branch prediction. Instead, it relies on compiler-generated branch prediction using *"prepare-to-branch"* instructions to redirect instruction prefetch to branch targets. The SPU was originally designed with compiled code being the main focus. Early availability of SIMD-optimized compilers also allowed development of high-performance graphics and media libraries entirely in the C programming language. The subsequent advancement in SPU compiler technology led to the development of an advanced parallelizing compiler with auto-SIMDization features based on IBM XL compiler technology.

Although IBM Cell was earlier designed and built for custom gaming consoles, the past few years have seen the contribution of Cell well beyond the realm of the gaming industry. The Cell-based blade gives better performance as compared to a single IBM Cell. Other prospective application areas may include HDTV sets, home servers, game servers

Table 2.1: Important CELL statistics

Observed clock speed	$>$ 4GHZ
Peak performance (single precision)	$>$ 256 GFlops
Peak performance (double precision)	$>$ 26 GFlops
Local storage size per SPU:	256KB
Area	221 mm^2
Technology	90nm SOI
Total number of transistors	234M

and supercomputers. Cell has evolved from being a single chip in the past to being a fully scalable system today. The number of SPUs in the Cell can be easily varied to achieve different power/performance and price/performance tradeoffs. The Cell architecture was conceived as a modular, extendible system where multiple Cell subsystems can form a symmetric multiprocessor system.

Chapter 3

PREVIOUS WORK

Extensive literature is available on the broad topic of SPM management. Existing work on SPM allocation can be divided roughly into two broad categories - statically allocated and dynamically managed scratchpad memories. In statically allocated SPM allocation, the scratchpad memory is initialized with the designated program segments at load time and its contents do not change during run time. In contrast, dynamically managed SPM, as the name suggests, is characterized by change in the contents of the SPM while the program executes. The program points where code and/or data are moved back and forth from the SPM to the main memory are usually predetermined locations, generally immediately before a substantial change in program behavior (e.g., before loops). Both statically-allocated and dynamically-managed SPM can be further sub-classified into techniques that consider only code (instructions), or only data, or both.

Static SPM allocation techniques have been extensively explained in [3, 4, 7, 20, 17, 23]. Most of these approaches require prior knowledge of the SPM size at the compile time itself, except [17]. In [3, 4], dynamic programming approach for SPM allocation is done in order to select code blocks, which subsequently leads to higher energy savings. While [3] requires special hardware support to split the SPM into several partitions, [4] uses a post-pass optimizer to modify the necessary instructions so that the application runs on a unified SPM. Another method [7] solves the static assignment with a Knapsack algorithm, both for code and data blocks. In [23], memory objects are selected based on a cache conflict graph obtained through cache hit/miss statistics. The optimal set of memory objects is selected using an Integer Linear Program (ILP), which is a variant of the Knapsack algorithm. Yet another technique [17] is a profile-based approach where some profiling information has to be embedded into the application binary. The decision of which blocks should go to the SPM is delayed until the application is loaded, making it independent of the actual scratchpad memory size. The work in both [10] and [22] aims at multi-tasking systems. While [10] proposes an Application Program Interface (API) that helps the programmer move blocks back and forth between the SPM and the main memory, [22] deals with an automatic approach. The work in [12] presents three sharing strategies - non-saving, saving, and hybrid. In the non-saving approach, the scratchpad memory is completely allocated to the currently active task. The saving approach divides the scratchpad evenly between the applications. The hybrid approach is a combination of the saving and non-saving method. Since [22] presents a static allocation method, both the SPM size as well as the sharing strategy must be decided at compile time.

The second category of techniques i.e dynamically allocated SPM algorithm are explained in depth in [9, 12, 14, 15, 16, 19, 21]. These approaches address code as well as data allocation issues to memory. The methodologies in [14, 15] consider data arrays from well-structured loop kernels. In such techniques, the arrays of data are split into tiles in order to allow only part of an array to be copied into SPM. This allows arrays that are bigger in size to the SPM to still be allocated to SPM since only part of the array is actually present at any given point in time. Along similar lines, data arrays are also utilized in [16] to assign to SPM. In order to determine the most beneficial data array, registers are first assigned to so-called register classes based on their size. Each register class gets a fixed size of the SPM. Using a conflict graph of live ranges, a graph coloring algorithm determines which array is to be allocated to the SPM at what program points.

Performance optimization and consideration for local and global data are the focal points of [21]. This work examines the relationship graph among the data, called as the Data Program Relationship Graph (DPRG). This is generated with the help of the program control flow graph, annotated with timestamp. Using greedy heuristics on DPRG, the most beneficial sets are copied to and from the SPM with the help of pre-defined copy points. The most promising set of loops for SPM are determined in [9] using the ILP approach. Code allocation to SPM by using concomittance metric is discussed in [12]. This metric is a representation of the correlation of execution time of different code blocks. In [19], code blocks are directly copied into memory just before the loop. To find the optimal solution, ILP is used instead of graph colouring algorithm. Work in [18], introduces the SPM code mapping as a binary integer linear programming problem and also proposes a heuristic, determining simultaneously the region along with the function-to-region mapping.

Finally, in [8], an SPM allocation scheme for heap data is proposed. Promising candidates are assigned a fixed-size bin that can hold *n* elements of a dynamically allocated variable. At runtime, only the first *n* objects are allocated to the SPM. While the bins are fixed in size, their memory location may change during program execution, making this allocation technique a dynamic one. A dynamic approach which uses a binary integer programming to find region sizes and function-to-region mapping is discussed in [18]. The extension to this work is proposed in [13] which capture the temporal ordering of functions. It uses a conservative estimate of the interference cost between functions to generate a overlay mapping.

In practical applications, the code size is generally very large as compared to the available scratch pad memory. In such a small available memory, both code and data need to be appropriately managed. Code overlay can be used to do this successfully and efficiently. In code overlay, the required code is uploaded and offloaded into and from the scratch pad memory. This must be done using some software and it should be efficient enough to give good performance. This work focuses on the code overlay approach proposed in [6] along with new proposed overlay mapping technique for the analysis of pre-fetch mechanism. Implemented pre-fetch technique optimizes the code overlay scheme to reduce overlay overhead cost.

Chapter 4

SPU CODE OVERLAY

Within the purview of general computing, overlaying means "replacement of a block of stored instructions or data with another". Overlaying is a programming technique that allows programs to be larger than the processing unit's available memory. An embedded system would normally use overlays because of limited available physical memory (which is the internal memory for a system-on-chip or SoC) and lack of virtual memory facilities. This is because most SOCs have high performance computation units with restricted memory resource.

4.1 Introduction

Most of the application programs running on desktops with general purpose processing units load the complete program into the main memory and execute it. This is the most optimal and efficient way to perform the operation. However, in case of embedded systems, the main memory of the processing unit is limited. In the IBM Cell, each SPU unit has its own LS memory of size 256KB. Loading the complete program onto the LS before its execution does not work for SPU due to its memory constraints, unless the program is a small computation task such as a limited dimension matrix multiplication, which requires relatively low memory for both code and data. The programs that are run on these units are most definitely of the order of a few hundreds of KB, without taking data into consideration. When the sum of the code and data lengths of the program exceeds the LS size, it is necessary to use overlays. Overlays may also be used in certain other circumstances; for example, performance improvement may be achieved if the size of data areas can be increased by moving rarely used functions to overlays.

Overlay mapping consists of assigning memory regions with more than one segment mapped onto it. A segment is the smallest unit which can be loaded as a single logical entity during execution. It contains program sections such as functions and data areas. A region may contain more than one overlay segment, but a segment will never cross a region boundary. The size of a region is the largest segment size among its assigned segments. Regions with only single segment mapping are not considered as overlay regions.

The overlay feature is supported for Cell SPU programming but not for PPU programming. This is because the PPU has cache which can hold large sized programs for execution. The overlaid program segments are not loaded onto the SPU LS before the main program begins its execution. They actually reside in the Cell main storage until that segment is required to be executed. When the SPU program invokes code in one of the overlay segments, and this segment is not currently present in the local store, then it is first transferred to the LS using DMA, from where it can be executed. This transfer will usually overwrite another overlay segment in the LS. This overwritten segment will most likely be from that part of the memory region which is not immediately required by the execution program.

In an overlay structure, the LS is divided into a root segment, which is always in storage, one or more overlay regions, where overlay segments are loaded as and when needed, and at times, regions with only one segment assigned to them. The root segment is loaded to the LS before program execution. Similarly, regions with unique segment assignments are also initially loaded since these regions do not undergo segment loading and offloading. Any given overlay segment will always be loaded onto the same region. This is governed by the overlay manager with the help of overlay mapping schemes provided in the *linker script*.

4.2 Overlay Working

The challenge of fitting large codes into the limited LS can be addressed through the generation of overlays by the linker or by providing one's own overlay scheme. Two or more code segments can be mapped to the same physical address in the LS. The linker plays an important role of generating call stubs for each segment, for all the regions, and the associated tables, which has all the tags stored for the reference of the overlay manager. These stubs and tables, both reside in the root segment. Instructions to call functions in the overlay segments are replaced by branches to these call stubs, which load the function code to be invoked, if necessary, and then branch to the function.

In most cases, all that is needed to convert an ordinary program into an overlay program is the addition of a linker script to structure the module. The script specifies which segments of the program can be overlaid in which region. The linker then prepares the required segments so that they may be loaded when needed during program execution. It also adds supporting code from the overlay manager library.

Once a particular function call is made by the currenty executing program segment, the overlay manager uses the table to check whether the requested segment containing the called function is already in the LS. If it is already present, the program sequence jumps to the starting address of the target segment and begins execution from there. If it is not present in the LS, the segment is loaded into the appropriate memory region, to its specific memory address during run-time, by performing DMA operation. This loading process overwrites the existing segment present in that location. The DMA command is issued, controlled and executed by the overlay manager. Before jumping to the target address once the segment has been loaded, the overlay manager also checks to ensure successful completion of the DMA process to avoid any unwanted behaviour in the program execution.

4.3 Overlay Limitations

Overlay can be of both, code and data. However, in general, when using overlays, the scope of the data restricts the freedom of managing code overlay. Conventional practice dictates that code sections must be grouped along with the data segments used by them. Due to such an efficient grouping, the data segments are used only by the segment which is currently loaded in a region and executing. This data is swapped out or written back to main storage only when a new segment is loaded, which also loads its own data to operate. Ideally all data sections are kept in the root segment which is never overlaid, provided that the data segments are not too large in size, which is a quite rare for present-day embedded systems applications. To address the issue of data size, sections for transient data may be included in the overlay regions, which is also swapped in and out as per requirement. However, the subsequent implications for this should be carefully considered. Most of the time, it is advisible for the user to analyze and handle the data segments to have error free results.

4.4 How to use Overlays

The process of overlay structuring involves performing code size estimation for a given application program. Some knowledge of data segment requirements also helps to get a better insight into the problem. Based on the available code and data requirements of a particular application, the first few things to be considered are as follows:

1. number of overlay regions that are required
2. number of segments which will be overlaid into each region
3. number of functions within each segment

At this stage of overlay structuring, it is better to overestimate the number of segments to produce conservative deductions. This is because it is easier to combine segments later rather than break up oversize segments after they are coded. The best practice may be function level resolution for segments.

Overview

The structure of a SPU overlay program completely depends on the relationships between the segments within the program code. Two segments which do not have to be in LS at the same time can be overlaid in the same region where they share an address range. These segments are assigned with the same load addresses, which is the starting address of the region. This does not create a problem, as they are loaded only when called. For example, code segment which reads the data from some input and writes back after the computation, need not be loaded into the memory at the same time.

Some program sections are required to be present in the LS at all times of the program execution. These sections are grouped together and placed in a special segment called the root segment. This segment remains in storage throughout the execution of the program. Some overlay segments may be called by several other overlay segments. This can be optimized by placing the called and calling segments in separate regions. In other words, such segments are not overlaid. Designing an overlay mapping scheme requires a comprehensive scrutiny of the program code. This facilitates in deciding which code sections need to be kept in the root segment for continuous availability and which code segments need to be overlaid such that they occupy the same LS locations at different times during execution.

While making decisions for overlay mapping, the finest resolution is at the function level. For any region, its minimum size is the size of the largest overlaid function mapped onto it. Linker checks for size constraints and throws an error if the function does not fit into the LS. This can be easily handled by splitting the function into multiple smaller functions without changing it's functional behaviour on the whole.

Another important factor to be considered is the total size of the SPE executable. Overlays cannot use complete available LS memory to implement the mapping scheme. Overlay manager code, tables and stubs for segments all need to be in the LS at all times during the execution of program code. Overlay manager does the job of loading and offloading segments, as and when needed. It uses the table entries for these operations. The stubs act as the starting point for the segments which are overlaid. Basically, stubs are called when a segment is called by the program. For a program with *s* overlay segments in *r* regions, making cross-segment calls to *f* functions, this infrastructure requires the following amounts of local storage:

$$\begin{aligned}
&\text{manager: about } 400 \text{ bytes},\\
&\text{tables: } s * 16 + r * 4 \text{ bytes},\\
&\text{stubs: } f * 8 + s * 8 \text{ bytes}.
\end{aligned}$$

The numbers shown above indicate the absolute minimum requirement for the LS.

Leaving this memory aside, the rest of the LS can be used for code overlay and data. Besides these, the design of overlay schemes does not have any restrictions on the number of overlay segments or regions supported.

Overlay Structure Example

Consider the following example to illustrate overlay structuring, having eight functions including a `main` function which acts as the source point. Functions are labelled as func_A through func_G. The individual functions size are shown in the Table 4.1. The sizes of the functions are selected in such a way that the total code size exceeds the size of the LS. To find the best available overlay mapping scheme, the relation between functions needs to be understood. Figure 4.1 shows the tree structure or call graph of the code, which highlights the relations between different functions. The functions are assigned to five different segments as shown in Figure 4.1. It also helps in understanding how the segments use the LS at different times.

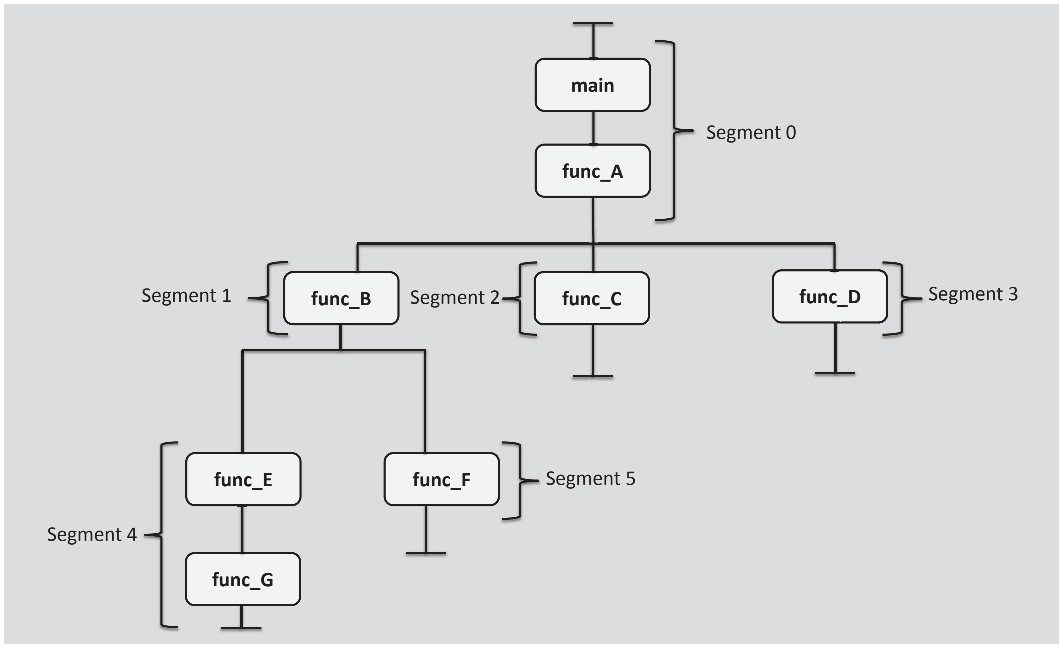


Figure 4.1: Overlay structure

Some of the questions that need mention are - What if the code is not overlayed?

How much total LS is required in that case? If the program did not use overlay mechanism, it would require 330KB of total storage, which is the sum of all sections. In order to fit the 330KB size of code in less than 256KB of LS, overlays are used. Segments which do not require simultaneous presence in the memory can be assigned to one region. As shown in the Figure 4.2, segments are assigned to each region. Segment 0 is the largest segment in region 0, and similarly, segment 2 in region 1 and segment 4 in region 2.

The position of the segments in the mapping scheme does not determine the order in which they will be called. Any segment may get called at any time during program execution and this completely depends upon the program code logic along with the input data. The same segment may get called, and consequently get loaded, into the LS multiple times due to the change in the sample input.

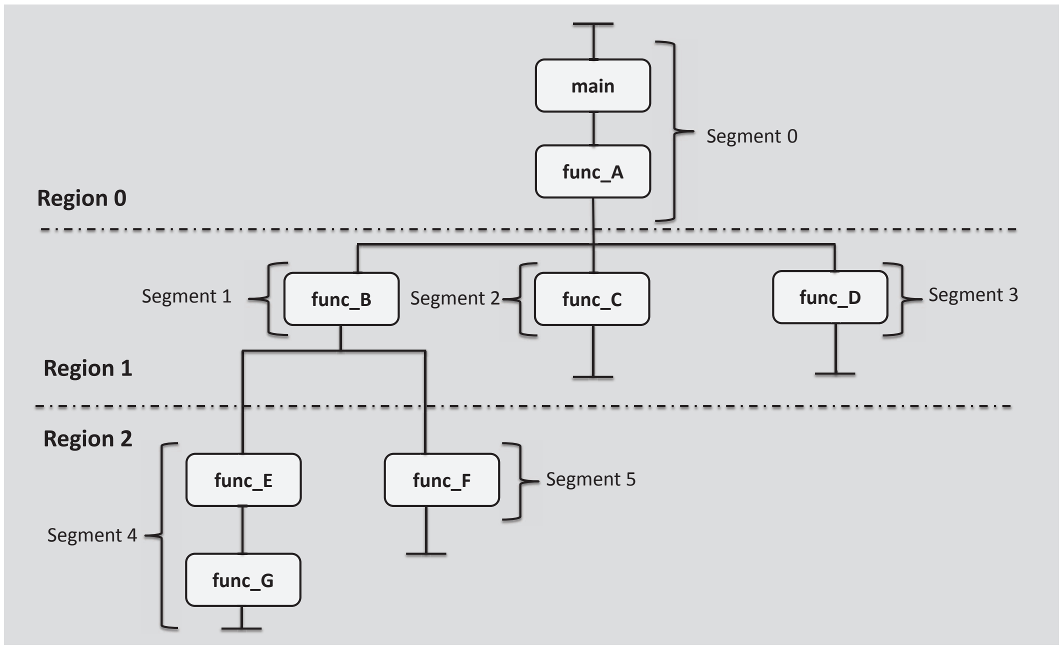


Figure 4.2: Overlay structure with region assignment

Load Point for Segments

Conventionally, the root segment is the initial segment assigned by the linker, which has 0x80 as the starting address. The rest of the regions and corresponding segments get load

Table 4.1: Sample program lengths

Section	**Length (in bytes)**
main	15,000
func_A	25,000
func_B	40,000
func_C	70,000
func_D	50,000
func_E	85,000
func_F	30,000
func_G	15,000

Table 4.2: Segment load points

Segment	**Load point**
0	0x80 + 0
1	0x80 + 40,000
2	0x80 + 40,000
3	0x80 + 40,000
4	0x80 + 110,000
5	0x80 + 110,000

addresses relative to the previous load addresses, along with this 0x80 offset. Segments belonging to the same region have the same load address. For example, segments 1, 2 and 3 mapped to region 1 will have the same address. This segment origin is also called the load point, because it is the relative location where the segment is loaded. Table 4.2 indicates the load points for each segment in the sample program.

Overlay Manager

When a program execution begins, it may invoke code from a particular segment. If that segment is not present in the LS, then it needs to be brought in via DMA from the main memory. The controller which controls the overlay operations is called *overlay manager*. The overlay manager checks to see whether the segment is already loaded due to previous calls. If not, then it initiates the DMA transfer of the segment from main memory to the appropriate load point. No overlay occurs if one section calls another section which is in a segment already in the LS.

The overlay manager uses special stubs and tables. These stubs and tables are generated by the linker automatically and are part of the output program module. Special stubs are used for each inter-segment call. The tables generated are the overlay segment table and the overlay region table. With the help of these tables, the overlay manager decides which segment is currently loaded in a particular region. These tables also give information about where to load the segment along with its size. Figure 4.3 shows the location of the call stubs and the segment and region tables in the root segment of the sample program. These table sizes need to be considered when planning the resource usage of the LS.

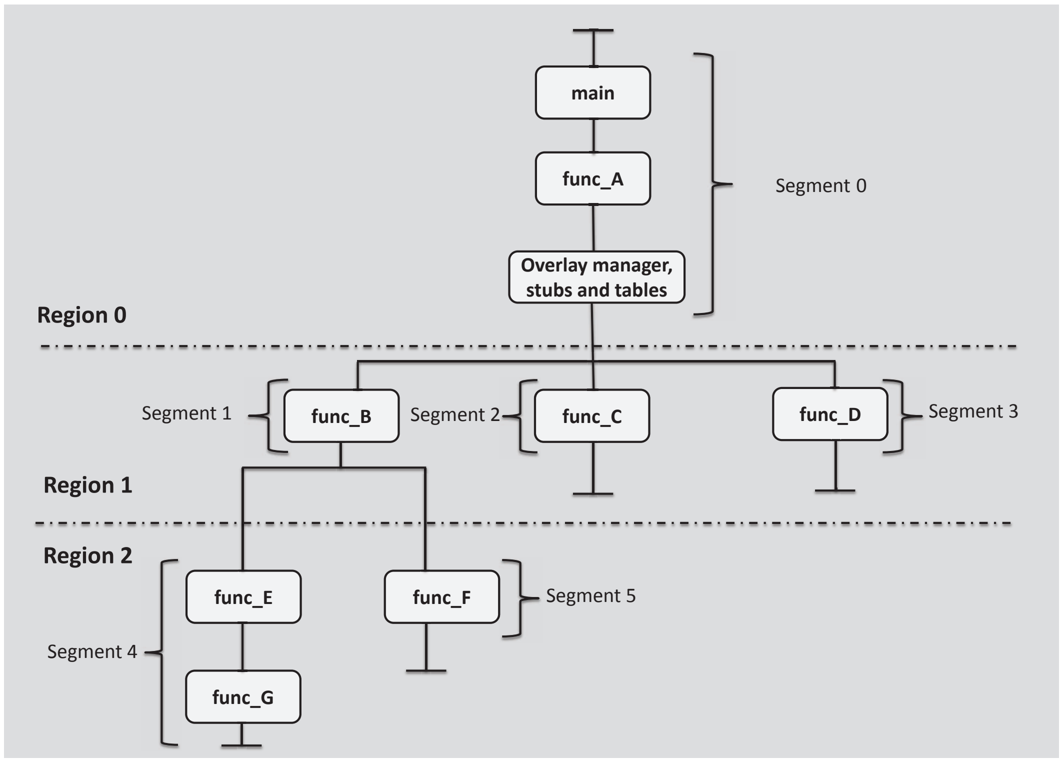


Figure 4.3: Overlay structure with overlay manager, stubs and tables

Figure 4.4 shows code overlay mapping with addresses labeled across the bottom.

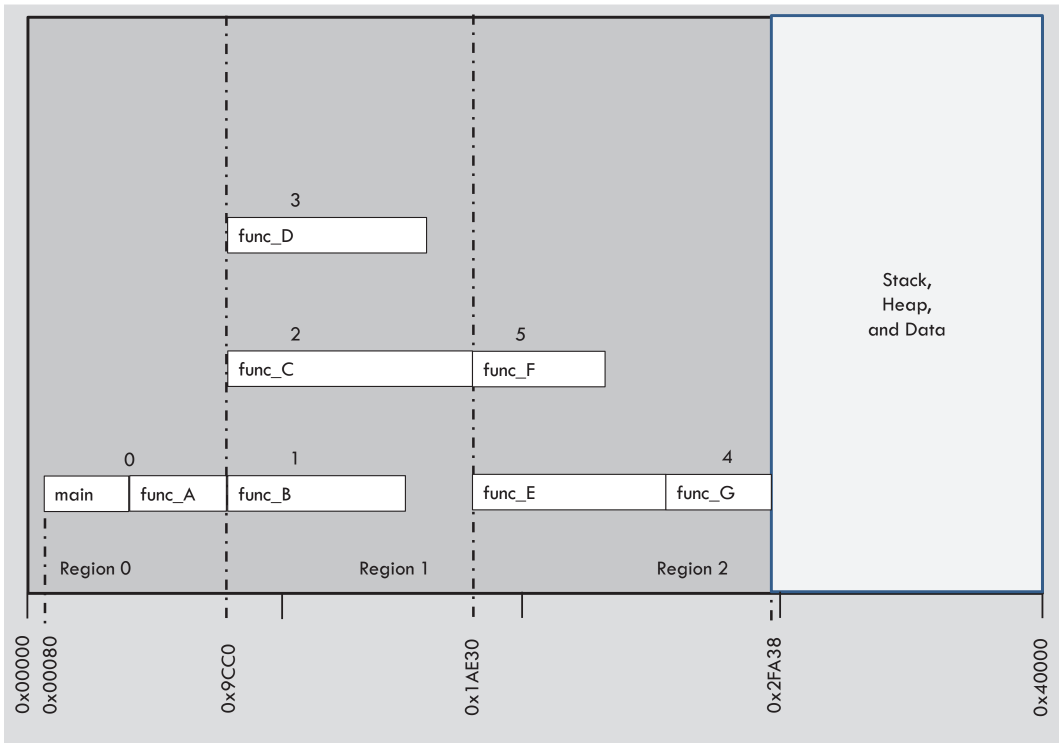


Figure 4.4: Memory layout for given code example

Call Stubs

There is one call stub for each function in an overlay segment, which is called from a different segment. No call stub is needed if the function is called from within the same segment. All call stubs are in the root segment. During execution, the call stub specifies the specific segment to be loaded and also the segment offset to the overlay manager, to transfer control to invoke the function to be accessed.

Segment and Region Tables

Each overlay program contains one overlay segment table and one overlay region table, located in the root segment. Section 4.6 talks about it in detail. The segment table contains static (read-only) information about the relationship of the segments and regions in the program. During execution, the region table contains dynamic (read-write) control information such as which segments are loaded into each region. Unlike segment table, region table is dynamic because it updates the segment number which is currently loaded.

4.5 Linker Script for SPU Overlay Program

Once the overlay structure is designed, the program should be arranged into the structure. The complier compiles the code to generate the object files which the linker uses to make the executable. For this, the linker needs to know the sections in each segment, and the mapping of segments to specific regions. This can be done by providing a linker script file which contains `OVERLAY` statements. The specifications given in the linker file are as under.
Regions are defined by each `OVERLAY` statement. Each `OVERLAY` statement begins a new region.
Segments are defined within an `OVERLAY` statement. Each segment statement within an overlay statement defines a new segment. Additionally, it provides a means to equate each load point with a unique symbolic name.
Sections are positioned in the segment specified by the segment statement with which they are associated.

The origin of every region is specified with an `OVERLAY` statement. Each `OVERLAY` statement defines a load point at the end of the previous region. The load point is logically assigned a relative address that follows the last byte of the largest segment in the preceding region. Subsequent segments defined in the same region have the same load point as their origin.

In the sample tree overlay program, two load points are assigned to the origins of the two `OVERLAY` statements and their regions, as shown in Figure 4.2. Segments 1, 2 and 3 are at the first load point whereas segments 4 and 5 are at the second load point.

The following sequence of linker script statements results in a structure as shown in Figure 4.1

```
OVERLAY     {
                 .segment1 {./func_B.o(.text)}
                 .segment2 {./func_C.o(.text)}
                 .segment3 {./func_D.o(.text)}
            }
OVERLAY     {
                 .segment4 {./func_E.o(.text)
                            ./func_G.o(.text)}
                 .segment5 {./func_F.o(.text)}
            }
```

Note: main and func_A are not specified in the `OVERLAY` statements because they are a part of root segment. In other words, the segments not specified in the linker file, automatically become a part of the root segment. They also do not have entries in the tables since the overlay manager does not have to load or offload them. Regions with single segment mapping also have similar features.

4.6 GNU SPU Linker

The GNU SPU linker makes use of object files, object libraries, linker scripts, and command line options to generate a fully or partially linked object file. The linker script is used to control the generation of overlays as this allows maximum flexibility in specifying overlay regions and in mapping input files and functions to overlay segments. One or more overlay regions can be easily created by inserting multiple `OVERLAY` statements in a standard script as described in Section 4.5. No further modification of the output section specifications, setting the load address for example, is necessary. In addition, by defining loadable output sections with overlapping virtual memory address (VMA) ranges, it is possible to generate overlay regions without using `OVERLAY` statements.

Once the linker detects overlays, it automatically generates the data structures used to manage them. It also scans all non-debug relocations for calls to addresses which map to overlay segments. The linker generates an overlay call stub for such called addresses and remaps the call to branch to that stub in response to such calls. This is for calls that are apart from those used in branch instructions within the same section. At execution time, these stubs call an overlay manager function which loads the overlay segment into storage, if necessary, before branching to the final destination.

If the following linker command option is specified: `-extra-overlay-stubs`, then the linker generates call stubs for all calls within an overlay segment, even if the target does not lie within an overlay segment (for example, if it is in the root segment). Note that a non-branch instruction, referencing a function symbol in the same section will also cause a stub to be generated. This ensures that function addresses which escape via pointers are always remapped to a stub as well.

The overlay management data structures that are generated include two overlay tables in the `.ovtab` section.

`_ovly_table:`

This table has only one entry per overlay segment. The overlay manager has only read permission for this table. This table should never change during execution of the program. It has the format:

```
struct {
        u32 vma;        // SPU local store address that the section is loaded to
        u32 size;       // Size of the overlay in bytes
        u32 offset;     // Offset in SPE executable where the section
                        //can be found
        u32 buf         //One-origin index into the _ovly_buf_table
      } _ovly_table [];
```

_ovly_buf_table:

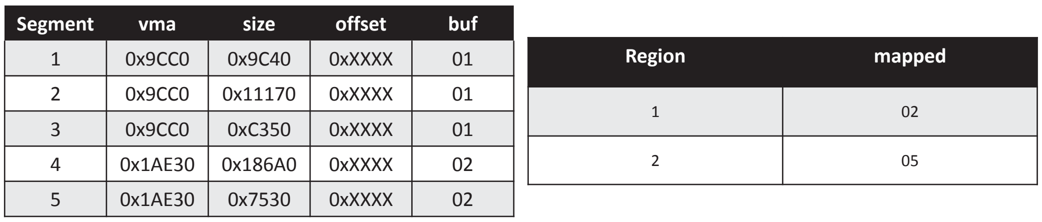
This table has only one entry per overlay region. The overlay manager has only read-write permissions for this table. It changes to reflect the current overlay mapping state. The format is:

```
struct {
          u32 mapped;     // One−origin index into  _ovly_table  for the
                          //  currently  loaded overlay. 0 if none
         }  _ovly_buf_table [];
```

Note: These tables, all stubs, and the overlay manager itself must reside in the root (non-overlay) segment as shown in the Figure 4.3.

Whenever the overlay manager loads a segment into a region, it updates the `_ovly_buf_table` with the corresponding segment number. The overlay manager may be provided by the user as a library that contains the routines `__ovly_load` and `_ovly_debug_event`. If these routines are not provided, the linker will use a built-in overlay manager which contains these symbols in the `.stub` section [25].

Figure 4.5 shows contents of `_ovly_table` and `_ovly_buf_table` for given example. In `_ovly_table` coulmn `vma` indicates the spu local address where corresponding segment is loaded which can be verified with the memory layout shown in Figure 4.4. Coulmn `buf` shows to which region corresponding segment is mapped. In `_ovly_buf_table` column `mapped` shows which segment is currently loaded into the region.



Segment	vma	size	offset	buf
1	0x9CC0	0x9C40	0xXXXX	01
2	0x9CC0	0x11170	0xXXXX	01
3	0x9CC0	0xC350	0xXXXX	01
4	0x1AE30	0x186A0	0xXXXX	02
5	0x1AE30	0x7530	0xXXXX	02

Region	mapped
1	02
2	05

Figure 4.5: Contents of `_ovly_table` and `_ovly_buf_table` for given example

Chapter 5

IMPLEMENTATION

5.1 Problem Statement

Previously proposed algorithms for code overlay have been found to perform well, however, there is much scope for improvement. Loading and offloading operations are performed for every function call, which is fetched according to the code overlay scheme. Hence, there is a fair amount of overhead cost involved due to the overlay manager being invoked numerous times. Consider the example of an application which has hundreds of functions and where, each function is being called a number of times. For every function call, program execution must stall for the code to be loaded into the size-limited SPM from main memory. In such a case, the overlay manager overhead becomes significant.

5.2 Proposed Solution

In order to reduce this overhead, we may perform function pre-fetching where functions are fetched to the memory regions even before they have been called. The overlay manager will be notified in advance to fetch the function to the SPM, while the application code is performing some other computational work or DMA. In the normal scenario, overlay manager performs DMA to get that function when it is called. Additionally, the overlay manager must wait until the DMA operation is completed. With pre-fetching, DMA is launched in parallel along with some other DMA or computational work.

The technique proposed here for pre-fetching is a manual approach, where the programmer has to add pre-fetch calls to initiate DMA. Automatic version of pre-fetch may also be used, which does not require programmer intervention. This can be done by modifying the existing overlay manager which comes with the compiler. Overlay manager does the work of loading the required code to the SPM. This work intends to add extra functionality to the overlay manager so that the programmer can ask the overlay manager to pre-fetch the code to the SPM. The added functionality does the job of fetching the next expected section of code to the SPM. While doing this, the overlay manager must ensure that it is not overwriting the same overlay region from which it was called. With the help of the code snippet shown in the Figure 5.1, the pre-fetching technique is illustrated along with the performance gain possible with this mechanism.

```
MAIN ()
    F1 ()
    FOR
            f2()
    END FOR
    WHILE
            F3()
    END WHILE
    F4()
    F5()
END MAIN
```

```
F2()
    //some calculation
     F6()
END F2
F3()
   //some calculation
  F7()
END F3
```

Figure 5.1: Pseudo code example

Main starts running and it calls function F1(). If that function is not in the required region, then the manager will initiate DMA. Once DMA is finished, it will start executing the function. Now when function F1() is running, we can check to see which function is called next . Function F2() is called next, and so, a check is performed first to see if it is present in the region or not. If not, DMA is initiated for F2(). Here, the advantage is that the manager will not wait for this DMA to finish because it is doing computations of function F1(). The manager only initiates the DMA, which runs in the background, while other work can be performed. This is with respect to a very small example code. Embedded systems have a large number of functions and data. We need to manage these functions in the overlay. In order to achieve the best out of the architecture, effective overlay schemes are required and we can use the pre-fetching concept in order to add more performance. Sections 5.3 and 5.4 discuss code overlay schemes followed by details of pre-fetching algorithm in Section 5.5.

5.3 Random Code Overlay Mapping for Generating Overlay Regions

Random Code Overlay Mapping (RCOM), is used to generate overlay mapping for available SPM size. RCOM is a relatively simple mapping technique. It works better than the IBM Cell SPU complier, *spu-gcc* because it generates overlay mapping with multiple regions if possible. It uses the function sizes to generate the mapping. As the name suggests, it involves random segment mapping to the region, without crossing the memory constraint. The RCOM algorithm creates overlays using function or object file resolution, where an object file may contain one or more functions. The level of resolution it uses is limited by the GNU linker *ld*. This linker takes a script (a .script file) which describes the regions mapped with the code segments, where these segments are nothing but the object files. Overlay mapping has one or multiple regions depending upon the available size, with one or more segments mapped to each region.

RCOM algorithm is presented in Algorithm 5.1. For each function, f, to be assigned from the set of functions that have been sorted by their sizes, RCOM checks whether *remaining_mem* of SPM is sufficient to create a new region and assign function f to it. A new region is created only when memory size is not overflowing. Otherwise function f is assigned randomly to any region R from the current overlay mapping. Worst overlay performance occurs when available memory restricts overlay mappings to a single region that is the size of the largest function in the program, or the smallest possible size for a valid mapping. With one function per segment, interference overhead occurs between every function switch, during every function call and return. The computational complexity of this simplistic algorithm is $O(n)$.

5.4 Code Overlay Generator for Generating Overlay Regions

In order to perform pre-fetching, an overlay scheme first needs to be generated. The algorithm used is “Code Overlay Generator” (COG), used for producing high performance dynamic SPM code mappings [6]. These SPM mappings have regions specified with the

Algorithm 5.1 RCOM algorithm

```
1: $remaining\_mem$ = unused instruction memory
2: $OVL$ {set of r empty overlay regions}
3: $R$ {an overlay region in $OVL$}
4: $\vec{F}$ {functions sorted from largest to smallest function sizes}
5: for $\forall f : f \in \vec{F}$ from largest to smallest function sizes do
6:     if $size(f) < remaining\_mem$ then
7:         create a new region and add $f$
8:     else
9:         add $f$ to $R$, any region form exiting mapping randomly
10:    end if
11: end for
12: return $OVL$
```

segments assigned to them. The high performance overlay mapping generated using this algorithm minimizes the overall overhead. It has couple of extensions for the purpose of achieving extra optimization as compared to the original one, by considering some more constraints. These extensions are called as “COG Expansion” and “COG compression”. These optimized algorithms perform better than the automatic overlay mapping generator in the IBM Cell SPU compiler, *spu-gcc*.

The COG algorithm is designed to produce overlays which result in the smallest possible number of misses. This algorithm generates high performance overlays by analyzing the interference costs calculated to minimize the overall cost of the overlays. The algorithm starts with fixed number of regions and generates mapping with minimum misses without considering the function sizes. So, the generated mapping may be bigger than the available size. To account for this, the algorithm does multiple iterations, each of which starts with lower bound of regions and generates mappings with increasing region number without overflowing memory limitations. The overlay mapping with the best result is considered after every iteration.

In the worst case the algorithm runs n times for overlay mappings i.e. 1 to n regions. For each run, n functions are allocated and at most n comparisons are made for each function to find the best region. This gives rise to a computational complexity of $O(n^3)$. The algorithm is explained in [6].

5.5 Pre-fetching Algorithm

The pre-fetching algorithm is an extension or plug-in to the existing SPM mapping algorithm. This can be run for any mappings available for a specific size of SPM. This is basically implemented in the overlay manager which is provided by the IBM Cell SPU compiler, *spu-gcc*. Overlay manager works as a subset of the *spu-gcc* compiler to do all segment loading and off-loading, whenever necessary. The implemented pre-fetching works along similar lines, except that it loads the content to the regions well in advance of the actual invocation.

Implementation includes the code to pre-fetch the code segment into the memory, whenever the user adds explicit call to do the pre-fetch operation. It is implemented as a simple function call in C, which acutally invokes the assembly code to perform DMA. The function is `__prefetch_load(int);` where, the only parameter is the index number of the segment which is to be pre-fetched. This number is the segment number in the overlay mapping specified in the provided linker script. As discussed in Chapter 4, the overlay manager has the table `_ovly_table` which has one entry for each segment. This number is used to refer to the details of the segment from the `_ovly_table`. This table stores the SPU LS address that the segment is loaded onto. By performing the indexing to the table, the SPU address is obtained to initiate DMA from the main memory. The table entry also indicates the size of the segment to be loaded from starting address.

This DMA process is performed concurrently along with the current execution of the function. Since this DMA operation overlaps with the computation work or/and with other DMA operation, the communication overhead is reduced.

Process of Pre-fetching

1. The pre-fetch process checks with the `_ovly_buf_table`, which has one entry for each region with the number of currently loaded segment in that region. It also checks the lower bit of the `.size` parameter of the `_ovly_table` which indicates segment

size. If the segment is already loaded into the region, the DMA initiation operation is skipped. Otherwise, it retrieves the segment details from the table `_ovly_table` with the help of new overlay segment number.

2. Once it has the starting address and the size of the segment to be loaded, it initiates DMA by issuing the `MFC_Cmd` command. Before executing this command, the pre-fetch code sets some other parameters such as transfer size to control the execution of the DMA.

3. While DMA runs in the background, the lower bit of `.size` needs to be cleared in the `_ovly_table` entry corresponding to the evicted segment, and set at the entry for the newly loaded overlay segment.

4. Since, it is just pre-fetching and not loading the content at the function call, it will not branch to the target address as it does in case of normal segment loading.

Pre-fetch Approaches Implemented

In order to verify and analyze the overlay overhead cost reduction obtained by the use of pre-fetch code, the user has to insert the pre-fetch call into the code where the pre-fetch operation is to be performed. This work analyzes the performance gain due to pre-fetch for the given overlay mapping scheme, which is generated by existing SPM mapping algorithms COG [6] and the proposed RCOM as discussed in Section 5.3. These pre-fetch approaches either based on data obtained by *gprof* tool or profile data generated from Overlay Miss Count Model (OMCM) explained later.

Gprof Tool

Gprof [11] is GNU profile used to analyze the given source code. Profiler uses information collected during the actual execution of the program. In order to get profile data, first program needs to be compiled and linked with profile enabled. Secondly, execute the program to generate profile data file and last step is run *gprof* to analyze the profile data. The result of the analysis is a file containing two tables, the flat profile and the call graph. The flat profile in *gprof* shows the total amount of time the source code has spent executing each function and how many times that function was called. During interpretation, functions with no apparent time spent in them, and no apparent calls to them, are not mentioned in the gprof analysis, unless the '-z' option is given. The call graph shows how much time was spent in each function and its descendants. It also has the count for the function calls for each function. This work uses the function call count for pre-fetch prediction appraoches.

Overlay Miss Count Model

Overlay Miss Count Model (OMCM) is profiling tool proposed to find overlay misses for the overlay mapping scheme. This tool is implemented in overlay manager. To get the miss counts for all the functions, program needs to be run with input data. The profile data is generated and written into a separate file with function names and their corresponing overlay misses. The advantage of this tool is it's 100% accuracy. Since, it is implemented in overlay manager, tool takes care of all the misses for a function. Overlay miss occurs for any function during either function call or at function return. OMCM accounts for both these misses.

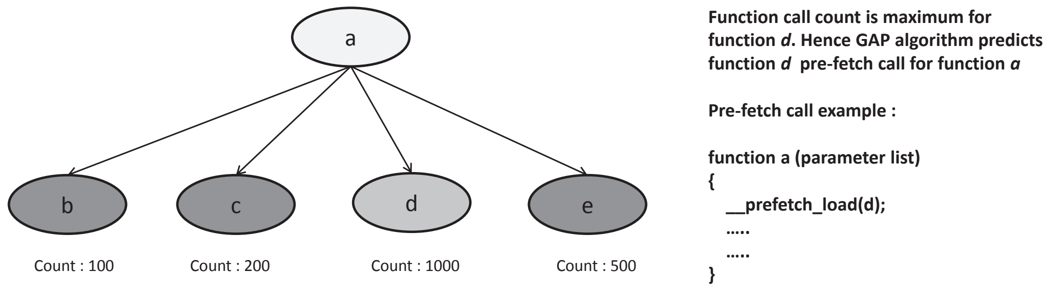
Algorithm 5.2 GAP algorithm

```
 1: $PREFETCH\_CALL$ {set of n empty pre-fetch predictions }
 2: $OVL$ {set of r overlay regions mapping}
 3: $\vec{F}$ {all the functions from the source code}
 4: $FUNCTION\_CALL$ {set of function calls for each function in $\vec{F}$}
 5: for $\forall f : f \in \vec{F}$ do
 6:     $prefetch\_pred = null$ {pre-fetch prediction for $f$ }
 7:     for $\forall fc : fc \in FUNCTION\_CALL[f]$ do
 8:         if $count(prefetch\_pred) < count(fc)$ && $region\_num(f)! = region\_num(fc)$ then
 9:             $prefetch\_pred = fc$
10:         end if
11:     end for
12: end for
13: return $PREFETCH\_CALL$
```

The following four approaches have been considered in this work for the pre-fetch prediction analysis.

(I) **Gprof data Analyzed Pre-fetch (GAP) :** This approach uses *gprof* data to predict the best suited pre-fetch function for almost all possible functions. Profiling with *gprof* has been done for the given source code to obtain *gprof* analysis. This data is analyzed for the total number of times the function is called and the total count of function calls made by one function to another. To predict the pre-fetch call, consider the function which is called the most. This is based on the data provided by the *gprof* tool. The code detects the pre-fetch function calls with the help of gprof analysis and overlay mapping scheme.

The algorithm for GAP prediction is shown in the Algorithm 5.2. In the worst case condition, the algorithm runs n times for all functions i.e. 1 to n functions. Each function can have maximum $n-1$ function calls. For each run, function calls are compared and at most $n-1$ comparisons are made for each function to find the best pre-fetch call. This gives rise to a computational complexity of $O(n^2)$.



Function call count is maximum for function *d*. Hence GAP algorithm predicts function *d* pre-fetch call for function *a*

Pre-fetch call example :

```
function a (parameter list)
{
   __prefetch_load(d);
   .....
   .....
}
```

Figure 5.2: GAP prediction exmaple

Figure 5.2 explains GAP algorithm with small exmaple. Function a calls functions *b, c, d, and e*. Each function is marked with function call count, indicates how many times the particular function is being called by function a. This count is obtained form *gprof* tool. GAP finds out the function with maximum function call count, function d in this case and code automatically inserts pre-fetch call as very first line inside function a as shown in Figure 5.2. While predicting it also checks that function d is not in the same overlay region as function a.

Limitation: The timing analysis given by *gprof* cannot be considered for the prediction. This is because it estimates them by making an assumption about the code that may or may not be true. It also does not consider the size of the function which contributes to the actaul DMA overhead. GAP might end up predicting a function with less DMA overhead as opposed to a function with more DMA overhead, simply because the former has been called more often. This indicates that DMA cost must be considered as a factor when predicting the pre-fetch calls.

Algorithm 5.3 CAP algorithm

```
1: $PREFETCH\_CALL$ {set of n empty pre-fetch predictions }
2: $OVL$ {set of r overlay regions mapping}
3: $\vec{F}$ {all the functions from the source code}
4: $FUNCTION\_CALL$ {set of function calls for each function in $\vec{F}$}
5: $DMA\_COST$ {DMA cost for each function in $\vec{F}$}
6: for $\forall f : f \in \vec{F}$ do
7:     $prefetch\_pred = null$ {pre-fetch prediction for $f$ }
8:     for $\forall fc : fc \in FUNCTION\_CALL[f]$ do
9:         if $count(prefetch\_pred) \cdot DMA\_COST[prefetch\_pred] < count(fc) \cdot DMA\_COST[f]$ && $region\_num(f) != region\_num(fc)$ then
10:            $prefetch\_pred = fc$
11:        end if
12:    end for
13: end for
14: return $PREFETCH\_CALL$
```

(II) **Count Analyzed Pre-fetch (CAP) :** CAP is a fully automated approach. It is also based on *gprof* `call_graph` data. To overcome the limitation described in (I), CAP incorporates DMA cost consideration for predicting pre-fetch calls. DMA overhead cost of a function, *f*, is calculated as a function of the segment size according to real DMA overhead measurements taken on the IBM Cell. This cost can be modeled to within an average error of 0.4% using the following equation:

$$DMA\_cost(f) = \begin{cases} 3.9E\text{-}5 \cdot size(f) + 0.17 \cdot \mu s & \text{:size(} f \text{)} \leq \text{2kB} \\ 7.3E\text{-}5 \cdot size(f) + 0.1 \cdot \mu s & \text{:size(} f \text{)} > \text{2kB} \end{cases}$$

While predicting a pre-fetch call, the overall code overlay cost is considered by using the above analysis along with function call counts, ultimately giving rise to better predictions. This approach definitely tries to minimize the code overlay cost further as compared to GAP.

The CAP algorithm is explained in the Algorithm 5.3. In the worst case condition, the algorithm runs n times for all functions i.e. 1 to n functions. Each function can have maximum $n-1$ function calls. For each run, function calls are compared and at most $n-1$ comparisons are made for each function to find the best pre-fetch call. This gives rise to a computational complexity of $O(n^2)$.

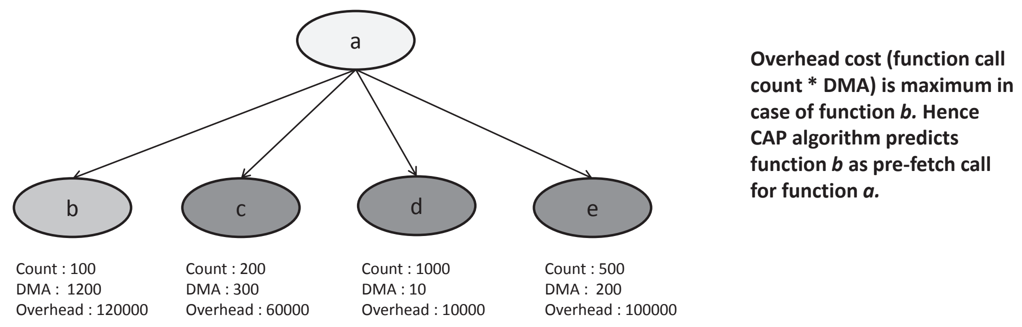

a
b
c
d
e
Count : 100
DMA : 1200
Overhead : 120000
Count : 200
DMA : 300
Overhead : 60000
Count : 1000
DMA : 10
Overhead : 10000
Count : 500
DMA : 200
Overhead : 100000
Overhead cost (function call count * DMA) is maximum in case of function *b.* Hence CAP algorithm predicts function *b* as pre-fetch call for function *a.*

Figure 5.3: CAP prediction exmaple

Figure 5.3 shows how CAP algorithm predicts the pre-fetch call for a praticular function with the same example used in (I). It uses additional information of DMA cost require to load the function into SPM. Overhead cost for function *b* is maximum, hence CAP algorithm predicts function *b* as pre-fetch call for function *a*.

Limitation: gprof data gives only function call counts. But in case of code overlay, misses happen for both function calls as well as call returns. The counts given by *gprof* does not address the return miss counts. Hence, it is imperative to take this factor also into account for prediction.

Algorithm 5.4 PAP algorithm

```
1: $PREFETCH\_CALL$ {set of n empty pre-fetch predictions }
2: $OVL$ {set of r overlay regions mapping}
3: $\vec{F}$ {all the functions from the source code}
4: $FUNCTION\_CALL$ {set of function calls for each function in $\vec{F}$}
5: $DMA\_COST$ {DMA cost for each function in $\vec{F}$}
6: $PROFILE\_COUNT$ {profile miss count obtained by using OMCM}
7: for $\forall f : f \in \vec{F}$ do
8:     $prefetch\_pred = null$ {pre-fetch prediction for $f$ }
9:     for $\forall fc : fc \in FUNCTION\_CALL[f]$ do
10:        if $PROFILE\_COUNT[prefetch\_pred] \cdot DMA\_COST[prefetch\_pred] < PROFILE\_COUNT[fc] \cdot DMA\_COST[f]$ && $region\_num(f) != region\_num(fc)$ then
11:            $prefetch\_pred = fc$
12:        end if
13:    end for
14: end for
15: return $PREFETCH\_CALL$
```

(III) **Profile overlay manager Analyzed Pre-fetch (PAP) :** In order to address the issue raised in (II), the PAP approach first runs the given overlay scheme and generates miss count for both function calls and returns. This count is obtained from the OMCM explanined in Section 5.5. The code then analyzes the miss count data for each function and decides which function should be pre-fetched so as to ensure low overlay overhead. The DMA cost is calculated from equation used in (II). This analysis actually deals with overlay misses at the system level. To get these overlay misses, OMCM tool is used, with added switch capability that allows run-time control. The overlay misses are counted for every segment loading when DMA is initiated, for both `__ovly_load` and `__ovly_return`. These stubs are called for every function call and return. This approach works better than GAP and CAP types of pre-fetch. PAP is explained in Algorithm 5.4. Computational complexity of PAP algorithm is also $O(n^2)$ as explained for GAP and CAP.

PAP algorithm prediction is explained in Figure 5.4 with the same example used in (I) and (II). The count shown for each function is obtained from OMCM tool. As shown in the Figure 5.4 overhead cost is mamximum for function *c* as comapre to all other functions, hence algorithm predicts function *c* as pre-fetch call for fucntion *a*.

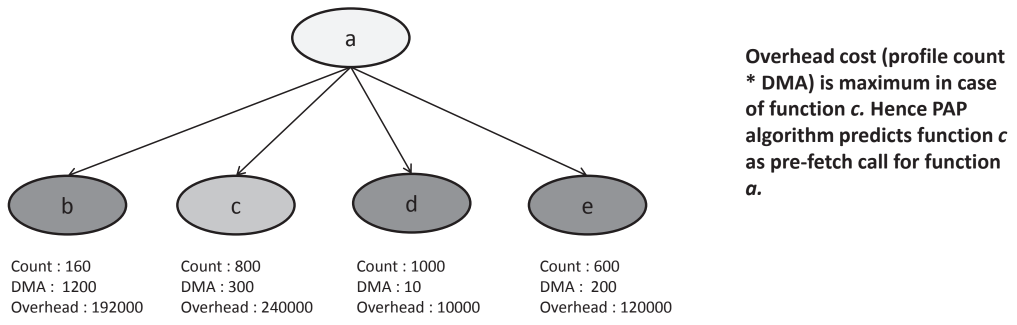


Figure 5.4: PAP prediction exmaple

(IV) **Hand Optimized Pre-fetch (HOP) :** HOP is proposed to extract improved performance out of pre-fetching. With HOP, user provides a `.prefetch` file which has pre-fetch calls specified, for example, if the user wants to pre-fetch function C when the program execution is in function A, the file will have entry saying function A $->$ function C. This is called hand-optimized pre-fetch since user has to manually analyze the code for its branch and jump control flow, along with loops in the code. In addition, the number of times a particular function calls the other function may also be considered. Sometimes, the actual execution time of a function helps add more granularity for prediction. Same application for a given overlay mapping scheme can end up having different pre-fetch calls due to different input data on which it operates. This is because code behavior varies with input data. HOP should give the best possible results since it is meticulously performed by the user and manual verification is also performed.

All the four approaches discussed above are automated as far as pre-fetch call exection is concerned. HOP has user involvement of providing the pre-fetch calls. Due to human involvement, if the prediction is error-free, then the expected performance gain is maximum, since human analysis will take care of all the possible constraints. The experimental results for both algorithms, COG and RCOM, with the above four pre-fetch prediction approaches are presented in Chapter 6.

Chapter 6

RESULTS

6.1 Experimentation Setup

Benchmarks and Memory Setup

The pre-fetch scheme has been implemented on the CBE. Results were collected for three benchmarks described in in Table 6.1. These three benchmarks, *ffmpeg*, *cjpeg* and *gsm_untoast*, were executed on the IBM Cell processor. The *ffmpeg* is a powerful and versatile video transcoder. The *ffmpeg* benchmark has been configured for H.264 decoding and modified to run on the CBE. In this work, a parallelized version of one of the popular codecs, H.264, from *ffmpeg* is used to run on IBM Cell [5]. H.264 decoder parallelized code has large number of functions as indicated in Table 6.1. The function call graph for the source code shows the tree is wide, but it is relatively shallow. It supports a large number of inter predictions modes for different block sizes such as 4x4, 8x8, 16x16 etc, along with number of intra prediction modes. The wide range of modes give better compression at the encoding depending upon the macroblock characteristics. The variety of inter prediction modes gives rise to a larger number of functions to chose from at runtime. The function calls in the decoder at runtime are depending on mode selected in the encoder and can vary widely from video to video or even macroblock to macroblock.

Cjpeg is the reference JPEG encoder/decoder, whereas *gsm_untoast* is audio encoder/decoder. These benchmarks were modified to a function-level resolution, so that they can be run on the Cell architecture with code overlay.

Table 6.1 also specifies the minimum and maximum code overlay sizes selected to run these benchmarks. For each benchmark, select memory sizes needed to test performance across a meaningful SPM instruction memory. To get these overlay operating boundaries, a simple methodology was employed :

1. This analysis starts with the largest function in the program which defines the minimum overlay size. This can be justified by the argument that even if there is only one

Table 6.1: Benchmarks

benchmarks	**functions**	**code size (B)**	**min overlay (B)**	**max overlay (B)**
cjpeg	210	129,320	3,736	129,320
ffmpeg	106	100,012	7072	100,012
gsm_untoast	41	12,504	2,172	12,504

overlay region, it must be large enough to accommodate the largest function.

2. The upper operating boundary is defined by the total size of the program, such that no overlay is needed.

3. The SPU LS memory sweep is performed in ten steps for an ascending order of progression, for each benchmark.

These selected benchmarks are comprised of a large number of functions, and consequently a large code size, are indicated in Table 6.1.

Overlay Memory Requirement Estimation

The chosen memory resources, as shown in Table 6.1, can be justified by considering the following instance of *ffmpeg* similar to [5]. The 256KB SPU Local Store SPM is partitioned and shared among the program instructions, program data, the execution stack, and heap for any dynamically allocated structures. The stack size is constrained by the tree depth of the function call graph. Stack memory requirements for this code are well bounded as there are no recursive function calls. Although the tree is wide, it is also relatively shallow. The experiments performed for *ffmpeg* code on IBM Cell helped determine that the stack may approach 100KB, which must be reserved in the LS memory map to prevent a stack overflow. The implementation includes just under 100 functions, totaling approximately 160KB in the overall LS memory requirements. Additionally, while minimizing memory requirements wherever possible, it requires at least 50KB in data memory, with 100KB necessary for stack/execution memory, which means that 100KB of the available 256KB

LS must accommodate the 160KB of code. This 60KB memory shortage is overcome using the code overlay technique. Thus, 100KB is the upper bound of code overlay size for *ffmpeg*, with the lower bound being the largest function size in the code. This methodology of choosing the memory bounds was applied for the other benchmarks as well to obtain minimum and maximum overlay sizes.

IBM Cell Implementation

The Cell architecture, as mentioned before, is comprised of eight SPUs, each having a local store of 256KB. For the experiments Sony PlayStation 3(PS3) is used which limits the use of SPUs. Of these eight available resources, six SPUs can be used due to PS3 hypervisor. User-defined overlay mapping is provided for each run. The SPU complier, *spu-gcc* maps the source code onto the LS according to the provided overlay scheme [2]. The overlay mapping file (*linker.script*), has function assignments to segments and segment assignments to regions as discussed in Section 4.5. Code objects must be stated in the script as individual object files. Accordingly, in order to implement and evaluate any code overlay mappings at the function level resolution, due to GNU Linker limitation, each function must be placed in a separate source file, so that post-compilation, each function in the source code has an object file for the linker.

The experimental runs for all three benchmarks were performed using the following steps:

1. Linker scripts describing overlay solutions for each algorithm, at each specified memory size are generated.
2. Each benchmark is compiled for a given linker script, without performing pre-fetch operation for any function. It is then run on IBM Cell to produce overlay results without pre-fetch.
3. The same linker script, with the four different pre-fetch prediction approaches, as explained in Section 5.5, is compiled again and run on IBM Cell for overlay results

with pre-fetch calls. The code automatically inserts the pre-fetch calls and executes the benchmarks.

4. Overlay misses are obtained by using miss count model explained in Section 5.5. This is used to calculate code overlay overhead.

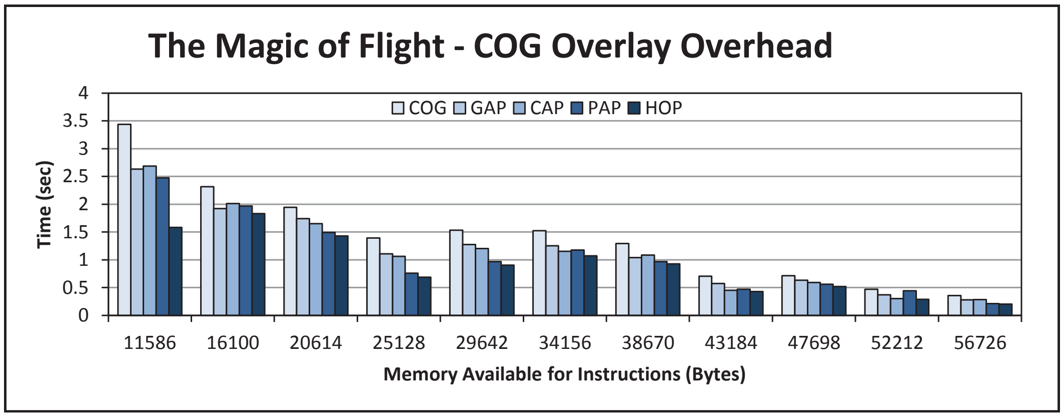
These benchmarks are executed multiple times and the resulting overhead data is averaged to obtain the results presented below. The *cjpeg* and *gsm_untoast* benchmarks are run for the same input data, whereas *ffmpeg* is run for three different videos. The choice of *ffmpeg* for additonal testing is owing to its properties of large code and data size, and is also very input sensitive.

6.2 Implementation Results

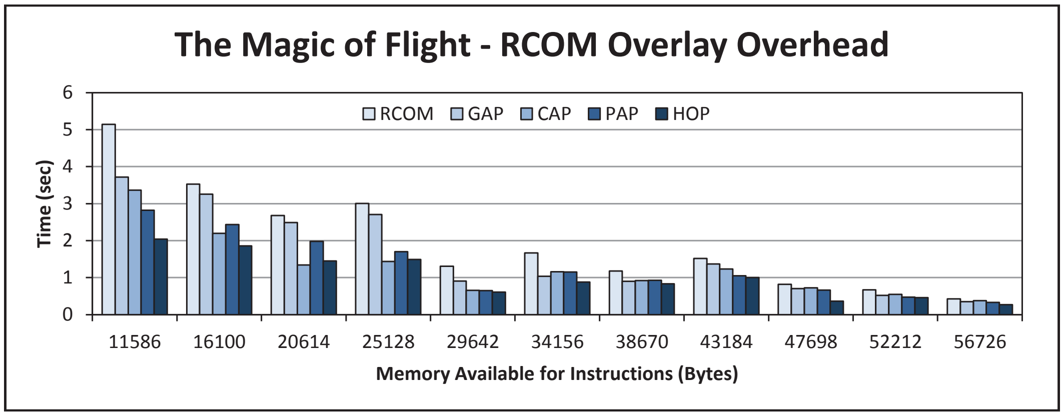
The implementation results discussed in this section compare the reduction in code overlay overhead by calculating the overlay overhead for the different pre-fetch methodologies considered as described in Section 5.5:

- *GAP (Gprof data Analyzed Pre-fetch)* - performs gprof function call count analysis.
- *CAP (Count Analyzed Pre-fetch)* - uses gprof function call counts along with the DMA overhead cost.
- *PAP (Profile overlay manager Analyzed Pre-fetch)* - uses function miss counts, caused due to the selected overlay mapping scheme, in addition to the DMA overhead cost.
- *HOP (Hand Optimized Pre-fetch)* - optimized version manually created by the end user.

Code overlay scheme with minimum code overlay size has only one region due to which it does not give any reduction in code overlay overhead cost. Overhead is maximum in this case, as each function call is miss. Result graphs does not include maximum overlay



(a) COG - Code Overlay Overhead Cost



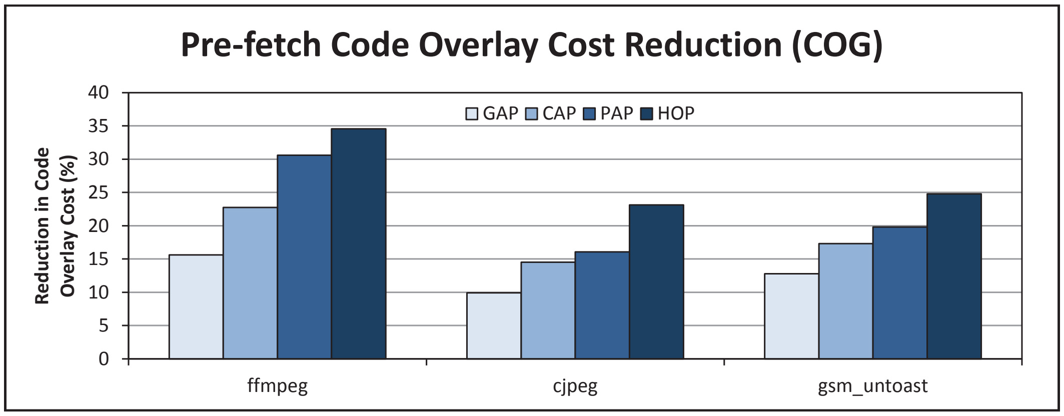
(b) RCOM - Code Overlay Overhead Cost

Figure 6.1: Effect of Available Memory on Code Overlay Overhead Cost for COG and RCOM algorithms on *ffmpeg* using The Magic of Flight video

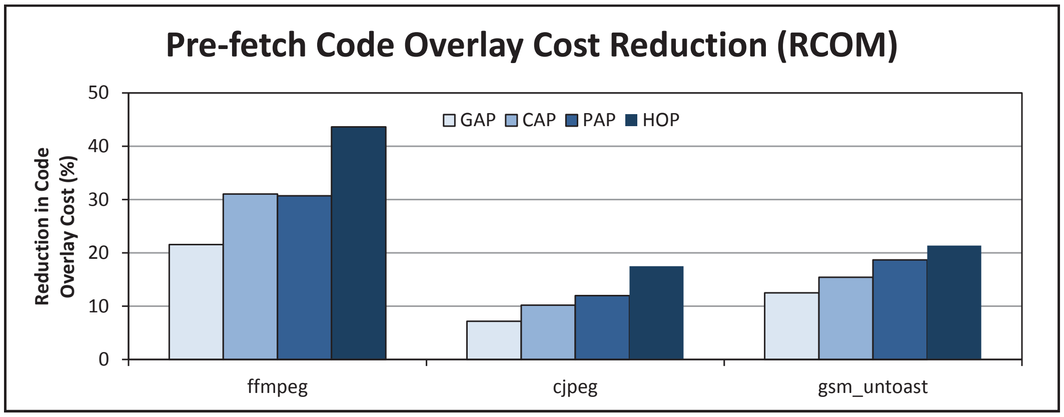
size possible, because for higher memory range overall overlay overhead is reduced significantly. Comparison excludes such minimum overlay size and some of larger overlay size readings.

Effect of Available Memory on Code Overlay Overhead Cost

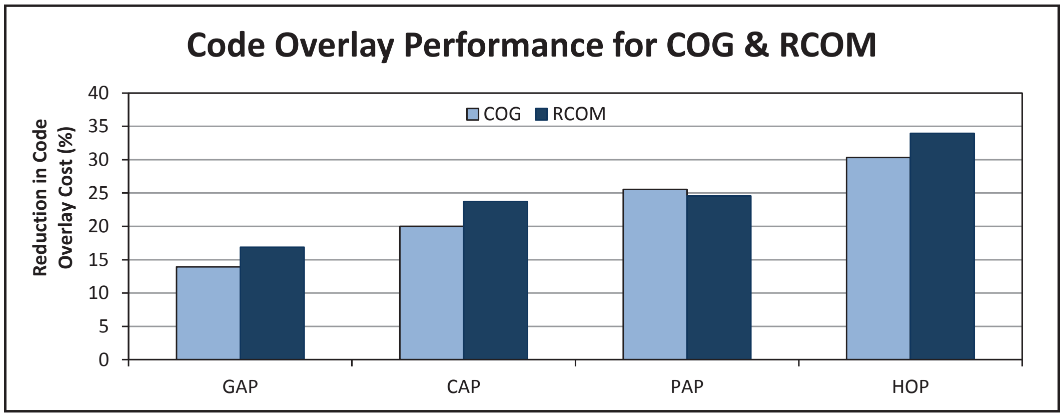
Figures 6.1a and 6.1b show the effect of available instruction memory on code overlay overhead cost for *ffmpeg*. As instruction memory increases, the overhead cost decreases. It also indicates for a particular memory size, overhead time is decreasing from GAP to HOP appraoch. Overhead cost decreases, as memory size increases because generated overlay mapping schemes have more regions to assign the functions. As functions are distributed



(a) COG Algorithm Overall Analysis



(b) RCOM Algorithm Overall Analysis



(c) COG & RCOM Overall Algorithm Analysis

Figure 6.2: Code Overlay Overhead Cost reduction for COG and RCOM algorithms on all three benchmarks.

in different regions, overlay misses are reduced. Smallest memory available for overlay mapping shows no overlay overhead reduction as it has only one region with all functions mapped to that region. The overlay overhead is maximum in this case as each function call is an overlay miss. For higher memory sizes the overhead becomes negligible, becoming zero if complete code is mapped to the memory.

Pre-fetch Code Overlay Performance Comparison for COG and RCOM

Figure 6.2a presents the code overlay performance analysis results for the COG algorithm, for all four pre-fetch prediction approaches. Similarly, Figure 6.2b shows the implementation results for the RCOM algorithm. Both plots show reduction in code overlay overhead cost for the benchmarks. In each plot, the HOP performance is found to be consistently better than the other three pre-fetch prediction approaches. The code overlay performance is intuitively expected from HOP as it is a fully hand optimized version, where the user has performed adequate profiling before finalizing all the pre-fetch calls. Additionally, the code overlay performance gain is found to be the highest in case of the *ffmpeg* benchmark.

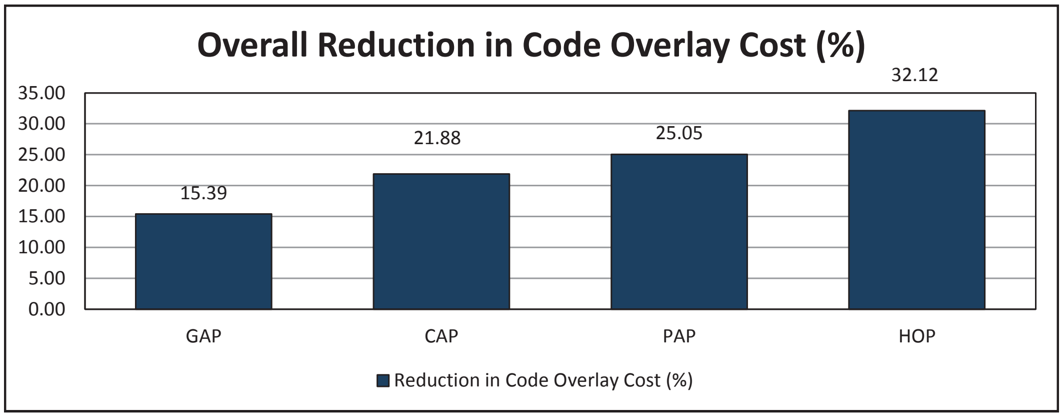
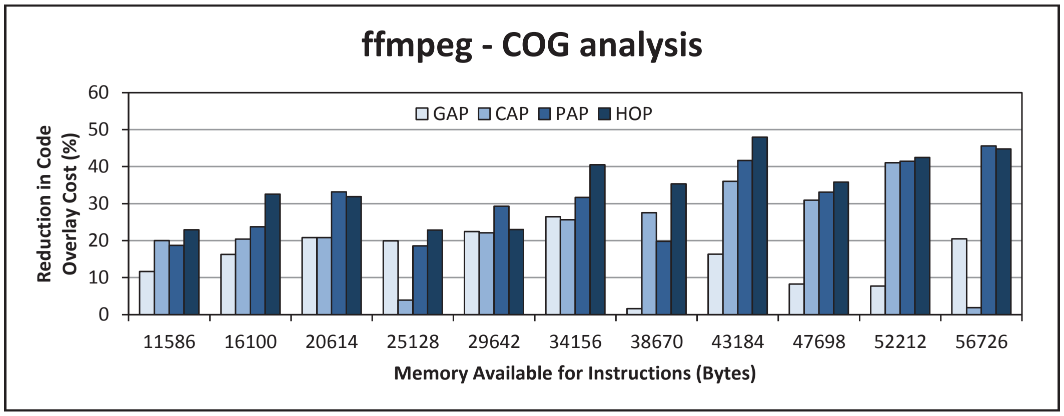
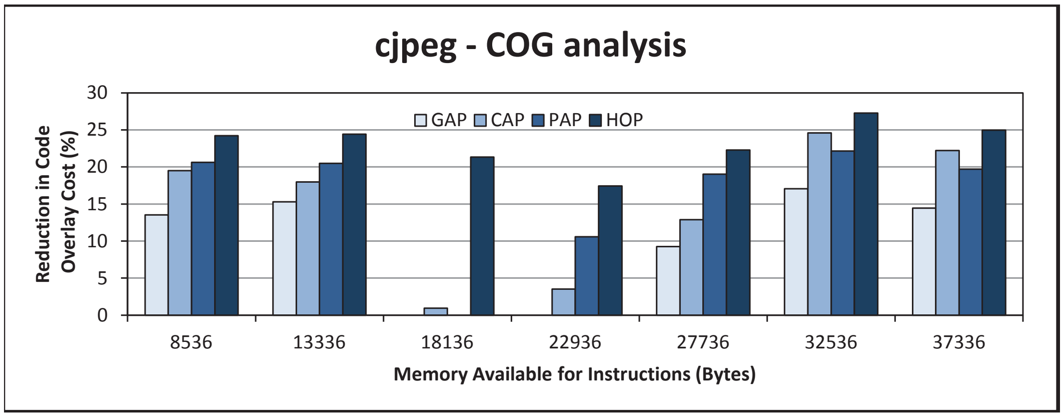


Figure 6.3: Code Overlay Overhead Cost Reduction

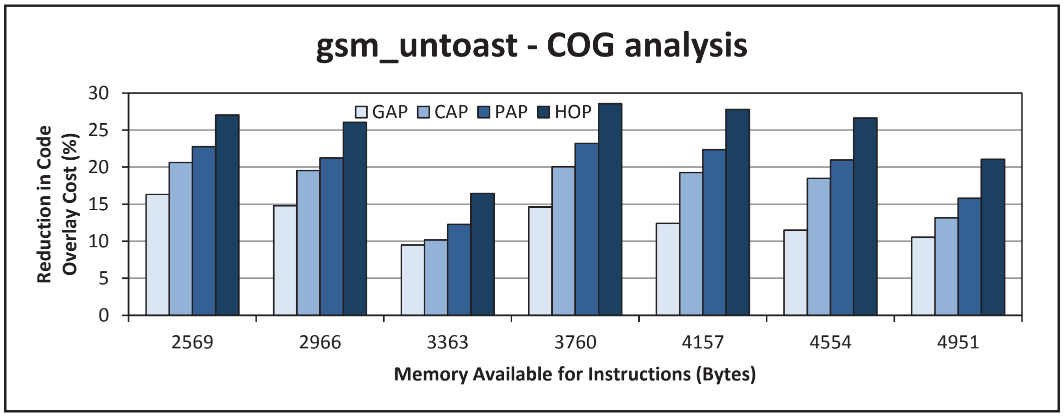
Figure 6.2c compares the implementation results for the COG and RCOM algorithms. It is evident from the plots in Figures 6.2a, 6.2b and 6.2c that the reduction in overlay overhead cost is better for the RCOM algorithm for all the prediction approaches, for all benchmarks, except PAP approach. This is because RCOM mapping has more scope



(a) COG Algorithm ffmpeg Analysis



(b) COG Algorithm cjpeg Analysis



(c) COG Algorithm gsm_untoast Analysis

Figure 6.4: Code Overlay Overhead Cost reduction for COG algorithm using four pre-fetch prediction appraoches on (a) *ffmpeg*, (b) *cjpeg* and (c) *gsm_untoast*.

for pre-fetch calls as compared to the optimized COG overlay mapping.

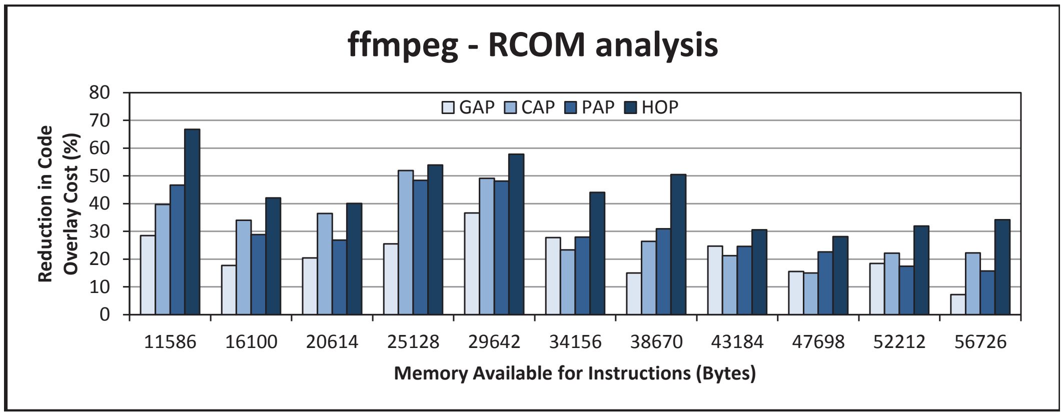
Figure 6.3 summarizes the overall code overlay performance for all four approaches. When compared to overlay without pre-fetch, it is apparent that the HOP approach gives the best results followed by PAP, CAP and GAP. A minimum reduction of overhead cost of 15% for GAP and a maximum of 32% for HOP is observed.

COG Analysis for Benchmarks

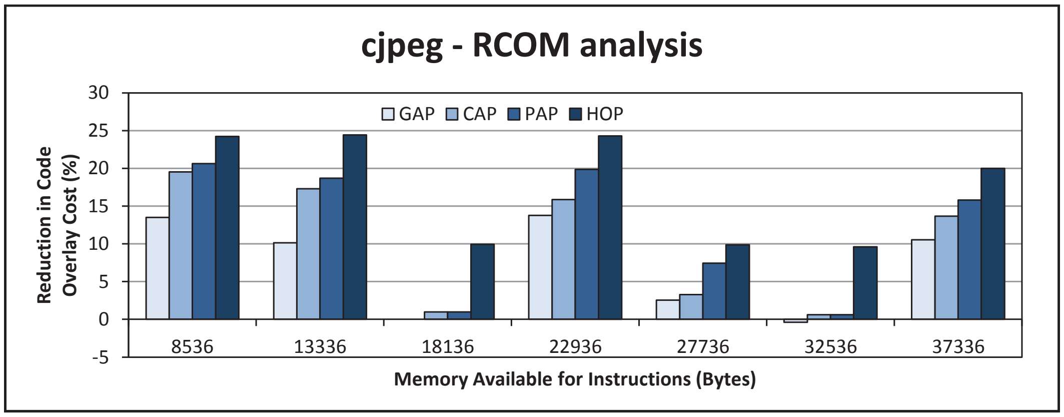
COG performance anlaysis plots are shown in the Figures 6.4a, 6.4b and 6.4c for *ffmpeg*, *cjpeg* and *gsm_untoast* respectively. In these graphs, code overlay overhead reduction cost is plotted as a function of instruction memory in bytes. For *ffmpeg*, the trend begins with lower reduction in overhead for lower memory sizes. This is because of the limited number of regions available, which in turn limits the pre-fetch calls that can be made. As memory size increases, the number of regions increase, which gives more option to add pre-fetch calls for all other regions available. Hence, as seen from the plots, the reduction in overhead cost improves. However, as memory size increases further, for very large memory, the performance gain begins to saturate. For other two benchmarks shown in Figures 6.4b and 6.4c HOP performs better than all other three pre-fetch approaches as expected.

RCOM Analysis for Benchmarks

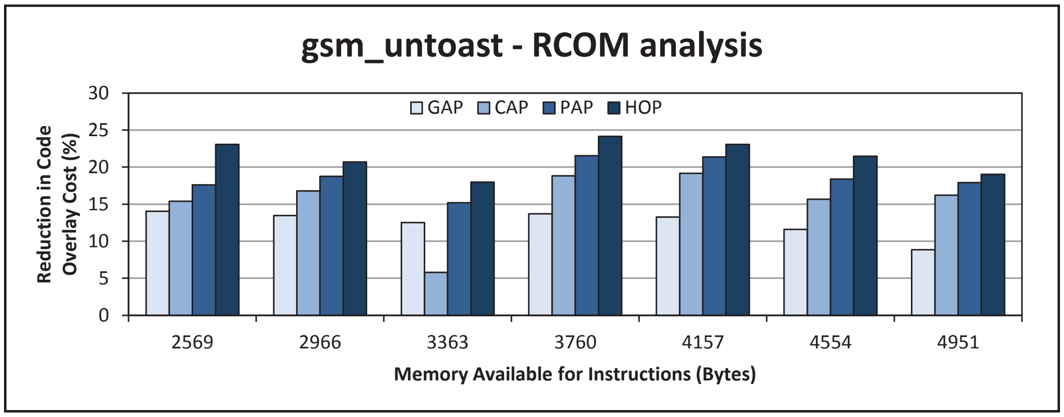
RCOM performance analysis results are presented in the Figures 6.5a, 6.5b and 6.5c for all the benchmarks. Here, the reduction in overlay cost is again plotted as a function of instruction memory in bytes. RCOM algorithm gains are relatively higher as compared to COG. The rationale behind this is that the highly optimized COG overlays leave very little scope for further improvement. Figure 6.5a shows increase in reduction in overhead cost with increasing memory size, until the memory size becomes large enough so that further overhead reduction enhancement is difficult to achieve. Figures 6.5b and 6.5c varifies that HOP performs better than GAP, CAP and PAP.



(a) RCOM Algorithm ffmpeg Analysis

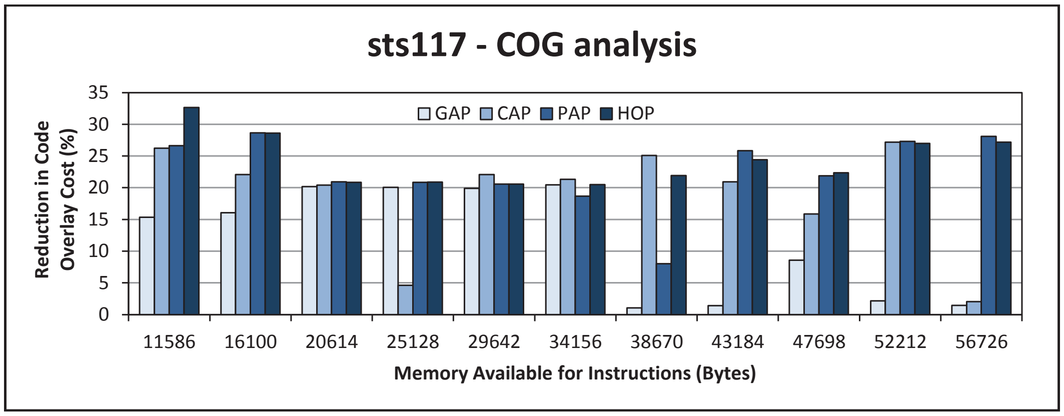


(b) RCOM Algorithm cjpeg Analysis

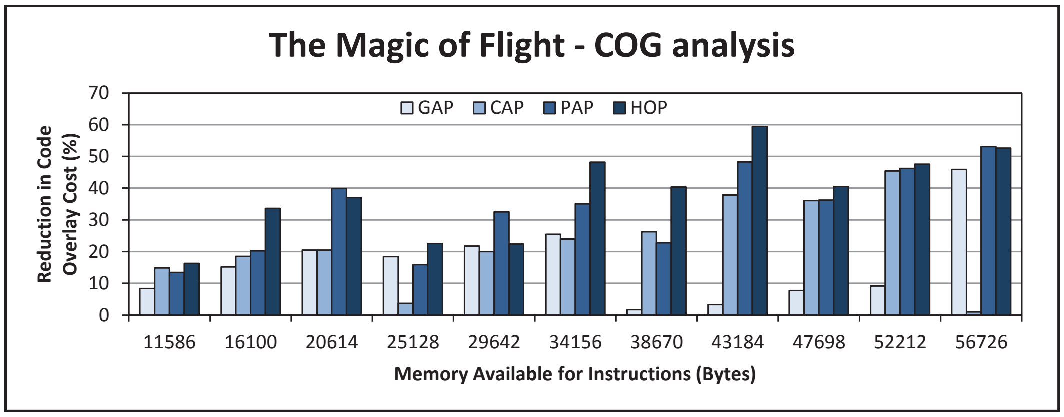


(c) RCOM Algorithm gsm_untoast Analysis

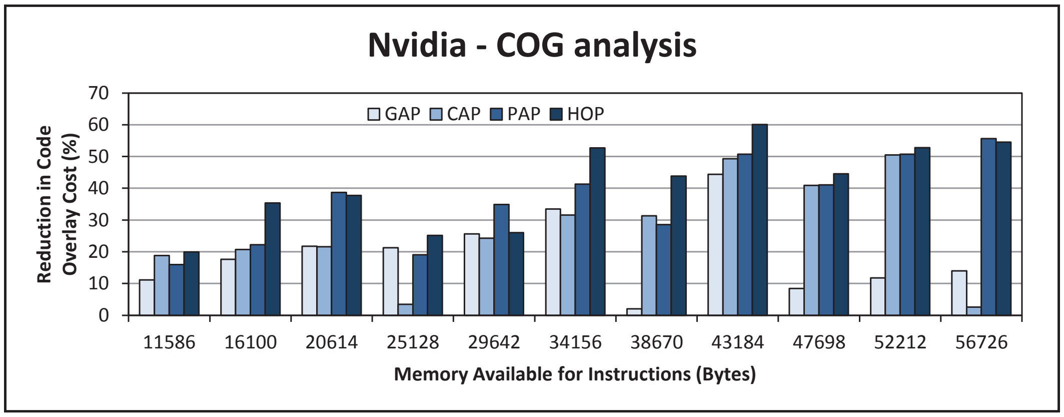
Figure 6.5: Code Overlay Overhead Cost reduction for RCOM algorithm using four pre-fetch prediction appraoches on (a) *ffmpeg*, (b) *cjpeg* and (c) *gsm_untoast*.



(a) sts117 video - COG



(b) The Magic of Flight Video - COG



(c) Nvidia video - COG

Figure 6.6: Code Overlay Overhead Cost reduction for COG algorithm using four pre-fetch prediction appraoches on (a) *sts117*, (b) *The Magic of Flight* and (c) *Nvidia* videos.

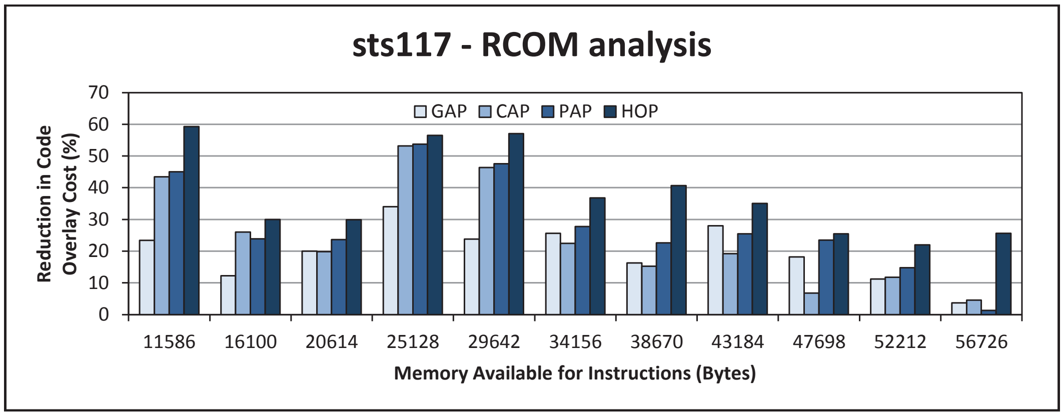
Analysis of ffmpeg for Different Videos

The *ffmpeg* code was used to carry out additional experimental runs. The reasons for choosing *ffmpeg* code for more analysis include large code size, greater number of functions, higher data size (heap/stack) and it is also highly sensitive to input data. These experiments were carried out using three different videos - sts117 (space shuttle launch video), Nvidia and The Magic of Flight. Figures 6.6a to 6.6c and 6.7a to 6.7c depict the reduction in code overlay overhead plotted with respect to the available instruction memory for these three videos. Selection of these three video variants gives wider range of data to analyze the code overlay overhead reduction due to pre-fetch. These videos show different coding characteristics due to motion vector and inter/intra prediction used for encoding/decoding, since each video contents are different, and consequently show different behavior in the analysis.

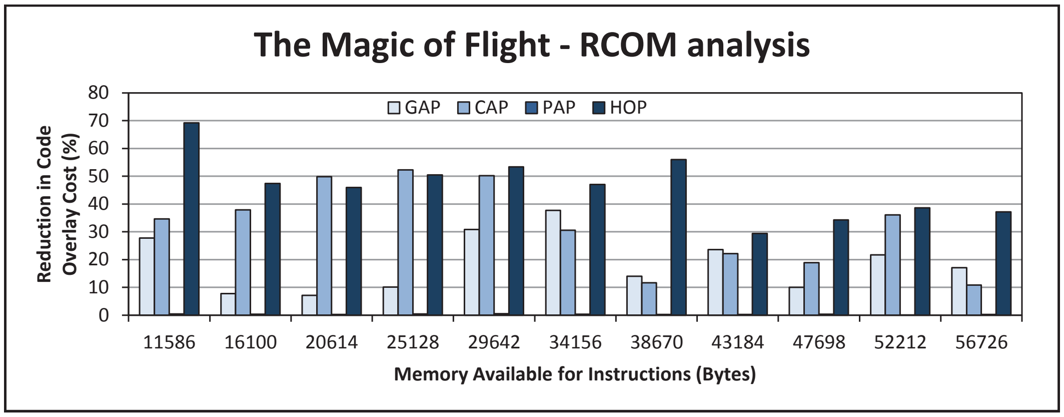
Figure 6.6a shows limited performance gain due to highly optimized COG algorithm as compare to Figure 6.7a which is RCOM algorithm. Figures 6.6b and 6.6c show more gain due to different function calls to decode the video. Similar trends can be seen for RCOM algorithm plots as shown in Figures 6.7a, 6.7b and 6.7c. Less reduction in overhead cost for sts117 video by using pre-fetch suggests that most of the decoding involves function calls which does not cause high numbers of overlay misses, left with fewer options for pre-fetch calls. For same overlay scheme, sts117 shows less reduction in overhead compare to Nvidia and The Magic of Flight because of sts117 video might be encoded with limited modes of inter prediction. Large range of modes of inter prediction for input data causes different function calls which might result into overlay misses according to overlay scheme. This overlay misses are avoided by using pre-fetch for Nvidia and The Magic of Flight.

Framewise Analysis for Different Videos

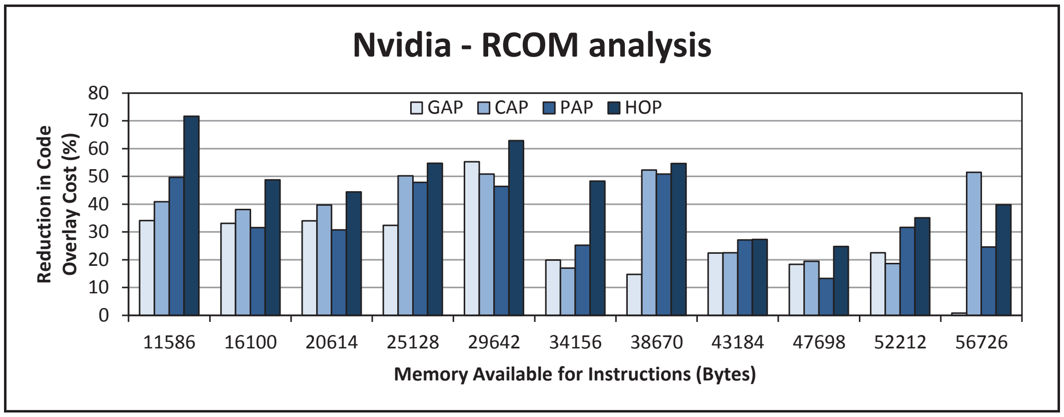
The *ffmpeg* code is executed for sts117 and Nvidia video for obtaining code overlay overhead at different number of frames decode run. The Figure 6.8 show the frame level analysis for code overlay overhead for memory sizes of 11586B and 34156B for different pre-fetch



(a) sts117 video - RCOM

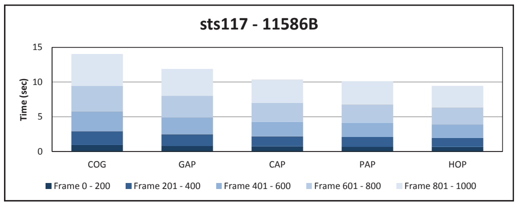


(b) The Magic of Flight Video - RCOM

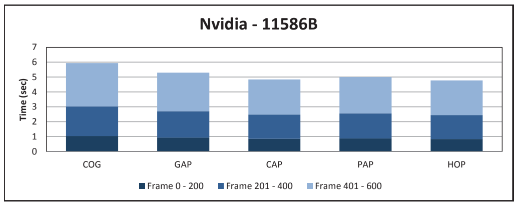


(c) Nvidia video - RCOM

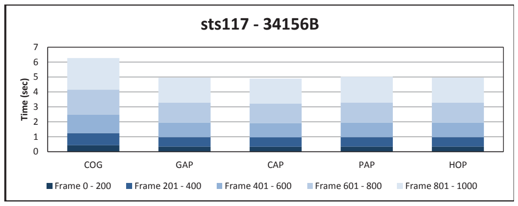
Figure 6.7: Code Overlay Overhead Cost reduction for RCOM algorithm using four prefetch prediction appraoches on (a) *sts117*, (b) *The Magic of Flight* and (c) *Nvidia* videos.



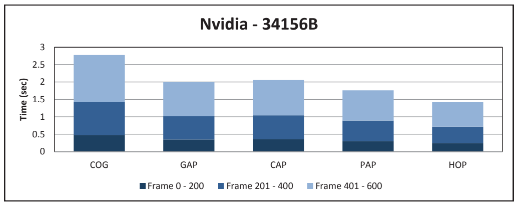
(a) sts117 video for 11586B



(b) Nvidia video for 11586B



(c) sts117 video for 34156B



(d) Nvidia video for 34156B

Figure 6.8: Frame level Analysis

approaches. The plots indicate that the code overlay overhead time goes on decreasing as memory size increases from 11586B to 34156B for different frame numbers. The graphs plotted in Figures 6.8a and 6.8c should ideally have linear distribution of time over the frame range, but the sts117 video does not follow this trend. The sts117 video has static image for initial few frames due to which it decodes the subsequent image with the help of initial I - frame using the same prediction blocks. This reduces the different function calls results into fewer overlay misses. As video progresses, it has continuously changing images which are possibly encoded with different modes of inter prediction for compression. To decode such frames with diversified inter prediction modes, varies from macroblok to macroblock and frame to frame, requires different function calls. This leads to overlay misses. This might be the reason for having more overahed in later frames of sts117 video. The Nvidia video has equal distribution of the overlay time over the frame range as shown in the Figures 6.8b and 6.8d. This can be justified by the same argument of the video contents and the encoded technique.

Chapter 7

CONCLUSION & FUTURE WORK

Code overlay performance for size-limited SPMs used in embedded systems was experimentally verified on the IBM Cell. The proposed pre-fetch mechanism for memory-restricted SPM code overlay is found to perform significantly better on COG, which is an existing SPM mapping algorithm and the newly presented RCOM algorithm. The proposed pre-fetch prediction techniques show a performance improvement of 32% in case of HOP, 24% for PAP, 22% for CAP and 15% in case of GAP for given code overlay mapping. The performance gain numbers for all these techniques support the argument that hand optimized approach is better over the *gprof* raw data analysis. The pre-fetch on any overlay mapping scheme gives better results as compared to the IBM Cell Broadband Engine compiler *spu-gcc*. *Spu-gcc* is not capable of using pre-fetch technique because it has only one overlay region. So, it has no scope for pre-fetching mechanism to be embedded in it. The overlay miss count modeling used in this work can be incorporated as a profiling tool with *spu-gcc* to get real-time overlay misses for any application. A detailed analysis of misses along with overlay overhead cost can also be achieved by using such a profiling tool. This information can further be used for performing extensive study on overlay mapping schemes.

All the proposed pre-fetch prediction techniques for code overlay are automated except for HOP, where user needs to provide the pre-fetch calls. The pre-fetch insertion mechanism is also automated which inserts pre-fetch calls according to the predictions, compiles the code, executes it and reverts the additions made to the code. Potential extensions to this work can be dynamically predicting pre-fetch calls for individual functions. To enable this feature, the overlay manager needs to be modified such that it dynamically predicts the next suitable pre-fetch calls, by maintaing an active history of previous function calls made. This feature would be similar to the branch prediction algorithms does in computer architecture. The developed tool chain for overlay miss count modeling along with estimation of overlay overhead can be made available for research purposes.

REFERENCES

[1] *Cell Broadband Engine Architecture*. IBM Systems and Technology Group., 2007.

[2] *Software Development Kit for Multicore Acceleration Version 3.1 Programmer's Guide*. IBM Systems and Technology Group., 2008.

[3] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '03, pages 318–326, New York, NY, USA, 2003. ACM.

[4] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. 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*, CASES '04, pages 259–267, New York, NY, USA, 2004. ACM.

[5] M. A. Baker, P. Dalale, K. S. Chatha, and S. B. Vrudhula. A scalable parallel h.264 decoder on the cell broadband engine architecture. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, pages 353–362, New York, NY, USA, 2009. ACM.

[6] M. A. Baker, A. Panda, N. Ghadge, A. Kadne, and K. S. Chatha. A performance model and code overlay generator for scratchpad enhanced embedded processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 287–296, New York, NY, USA, 2010. ACM.

[7] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[8] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1:521–540, December 2005.

[9] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 223–233, New York, NY, USA, 2006. ACM.

[10] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In

Proceedings of the 41st annual Design Automation Conference, DAC '04, pages 238–243, New York, NY, USA, 2004. ACM.

[11] http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html.

[12] A. Janapsatya, A. Ignjatović, and S. Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 612–617, Piscataway, NJ, USA, 2006. IEEE Press.

[13] S. c. Jung, A. Shrivastava, and K. Bai. Dynamic code mapping for limited local memory systems. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 13 –20, 2010.

[14] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 628–633, New York, NY, USA, 2002. ACM.

[15] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 690–695, New York, NY, USA, 2001. ACM.

[16] L. Li, H. Feng, and J. Xue. Compiler-directed scratchpad memory management via graph coloring. *ACM Trans. Archit. Code Optim.*, 6:9:1–9:17, October 2009.

[17] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 115–125, New York, NY, USA, 2005. ACM.

[18] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. Sdrm: simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Proceedings of the 15th international conference on High performance computing*, HiPC'08, pages 569–582, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] 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 '02, pages 213–218, New York, NY, USA, 2002. ACM.

[20] H. Takase, H. Tomiyama, and H. Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1124–1129,

3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[21] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '03, pages 276–286, New York, NY, USA, 2003. ACM.

[22] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: a first approach. *IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, 0:115–120, 2005.

[23] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, DATE '04, pages 21264–, Washington, DC, USA, 2004. IEEE Computer Society.

[24] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 600–605, Washington, DC, USA, 2005. IEEE Computer Society.

[25] S. Wuytack, F. Catthoor, L. Nachtergaele, and H. De Man. Power exploration for data dominated video applications. In *Proceedings of the 1996 international symposium on Low power electronics and design*, ISLPED '96, pages 359–364, Piscataway, NJ, USA, 1996. IEEE Press.