Automated Memory Allocation of Actor Code and Data Buffer in Heterochronous Dataflow Models to Scratchpad Memory

Shamik Bandyopadhyay

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-105
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-105.html

August 14, 2006

Automated Memory Allocation of Actor Code and Data Buffer in Heterochronous Dataflow Models to Scratchpad Memory

by Shamik Bandyopadhyay

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Edward A. Lee
Research Advisor

(Date)

* * * * * * *

Professor Alberto Sangiovanni-Vincentelli
Second Reader

(Date)

Abstract

A scratchpad memory is a fast compiler-managed on-chip memory that replaces the hardware managed cache in several embedded processors. Scratchpad memories provide better real-time guarantees, and significantly lower overheads in energy consumption and area as compared to caches [9]. Several allocation schemes exist for mapping variables, data and code to the scratchpad memory. However, none of these schemes seek to optimize the allocation based on the control flow and data flow information that can be gleaned from block diagram based programs in frameworks in LabVIEW and Ptolemy II [7].

In this report we present methods for scratchpad memory allocation of block diagram programs composed under the Heterochronous Dataflow (HDF) model of computation [3]. HDF is a heterogeneous composition of Synchronous Dataflow (SDF) [1] and Finite State Machines (FSM). It is significantly more expressive than SDF since it allows rate changes in actors during execution. The memory allocation method exploits the coarse grained software architecture and semantics of the HDF models to gather information about the temporal pattern and access frequencies of memory references and the memory requirements of the computational blocks. This information is processed using an Integer Linear Program (ILP) based framework to generate the optimal memory allocation for each state in the HDF graph. We also present a modified version of the algorithm which generates multiple allocations for each state.

The ILP based allocation algorithm is shown to perform significantly better than caches. We also show that the allocation algorithm increases predictability by being completely independent of the scheduling technique used to generate the execution

schedule for the HDF model. The algorithmic modification that generates multiple allocations per state is shown to perform better than the original algorithm for actor code allocation for intermediate scratchpad sizes. We present a strict upper bound on the memory access time of a HDF model allocated to scratchpad using the above algorithm. We also present several interesting directions for future exploration.

Acknowledgements

This work is the culmination of eighteen years of academic pursuit. Many more individuals have helped in this achievement than can be listed here, but a few people surely deserve a special mention.

I would firstly like to thank my research advisor, Prof. Edward Lee. Without his support, and guidance, this project would not have been possible. I really appreciate his encouragement and priceless advice that helped me make this project a success.

I would also like to thank Prof. Alberto Sangiovanni-Vincentelli for taking time out of his hectic schedule to serve as the second reader for this project. I am grateful for his thoughtful comments and suggestions.

I would like to thank all members of the Ptolemy research group for creating such a wonderful intellectual environment. I would like to specially thank Gang Zhou for helping me understand and use the Ptolemy software. The numerous discussions with Gang, and the suggestions provided by him have proved extremely valuable.

My love and appreciation goes towards my parents for their support, and guidance throughout my academic career. I am appreciative of my mother in law Chandana for providing a comforting and relaxing atmosphere in which to recharge. For her unwavering confidence, infinite patience, boundless encouragement and her cheerful comforting nature, my wife, Priyanka deserves more credit for this work than anyone else, myself included. Over much of the past decade, in her roles as girlfriend, fiancée and wife, Priyanka has provided perpetual support in both my personal and academic life. Without her, this work would not have been possible. I dedicate this report to my wife, Priyanka.

Contents

1. Introduction

Writing embedded systems software has long been considered as a detail oriented low level task involving the use of assembly language or C. It has been up to the programmer to ensure proper memory allocation, resource management, robustness, deadline guarantees, reactivity and concurrency. In recent years, graphical modeling and simulation systems like Ptolemy II from University of California, Berkley [7] and LabVIEW from National Instruments have provided the ability, among others, to use various models of computation for developing embedded applications at a higher level of abstraction, i.e. the block diagram level. Such systems have served as an experimental testbed to explore various scheduling techniques, liveness and robustness issues, concurrency, timing constraints, and resource management concerns. These systems also provide the mechanism to automatically generate embedded systems code for various applications designed at a block diagram level. The generated code is often comparable to hand optimized embedded code. This is because block diagrams provide a wealth of information about the structure of a program, its execution flow and its control flow. Algorithms can be developed that use this information to perform automatic optimizations that improve the quality and performance of the generated code. These algorithms can thus make optimal use of numerous special hardware features found in modern embedded processors.

This report explores ways to use the information found at a block diagram level to find the best utilization for a particularly efficient hardware structure found in most embedded processors – the scratchpad memory (SPM). It describes a memory allocation technique that aims at making the best possible use of this highly efficient memory

structure for models described using the Heterochronous Dataflow (HDF) model of computation. The method exploits the coarse grained software architecture and semantics of the heterochronous dataflow (HDF) models in gathering information about the temporal pattern and access frequencies of memory references and the memory requirements of the computational blocks. The technique then uses the gathered information to make a high level coarse granularity memory allocation for the actor code and buffer memory of the HDF model to the scratchpad memory.

The following gives an outline of the report. The next sections of this chapter briefly introduce the Synchronous Dataflow (SDF) and Finite State Machine (FSM) models of computation, which constitute HDF. The Heterochronous Dataflow (HDF) is described in the following section. The second chapter further motivates the importance of scratchpad memory (SPM) by providing a detailed description, identifying the benefits and making a comparison with the more traditional cache architecture. Chapter 3 describes the memory architecture chosen for study, and states the basic assumptions and observations made regarding the memory access profiles of HDF models. Chapter 4 presents a dynamic allocation scheme for allocation of memory elements to the SPM. It also presents a modification to the original scheme which aims to exploit the execution flow of the program in making allocation decisions. In Chapter 5 the performance analysis of the allocation algorithm and experimental results are presented. Chapter 6 concludes the report and outlines directions of future exploration.

1.1 Synchronous Dataflow (SDF)

Dataflow models of computation [4] represent concurrent programs through interconnected blocks called *actors*. Each such block represents a function or a set of

functions that maps the inputs to the outputs of the actor. Actors receive data on their *input ports* and produce data on their *output ports*. The basic behavior of an actor is to perform its specified computation upon every invocation or *firing*, and communicate *data tokens* with other actors over the interconnecting *channels*.

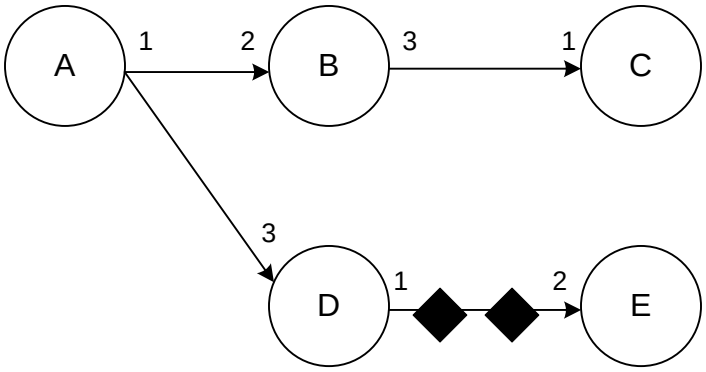

Figure 1.1 A SDF model shown as a directed graph

Synchronous Dataflow (SDF) [1], is a dataflow model of computation where actors communicate through FIFO queues and the number of data tokens produced and consumed by each actor on each invocation is specified *a priori*. The number of tokens produced and consumed by an actor on each firing is known as the *production rate* and the *consumption rate* respectively, and together they give the *rate signature* of an actor. Thus a SDF model can be represented as a directed graph ($G = (V,E)$), where each node in the set V corresponds to an actor and each edge in the set E corresponds to a FIFO communication buffer. The directed graph model also clearly outlines the primary high level memory requirements of a SDF model which consists of the code memory for the actor code and the data memory for the communication buffers. It should be noted that in actor-oriented design each actor can be hierarchically refined to any depth. Figure 1.1 shows an acyclic SDF graph and Figure 1.2 shows the model represented as a Ptolemy block diagram.

Each communication buffer can contain initial tokens or *delays*. The number of initial tokens on an edge is equal to the *initial production rate* of the source actor for that edge. Figure 1.1 and 1.2 show two units of delay on the buffer connecting actors D and E.

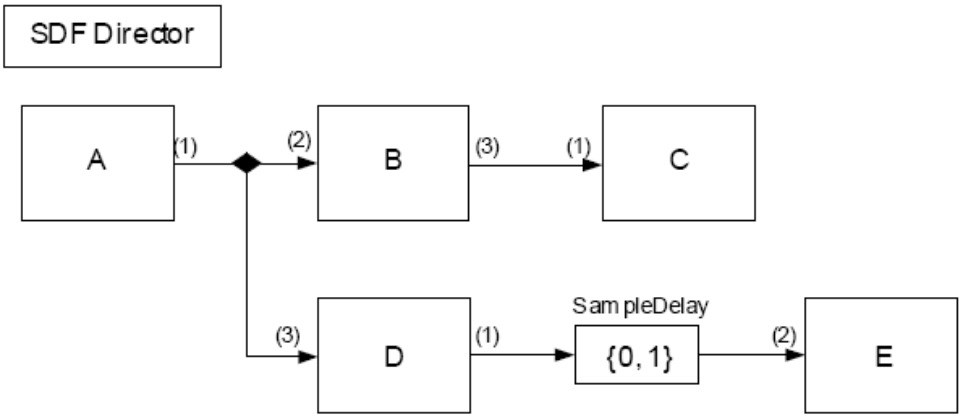

Figure 1.2 A SDF model shown as a Ptolemy block diagram [11]

An *iteration* of an SDF model is the set of minimal firings of the actors that returns the FIFO buffers to their original sizes. A *periodic admissible schedule* or *valid schedule* for a SDF model is a sequence of actor firings that fires each actor a_i, f_i number of times, does not deadlock and does not produce any net change in the number of tokens present on the edges [2]. The set of firings f_i for each actor a_i in a schedule is called the *firing vector* and is computed using balance equations on each of the edges of the SDF graph. The balance equations essentially state that the number of tokens produced by the source actor on an edge is equal to the number of tokens consumed by the destination actor. The balance equation for an edge connecting actor A to actor B is defined as:

$$r_A f_A = r_B f_B$$

where r_A and r_B are the production and consumption rates of A and B respectively, and f_A and f_B are the numbers of firings for actor A and B respectively. The balance equations are solved over all edges in the graph in order to compute the firing vector.

1.2 Finite State Machines (FSM)

A finite state machine (FSM) is a model of computation composed of *states* and *transitions* between states. The basic behavior of a finite state machine is to start execution in a particular *initial state*, and then to transition from one state to another based on the values of its inputs. Each transition is controlled by a *guard condition*, which is generally a Boolean function of the inputs to the state machine. If the guard condition evaluates to *true*, the guard is said to be enabled. If no transition is enabled, a transition back to the current state is implied. Associated with each transition, there may be *actions*, which cause parameters to be set or outputs to be produced. Figure 1.3 shows a simple *state transition diagram* for a FSM. The FSM starts in state A and transitions to state B when input *P* is greater than 0. When P becomes less than or equal to 0, the FSM transitions back to state A.

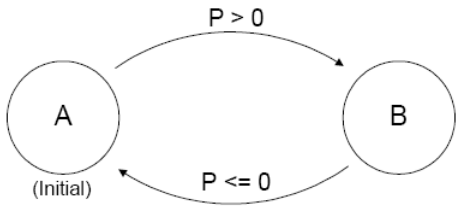

Figure 1.3 State Transition Diagram for a simple FSM [11]

It should be noted that FSM models can be constructed hierarchically. Each state in a FSM can be further refined to an arbitrary depth of hierarchy.

1.3 Heterochronous Dataflow (HDF)

SDF is a very robust and well studied model of computation that guarantees bounded memory usage, bounded schedule length, and deadlock-free execution. However, one key limitation of SDF models is that the rate signatures of actors are fixed

and must be defined *a priori.* For this reason, SDF models prove unsuitable for implementing control protocols and adaptive algorithms that require dataflow rate variation during execution [11]. Heterochronous Dataflow (HDF) is a model of computation that significantly increases the expressiveness of SDF and makes it ideal for implementing variable rate models.

The Heterochronous Dataflow (HDF) model of computation was originally introduced by Girault *et al* in [3]. In simple terms, HDF is a heterogeneous composition of the FSM and SDF models of computation, in which the changes in state correspond to the changes in dataflow rates.

An HDF model is a hierarchical composition of SDF and FSM. An actor in HDF has a finite number of rate signatures, where each rate signature specifies the number of tokens produced and consumed in one firing [3]. An actor which composes other actors, from the same or different domains, is called a *composite actor*. A composite actor in HDF is composed of a FSM, whose individual states further refine into SDF or HDF models. The current state of the FSM determines the current SDF or HDF refinement of the particular actor. The local schedule for the SDF or HDF refinement determines the current rate signature of the composite actor. Refinements in states other than the current state are thus disconnected from the system.

Each *state of HDF* is identified by a different set of connections between the dataflow graphs in the hierarchy, i.e. a unique combination of the current states of the constituent actors. The state of the HDF model specifies particular rate signatures for its constituent actors, which can be used to solve the balance equations and compute the global schedule for that state. Hence each state of the HDF model corresponds to a different schedule for the system. The *initial state* of a HDF model is the state in which

the model starts execution. A key point to note is that changes in rate signatures or state changes are restricted to occur only at the end of a *global iteration* of the whole system. Hence rate signatures stay constant for the duration of a global iteration. Figure 1.4 [3] shows a simple HDF model.

In the example in Figure 1.4, the actor B has two possible states B_1 and B_2. The states B_1 and B_2, refine into simple SDF models. Thus there are two possible states, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ for the HDF model itself, where:

$\boldsymbol{\alpha} = AB_1C$

$\boldsymbol{\beta} = AB_2C$

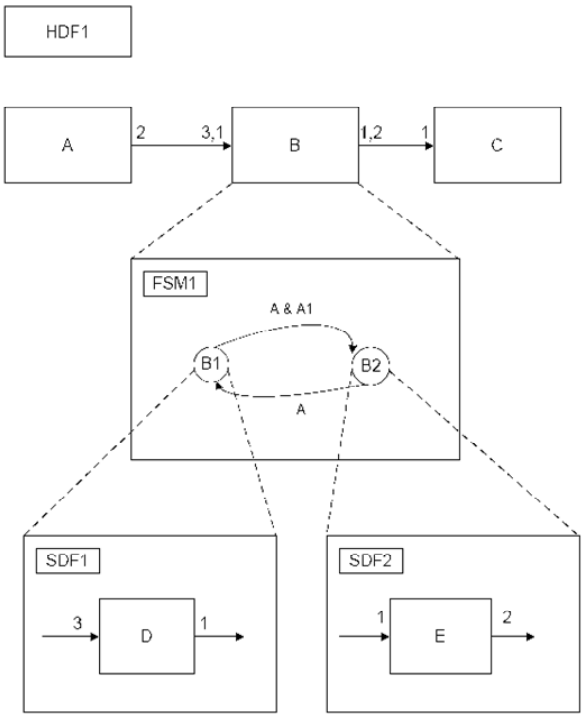

Figure 1.4 A simple HDF model - *adapted from* **[3]**

In state $\boldsymbol{\alpha}$**,** B consumes three tokens and produces one. This leads to the schedule **{A, A, A, B, B, C, C}** for state $\boldsymbol{\alpha}$. In state $\boldsymbol{\beta}$, B consumes one and produces 2, leading to the schedule **{A, B, B, C, C, C, C}.**

It is to be noted that HDF models can contain multiple composite actors at the same level and can have an arbitrary depth of hierarchy. In fact, a state of an FSM in the HDF model can further refine to another HDF model. Figure 1.5, from [11], shows such an example which has an initial state $A_{11}B_1$ and the set of states:

$$S = \{ A_{11}B_1 , A_{12}B_1 , A_{11}B_2 , A_{12}B_2 , A_2B_1 , A_2B_2 \}$$

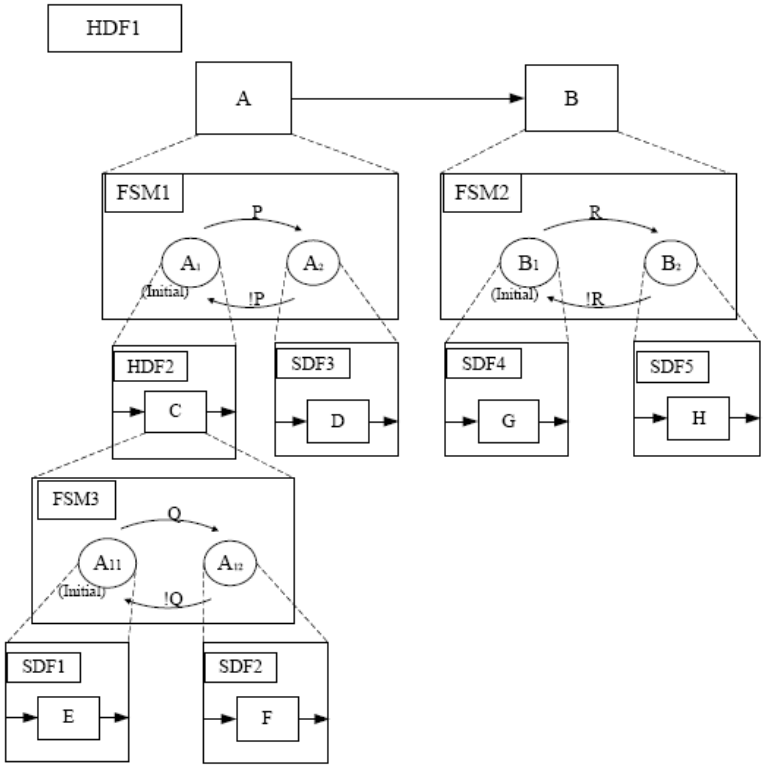

Figure 1.5 A hierarchical HDF model [11]

There are two alternatives to computing the schedule for an HDF model. It is theoretically possible to pre-compute all schedules for all possible states and iterations. However, this might be impractical for large HDF models, since the number of rate signature combinations is exponential in the number of HDF nodes. The second alternative would be to dynamically compute the schedules between iterations. It should be noted however, that the current code generation framework for HDF models in

Ptolemy uses pre-computed schedules for the generated code. A key reason to opt for this alternative is to avoid the code overhead and time penalty of running the scheduling algorithm in the generated embedded code itself. In this chapter we provided a brief introduction to SDF, FSM and HDF models of computation. The next chapter provides an introduction to the scratchpad memory, its benefits, drawbacks and intricacies.

2. Memory Hierarchy – Caches and Scratchpad Memory

A modern microprocessor executes instructions at a very high rate. To exploit its processing power fully, the processor must be supported by a memory system that is equally fast. Unfortunately, the rate at which processing power is increasing is much greater than the rate at which memory performance is increasing. This, often referred to as the processor-memory performance gap, is shown in Figure 2.1

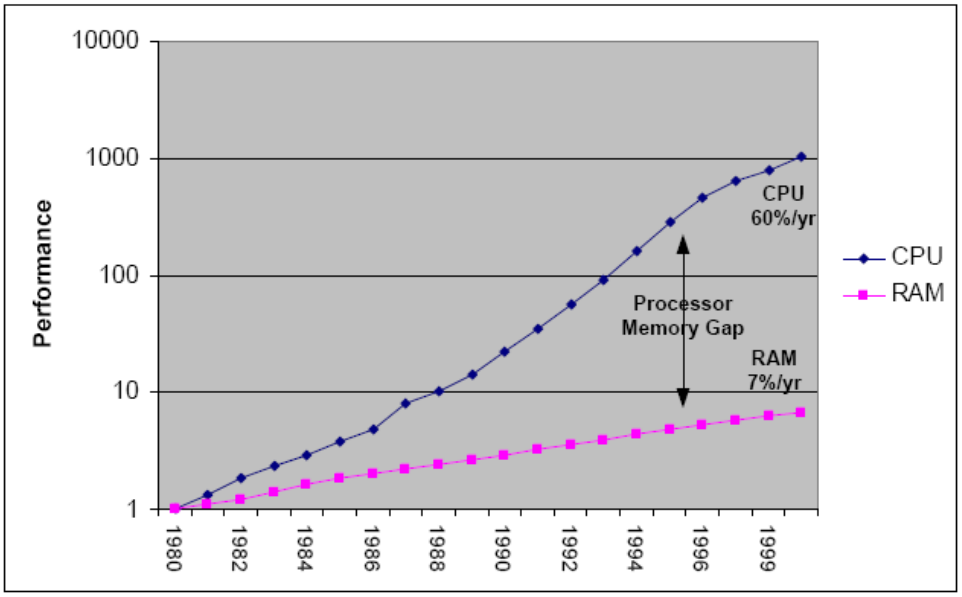

Figure 2.1. The Processor-Memory Performance Gap is grows by 50% per year - *adapted from* **[25]**

In an ideal case, a processor would require unlimited amounts of fast memory to meet its processing requirements. However, memory speed varies inversely with memory size or capacity, and faster memory is also more expensive. An economical solution to this problem and a way to bridge the processor-memory performance gap is to use multiple memories in a *memory hierarchy*. A memory hierarchy uses smaller and faster memory technologies close to the processor. Thus accesses that hit the highest level of hierarchy can be processed quickly. Accesses that miss go to lower levels of the

hierarchy, which are slower but larger and cheaper. If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest level and a size equal to that of the lowest level. Memory hierarchy thus takes advantage of the *principle of locality* [8], which states that an average computer program at any particular time tends to execute the same instructions and access the same blocks of data repeatedly. In order to fully exploit the memory hierarchy and locality of memory references, the highest levels of memory must attempt to store the most frequently accessed subset of memory references. Figure 2.2, adapted from [8], shows a typical memory hierarchy for embedded and desktop processors.

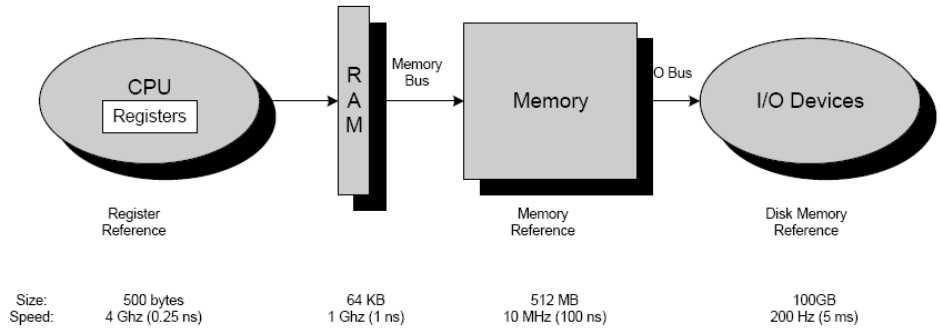

Figure 2.2 The levels in a typical memory hierarchy in embedded and desktop processors [8]

The highest level of memory is generally on-chip and has single or double cycle memory latency. The on-chip memory in embedded systems is architected either as a *cache* or as a *scratchpad memory (SPM).* A *cache* is a small, fast array of on-chip memory that stores portions of the most frequently accessed memory references, and is *controlled by hardware.* A scratchpad *memory (SPM),* also referred to as *tightly coupled memory (TCM)* in ARM processors, is also a small, fast array of on-chip memory. However, it is completely *controlled by software,* and it contents are also determined by software. The sections below describe and compare caches and SPMs in greater detail.

2.1 Caches

A *cache* is a fast on-chip memory, in which frequently used data elements are stored to make program execution faster. Each location in the cache contains a datum and a tag, which is the index of the datum in main memory and serves to identify the datum. When the processor wishes to read or write a location in main memory, it first checks whether that memory location is in the cache. This is accomplished by comparing the address of the memory location to all tags in the cache that might contain that address. If the processor finds that the memory location is in the cache, we say that a *cache hit* has occurred; otherwise we speak of a *cache miss*. In the case of a cache hit, the processor immediately reads or writes the data in the cache line. In the case of a cache miss, most caches allocate a new entry, which comprises the tag just missed and a copy of the data from memory, and replace an existing entry in the cache with this new entry. It is to be noted that the entire operation of the cache, described above, *is controlled by hardware.*

The heuristic used to select an entry for eviction, on a cache miss, is called the *replacement policy*. Hardware cache controllers implement one of several replacement policies like Least Recently Used (LRU) or Least Recently Replaced (LRR) [8]. The *associativity* of a cache determines the number of potential positions in the cache to which a particular entry from main memory can be mapped. If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called *fully associative*. At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is *direct mapped*. A compromise between fully associative and direct mapped is a *k-way set associative* cache, in which a particular entry can be placed in any one of *k* potential locations in the cache. 2-way, 4-way and 8-way set associative caches are the most common, in practice. It is to be noted that the greater the

associativity, the more the number of tag checks that need to be performed to determine a cache hit or miss. Thus, the replacement policy and the cache associativity helps the cache contents reflect the locality of references present in the program under execution. Figure 2.3 shows the basic hardware structure of a k-way set associative cache.

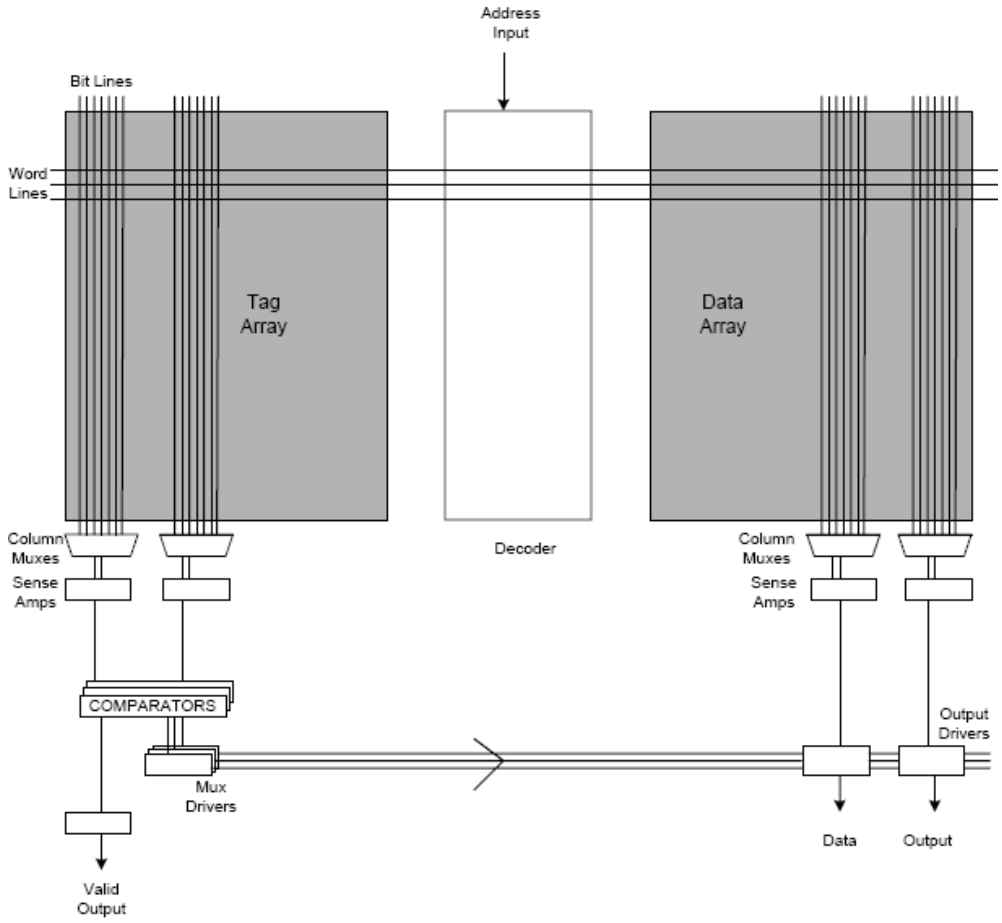

Figure 2.3 Hardware structure of a basic set-associative cache [13]

There are three types of cache misses. Compulsory misses are misses caused by the first reference to a datum, and are unaffected by changes in cache size and associativity [8]. Capacity misses are those caused by the size of the cache being insufficient, while conflict misses are those that could have been avoided, had the cache not evicted an earlier entry. Conflict misses can generally be reduced by increasing associativity or by altering the replacement policy.

The total power consumption of a cache is the sum of the power consumption of the tag array hardware, the data array hardware, comparators, multiplexers and output drivers as seen in Figure 2.3. Similarly, the total area of the cache is also the sum of the areas of each of the aforementioned components. This increase in energy and area is more significant for caches with higher associativities. It has thus been observed that caches are amongst the most power hungry components of the entire processor architecture. On-chip caches account for 25% of the total power consumption of the DEC Alpha 21164, and 43% of the total power consumption of the Strong Arm 1110 [12]. In the embedded space, caches are present in most ARM processors, the Motorola ColdFire MCF5 and the Intel PXA series processors.

2.2 Scratchpad Memories (SPMs)

A Scratchpad Memory (SPM) is a fast software-managed on-chip SRAM memory. The SPM is mapped into an address space disjoint from the off-chip main memory but connected to the same address and data buses.

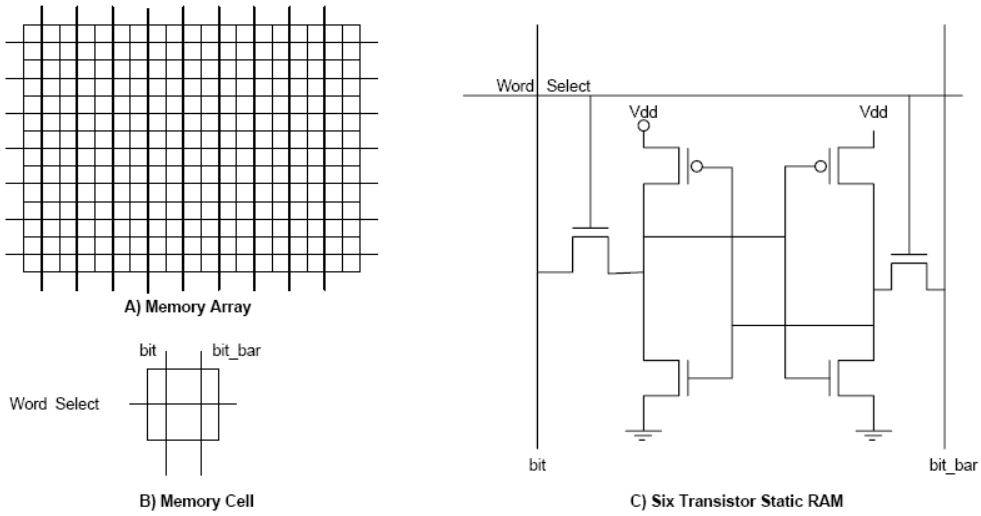

Figure 2.4 Hardware structure of the Scratchpad Memory array and its components [9]

The actual placement of data objects into the SPM address space is performed by software and is generally done in the last stage of the compiler. Thus, there is no need to check for the availability of the data/instruction in the SPM. From a hardware standpoint, this greatly reduces the hardware complexity of the SPM. There is no need for the comparators, multiplexers, the hit/miss acknowledge logic and the tag array. Figure 2.4 and Figure 2.5 show the basic hardware architecture of a scratchpad memory. The simplicity of the hardware architecture also greatly lowers the power consumption and area of a SPM. The total power consumption of the SPM is simply the sum of the power consumed by the decoder, the data array and the output drivers.

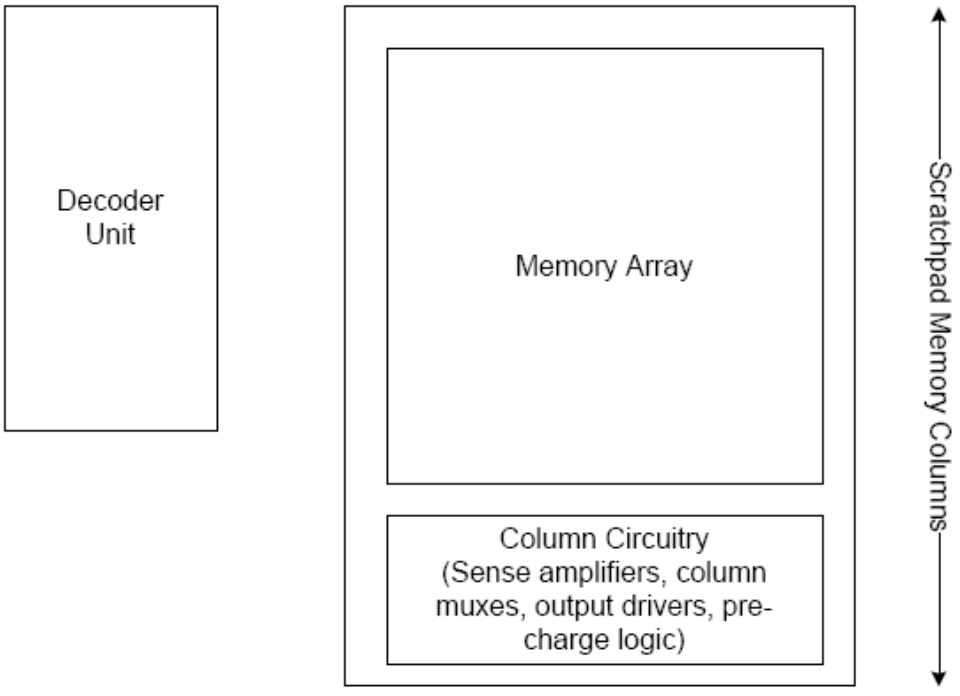

Figure 2.5 Architectural organization of a Scratchpad Memory [9]

It is to be noted that the effectiveness of caches is primarily based on their ability to maintain, at each time during program execution, the subset of data that is frequently used at that time in the cache. In other words, caches are able to dynamically track the changing locality of references in a program. This is achieved automatically by the hardware control and is completely invisible to the programmer. The SPM, being software controlled, requires specific mechanisms to ensure that the mapping of data to the SPM address space takes full advantage of the locality of references in the program being executed. In the simplest case, the programmer lays out the allocation of data and code to the SPM manually, based on the structure of the particular program. However, this often proves extremely tedious and requires considerable levels of expertise and experience on the part of the programmer. Several automatic scratchpad allocation algorithms have also been developed to address this issue. All the methods developed work at a low level and are primarily concerned with the allocation of stack and global variables, and affine arrays, based on compiler based profiling and liveness analysis. One such method is *software caching* [18], which emulates the workings of a hardware cache in software by maintaining software cache tags and hit/miss functions. Other methods aim at an energy optimized or latency optimized static allocation of variables by modeling the problem as an *Integer Linear Programming (ILP)* or *Dynamic Programming* problem [14, 15, 16, 19]. Yet other methods attempt to capture some of the dynamism in the program behavior and locality of references, by generating flow graphs and running graph partitioning algorithms on them [17, 20, 21].

Most embedded processors have scratchpad memories, including Motorola M-Core, the Sharc ADSP-21160M, the TigerSharc ADSP-TS201S, AT91C140 Atmel and

the DragonBall MC68SZ328. Some like the ARM 9 and ARM 11 series processors, have both caches and scratchpad memories.

2.3 A Comparison between Caches and Scratchpad Memories

In the context of embedded systems, scratchpad memories have proven to be the better choice for on-chip memory architecture. With the increase in the use of embedded processors for mobile applications, power consumption has become a critical limiting factor in view of limited battery capacity. As the prior sections have already outlined, the power consumption of scratchpad memories are significantly lower than that of caches. Scratchpad memories are also significantly smaller in area than caches of similar capacity. In [9], Banakar *et al* show that on average a scratchpad memory has 34% smaller area and 40% lower power consumption than a cache of the same capacity. Table 2.1 shows the energy per memory access for caches and scratchpad memories of different capacities.

MEMORY SIZE	CACHE	SCRATCHPAD
64 bytes	2.87 nJ	0.49 nJ
128 bytes	3.15 nJ	0.53 nJ
256 bytes	3.32 nJ	0.61 nJ
512 bytes	3.48 nJ	0.69 nJ
1024 bytes	3.75 nJ	0.82 nJ
2048 bytes	4.04 nJ	1.07 nJ
4096 bytes	4.71 nJ	1.21 nJ
8192 bytes	5.39 nJ	2.07 nJ

Table 2.1 Energy consumption per access for on-chip memories [14]

Caches also present a significant disadvantage in the context of deterministic memory access times and real time guarantees. It is often extremely difficult to deterministically predict the number of cache hits/misses for a particular program *a priori.* In certain cases, cache trashing, which is when a particular data element is read

into and evicted from the cache repeatedly, can cause significant increases in memory access times. Hence, the worst case execution time (WCET) for a cache based system has to assume a possible miss on every memory access. In the case of scratchpad memories, there is no concept of a hit/miss because the scratchpad mapping is controlled by software. Hence, it is always possible to deterministically compute the total memory access time for the entire program *a priori.*

The primary advantage of caches is the ease of integration with any software. Since cache control is done entirely in hardware, the cache is essentially invisible from the viewpoint of the programmer. This mechanism allows the use of software without any adaptation to the changed memory hierarchy. In order to make good use of the advantages of scratchpad memories, there is great need for efficient algorithms for scratchpad memory mapping and allocation. While low level algorithms already exist as mentioned above, there is a considerable lack of algorithms that work at a higher level of design. Bringing the advantages of scratchpad memories to a higher level of programming motivate the solutions developed in this thesis. The Heterochronous Dataflow (HDF) domain in Ptolemy II provides an expressive high level environment for describing a variety of operations and applications targeted at embedded systems. An efficient automated scratchpad allocation and mapping technique for HDF models would go a long way in making the use of scratchpad memories easier for programmers.

3. Problem Definition and Overview

The primary aim of this project is to formulate an automatic allocation scheme for mapping the key memory requirements of HDF models to the scratchpad memory. The scheme should provide a high level, coarse granularity scratchpad allocation for HDF model primitives. It should also primarily use the information intrinsically available in the very semantics and structure of the HDF model in order to make the allocation decisions. Such a high level allocation scheme shall prove advantageous to increasing the efficiency of code generated from block diagram programming, without requiring a programmer to have intimate knowledge of the hardware specifics of the device on which the generated code is run. Moreover, the allocation scheme shall be intimately related to the model of computation and its semantics, rather than the code generation technique or the generated low level code.

[22] identifies two primary memory requirements for SDF models. The first is the code memory which consists of the actor code for all the actors in the model. The second is the data memory which is the amount of memory used to buffer data between actors. Given that the HDF model of computation is an extension of SDF, we also consider actor code and buffer data memory as the key memory primitives of a HDF model for the purposes of scratchpad mapping.

3.1 Temporal Characteristics of Memory Accesses in a HDF Model

As described in the previous chapter, the on-chip memory must attempt to store the most frequently accessed memory references in order to improve memory access time significantly. The memory access patterns of a program change temporally and allocation

schemes must take these temporal changes into account to prove useful. Hence, it is necessary to study the temporal characteristics of the execution of a HDF model in order to appreciate the changes in its memory access patterns.

As described in chapter 1, HDF models can potentially change state at the end of a global iteration. However, for the duration of a particular global iteration, the state, the rate signatures and the schedule remain constant. This indicates that the execution of the HDF model can be viewed as a sequence of state transitions that occur at the end of every global iteration. If the state does not change, at the end of an iteration, it can be viewed as a transition back to its prior state. Thus the entire execution of a HDF model can be conceptually viewed as a path through a Trellis diagram.

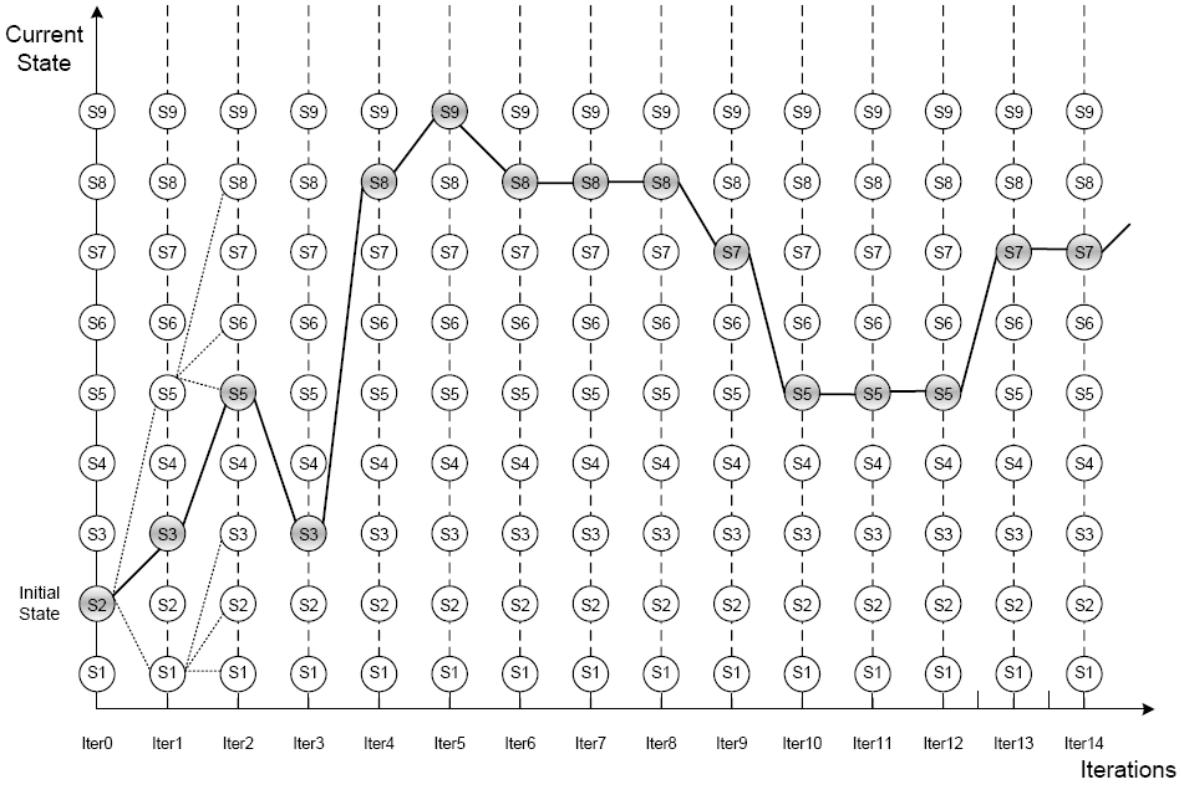

Figure 3.1 Trellis Diagram for the execution of a HDF model

Figure 3.1 shows the first fourteen iterations in the execution sequence of a HDF model with nine possible states. The model starts in initial state S_2 and then transitions to the highlighted state transitions at the end of each iteration. For the duration of any particular iteration, the model stays in the state highlighted for that iteration. The trellis diagram can also be embellished with all potential state changes at each iteration point. It would then depict all theoretically possible execution paths. All potential changes for the first two iterations are shown using dotted lines. Thus, the actual execution path is one of several possible paths through the trellis diagram. By virtue of the semantics of HDF, the actual path of execution is non-deterministic and not known *a priori*.

The diagram indicates that changes in memory access patterns follow the changes in state. Memory access patterns for a particular state, with its particular schedule and rate signatures, would be different from the access patterns for a different state. Thus, memory accesses would follow a particular trend for the duration of one iteration and then potentially change to a different trend at the next iteration.

3.2 Types of Scratchpad Allocation Schemes

There are three types of schemes for scratchpad memory allocation. These are *runtime allocation, dynamic allocation* and *static allocation.* The key point differentiating the types of schemes is the length of program execution for which the scratchpad memory allocation stays unchanged.

Runtime allocation refers to memory allocation schemes that continuously track the changing memory access profile by altering the scratchpad memory allocation at run time. The frequency of alteration might be as high as on every memory access. A runtime scheme would generate the best memory allocation given that the actual execution path

of a HDF model is not known *a priori*, but only at runtime. However, a runtime allocation technique is infeasible in the context of embedded software generated from HDF models. A runtime allocation technique would require the actual allocation algorithm to be implemented in embedded software, and executed at each stage to determine the next memory allocation. It is commonly the case that memory allocation algorithms are based on Linear Programming or Dynamic Programming solutions to NP-Hard and NP-Complete problems. The overhead for such an implementation, in embedded code, shall prove prohibitively expensive in both added code size and in execution speed. Moreover, the implementation of the allocation algorithm in embedded code would make the actual execution time less predictable. This would prove detrimental since predictability is often more important than optimality for many embedded applications.

Static allocation refers to a memory allocation scheme that remains unchanged for the entire length of program execution. In static allocation schemes, the memory allocation for the entire program is made *a priori* and left constant throughout program execution. Thus, static allocation schemes can be implemented prior to the code generation stage for HDF models and no significant overhead is incurred in the generated code. A key drawback of a fully static allocation is that it is often unable to capture the temporal changes in memory accesses. The restriction that the memory allocation is constant for the entire program makes it suboptimal for programs with wide variations in temporal localities of memory access.

Dynamic allocation refers to a memory allocation scheme can be altered at specific pre-identified program points but remains constant and unchanged in between these program points. Dynamic allocation schemes serve as a compromise between static

and runtime allocation schemes. By allowing the memory allocation to change, it allows temporal localities of memory accesses to be tracked. On the other hand, by allowing changes to occur only at pre-identified points, it ensures that the allocations can still be generated *a priori* without incurring significant overheads in the generated code. The key factor for the success of a dynamic allocation is the proper identification of the program points at which allocation changes can take place. The chosen program points should mark the boundaries between regions of varying memory access patterns. It should be noted that HDF exposes such boundaries at each point of transition from one state to another. This is one of the key reasons for choosing HDF as the model of computation for this work. In the case of HDF models, a dynamic allocation scheme would ensure predictable performance for any particular execution path through the trellis, as in Figure 3.1. A dynamic allocation scheme provides the benefits of both static and runtime allocation schemes and hence is the scheme of choice for our memory allocation algorithm for HDF models.

3.3 Key Assumptions

Assumption 1: Memory Architecture

For the purposes of this project, we assume a simple Harvard architecture for the memory system. We consider the code and data memory to be present in separate independent memory banks with independent buses. We consider the scratchpad memory to follow a Harvard architecture as well, with separate banks for code and data memory. Figure 3.2 depicts the memory architecture for this project. Since the scratchpad memory is completely controlled by software, the memory allocations determined by our allocation algorithm are pre-computed and stored in a *Memory Allocation List*. This list is

accessed at runtime at each memory allocation alteration point in the execution of the HDF model to assign the proper elements to the scratchpad memory.

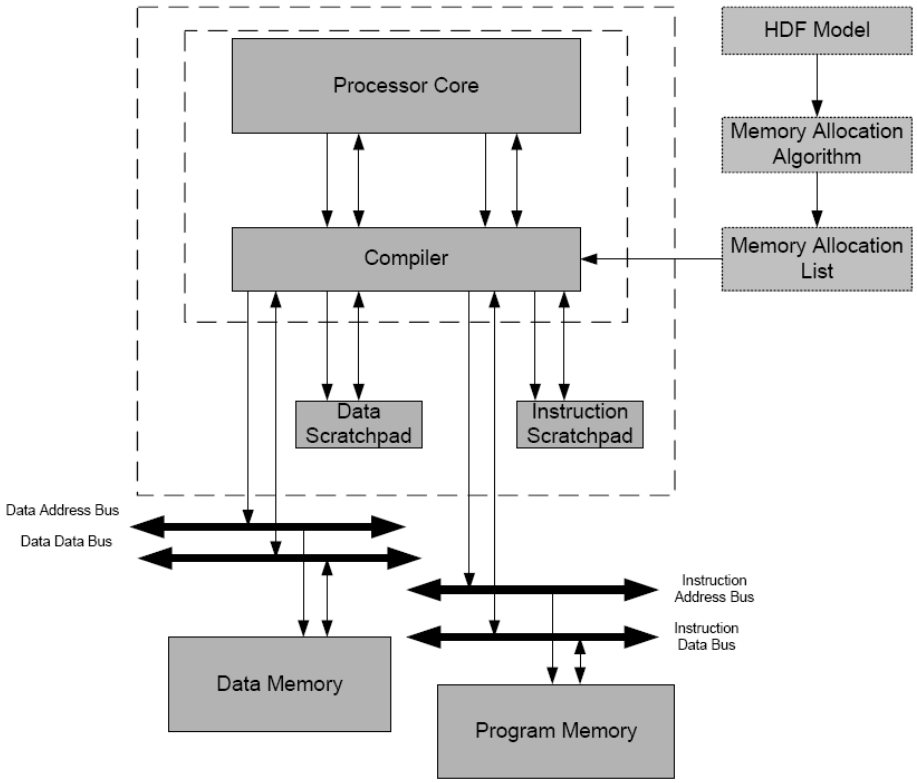

Figure 3.2 Modified Memory Architecture: The Instruction and Data Scratchpad are controlled by the compiler based on the allocations generated by the memory allocation algorithm

Assumption 2: The off-chip memory is large enough to contain all actor code and data buffer.

We assume a memory model where each lower level of memory contains all the contents of a higher memory level. In our case, this implies that the off-chip memory contains all the contents of the scratchpad in it. We also assume that the off-chip memory is theoretically infinite for algorithmic purposes and can accommodate the entire code and data memory requirements of the HDF model.

Assumption 3: All actors are opaque.

Atomic actors are considered opaque as is implied in the semantics of the HDF model of computation. We treat these actors solely as computational blocks and do not attempt to explore the implementation specifics of the actor itself. In other words, we are not concerned with any optimizations of memory requirements that are specific to the implementation of an atomic actor.

Assumption 4: Actors and Data Buffers are atomic units with respect to memory allocation.

We consider atomic actors and data buffers to be atomic entities for the purposes of memory allocation. We allow the code block for an atomic actor to be allocated to the scratchpad either in its entirety or not at all. The same applies for data buffers for inter-actor communication. Treating actors and buffers as atomic entities goes with our aim of providing a coarse granularity high level allocation that attempts to best utilize the semantics of the HDF model.

Assumption 5: Code size of an actor is representative of the number of accesses to code memory in a single invocation of an actor.

Given the assumption that actors are opaque and their internal implementation cannot be explored, we consider the code size of an actor to be the only information available to us to make allocation decisions. In order to make proper allocation decisions, we need to know the total number of accesses to code memory made during the execution of an actor. However, the actor code might contain branches, jumps and conditional loops that cause the actual number of accesses to be different from the code size parameter. The branches, jumps and conditional loops might also be input dependent and hence the number of memory accesses might vary between different runs of the HDF model.

Extensive profiling of the actor code serves as the only mechanism for gauging the number of accesses per invocation of an actor. However, for the purposes of this project we simplify the scenario by assuming the code size of an actor to be representative of number of accesses to code memory in a single invocation of the actor. It should also be stated that code size has been used as a parameter in our algorithms and hence can be easily replaced by the results of profiling, without requiring us to change the allocation algorithm.

3.4 Key Observations

As is clear from the discussion in Section 3.1, the memory access patterns vary from one state to the next while staying similar for the duration of a single state. It is also clear from section 1.3 that the schedule, rate signatures and buffer sizes remain constant for a particular state and vary across different states. Hence, we make a few key observations regarding the memory access characteristics of actor code and data buffer for a particular state.

Observation 1: The total number of code memory accesses for an actor in a particular state schedule is the product of its code size and firings.

Given *assumption 5*, it is clear that for an actor A_i, code size C_i represents the number of code memory accesses for a single invocation of an actor. For a given schedule in a particular state an actor is invoked f_i times, where f_i is the number of firings of the actor for that schedule. Hence:

$$\text{Number of code memory accesses} = f_i \cdot C_i$$

Observation 2: The code memory is read only with no write back occurring at any stage of model execution.

It is clear that in the HDF model no modifications are ever made to the actor code itself. Hence the actor code is always accessed in a read only fashion.

Observation 3: The cost of moving an actor to scratchpad memory is the product of the unit migration cost from off-chip to scratchpad memory and the code size of the actor.

In order to move the code block of an actor from off-chip memory to scratchpad memory, we need to move the number of memory elements equal to the code size C_i of the actor A_i. If the migration cost (time) for a single memory element is $T_{migration}$ then the cost of moving an actor to scratchpad is:

$$\text{Cost of migration} = T_{migration} \cdot C_i$$

Observation 4: The cost of evicting an actor from scratchpad memory is nil.

Given that the off-chip memory contains all contents of the scratchpad and that *observation 2* holds, it is clear that there is no need to write back any part of the code when an actor is evicted from the scratchpad. An eviction is thus no different than overwriting the appropriate memory locations.

Observation 5: The total number of data memory write accesses for a data buffer in a particular schedule is the product of production rate and firings of the source actor.

A single invocation of a source actor A_i produces p_i tokens on its outgoing channel, where p_i is the production rate of the actor. Storing each token results in a memory write. Hence for a given schedule, it clearly follows that:

$$\text{Number of data memory writes for a buffer} = p_i \cdot f_i$$

Observation 6: The total number of data memory read accesses for a data buffer in a particular schedule is the product of consumption rate and firings of the destination actor.

A single invocation of a destination actor A_i consumes c_i tokens on its incoming channel, where c_i is the consumption rate of the actor. Consuming each token results in a memory read. Hence for a given schedule, it clearly follows that:

$$\text{Number of data memory reads for a buffer} = c_i \cdot f_i$$

It also clearly follows that the total number of memory accesses to a data buffer in a particular schedule is the sum of the number of reads and writes. Hence:

$$\text{Number of memory accesses for a buffer} = p_{source} \cdot f_{source} + c_{destination} \cdot f_{destination}$$

Observation 7: The cost of moving a data buffer to scratchpad memory is the product of the unit migration cost from off-chip to scratchpad memory and the number of initial tokens present in the data buffer.

The preserved state of a data buffer D_i is the number of initial tokens I_i present on it prior to the beginning of an iteration. Hence when assigning a data buffer to the scratchpad, this initial state must be copied into the scratchpad. Therefore:

$$\text{Cost of moving a data buffer to scratchpad} = T_{OffChip->SPM} \cdot I_i$$

Observation 8: The cost of removing a data buffer from scratchpad memory is the product of the unit migration cost from scratchpad memory to off-chip memory and the number of initial tokens present in the data buffer.

Given that a single iteration returns the number of tokens in a data buffer to its initial number, the remaining tokens must thus be written back to off-chip memory upon evicting a data buffer from scratchpad. Therefore:

$$\text{Cost of evicting a data buffer from scratchpad} = T_{SPM->OffChip} \cdot I_i$$

4. Dynamic Scratchpad Allocation Scheme

In the previous chapters we have motivated the analysis of both the memory hierarchy and the dataflow models of computation. We have also laid the groundwork and stated the basic observations regarding the memory access characteristics of HDF models. In this section we develop an allocation technique to map the actor code blocks and data buffers of HDF models to scratchpad memory.

The allocation scheme developed is a *dynamic* allocation scheme. In chapter 3, we have already shown that the memory access patterns for a HDF model change along with changes in state. We therefore select state changes as the points in model execution when the allocation of the scratchpad can be altered. An allocation is kept unchanged for the duration of a particular state. Our scheme allows us to compute the memory allocations for each state *a priori.*

4.1 Overall Structure of Allocation Scheme

Figure 4.1 shows the overall organization on the dynamic allocation scheme. The scheme is separated into two distinct phases, a *pre-execution* stage completed prior to model execution and a memory mapping stage during execution.

In the pre-execution stage, the schedules and buffer memory requirements for all states in the HDF model are generated. It should be noted that this step is already a part of code generation methodology in Ptolemy II and hence adds no extra overhead to the memory allocation scheme. The generated state schedules and buffer memory requirements are supplied as inputs to the dynamic memory allocation algorithm described in Section 4.2. The algorithm generates a memory map for each state which

identifies the actors and buffers that are to be placed in the scratchpad. The memory map for each state schedule is stored in a list labeled with the rate signatures for state identification.

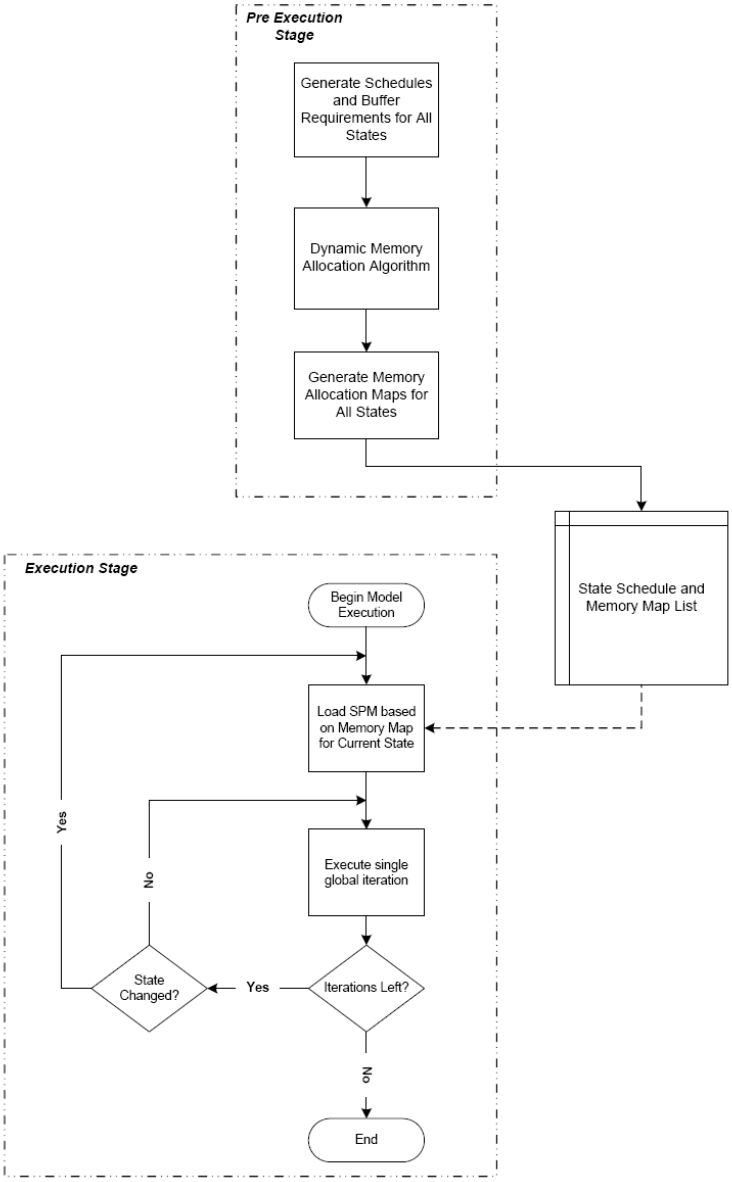

Figure 4.1 Overall Structure of Memory Allocation Scheme

During the *execution* of the HDF model, state transitions are identified at the end of each global iteration. If a state transition is detected then the scratchpad memory is

loaded with the actor code blocks and data buffers specified by the memory map for the particular state, prior to the execution of the iteration in that state. This stage is intelligent in that it only loads those actor code blocks and data buffers to the scratchpad memory that are not already present in it. This is repeated at the end of each iteration until the execution is complete.

4.2 Memory Allocation Algorithm

The memory allocation algorithm generates the list of actors and data buffers to be placed in the scratchpad for a particular state in the HDF model, based on the schedule, rate signatures and memory requirements for the actors and buffers. Since the general problem of optimal data allocation is known to be NP-complete, the problem has been formulated as a 0-1 Integer Linear Programming problem.

4.2.1 Formulation of Variables

Here we present the variables for the ILP formulation.

A = Number of actors in the current state

a_i = i th actor, $i \in [1, A]$

$S(a_i)$ = Code size of a_i in bytes

$F(a_i)$ = Number of firings of a_i in current state schedule

D = Number of data buffers in the current state

d_i = i th data buffer, $i \in [1, D]$

$S(d_i)$ = Size of d_i in bytes

S_{token} = Size of a token in bytes

$I(d_i)$ = Number of initial tokens on d_i

$prod(d_i)$ = Production rate of source actor for d_i

$cons(d_i)$ = Consumption rate of destination actor for d_i

$F_{source}(d_i)$ = Number of firings of the source actor of d_i in current state schedule

$F_{dest}(d_i)$ = Number of firings of the destination actor of d_i in current state schedule

$Size_{DataSPM}$ = Size of Data Scratchpad in bytes

$Size_{InstrSPM}$ = Size of Instruction Scratchpad in bytes

$T_{OffChip_{Rd}}$ = Time to read a byte from OffChip memory in cycles

$T_{OffChip_{Wr}}$ = Time to write a byte to OffChip memory in cycles

$T_{SPM_{Rd}}$ = Time to read a byte from Scratchpad in cycles

$T_{SPM_{Wr}}$ = Time to write a byte to Scratchpad in cycles

$T_{OffChip \rightarrow SPM}$ = Time to move a byte from OffChip to Scratchpad in cycles

$T_{SPM \rightarrow OffChip}$ = Time to move a byte from Scratchpad to OffChip in cycles

For our scratchpad memories, we assume that the read, write and migration times for both data and instruction scratchpad are identical and list them simply as SPM in the variables. The optimization problem is formulated as separate 0/1 Integer Linear Programs for actors and data buffers. Hence, the following set of 0/1 integer variables are defined:

$$M_{OffChip}(a_i) = \begin{cases} 1 \text{ if actor } a_i \text{ is located in OffChip Memory} \\ 0 \text{ otherwise} \end{cases}$$

$$M_{SPM}(a_i) = \begin{cases} 1 \text{ if actor } a_i \text{ is located in Scratchpad} \\ 0 \text{ otherwise} \end{cases}$$

$$M_{OffChip}(d_i) = \begin{cases} 1 \text{ if data buffer } \mathrm{d}_i \text{ is located in OffChip Memory} \\ 0 \text{ otherwise} \end{cases}$$

$$M_{SPM}(d_i) = \begin{cases} 1 \text{ if data buffer } \mathrm{d}_i \text{ is located in Scratchpad} \\ 0 \text{ otherwise} \end{cases}$$

It should be noted that the above 0/1 variables are complementary and hence one can be represented as the complement of the other, thereby reducing the computational complexity of the integer program.

$$M_{OffChip}(a_i) = 1 - M_{SPM}(a_i)$$
$$M_{OffChip}(d_i) = 1 - M_{SPM}(d_i)$$

They have been presented as shown to make it simple to extend it to a multiple memory hierarchy scenario.

4.2.2 Objective Function and Constraints for Actor Allocation

Here we present the objective function for the allocation of actor code to scratchpad memory. The key objective is to generate a cost optimal allocation. In our case, we are considering memory access times as the cost criterion. We will therefore seek to find the allocation that minimizes the total access time for actor code in a particular state. The total memory access time for all actor code in one complete iteration of a particular state is:

(Eq. 4.1)

$$\sum_{i=1}^{A}\left\{M_{OffChip}(a_i)\left(T_{OffChip_{Rd}}\cdot F(a_i)\cdot S(a_i)\right)+M_{SPM}(a_i)\left(T_{SPM_{Rd}}\cdot F(a_i)\cdot S(a_i)+T_{OffChip\rightarrow SPM}\cdot S(a_i)\right)\right\}$$

The first term specifies the time spent in memory reads from off-chip memory (*Observation 1, 2. Section 3.4)*. The first part of the second term specifies the time spent in memory reads from scratchpad while the second part specifies the time spent in moving the code block from off-chip memory to scratchpad (*Observation 1, 3. Section 3.4).* The summation ensures that the memory access times for all actors are taken into consideration. This is the function that we seek to **minimize**. As is the case in all linear optimization problems, we also have a few constraints:

$$M_{OffChip}(a_i)+M_{SPM}(a_i)=1 \quad \left(\forall i\in[1,A]\right)$$

$$\sum_{i=1}^{A}M_{SPM}(a_i)\cdot S(a_i)\leq Size_{InstrSPM}$$

The first set of constraints ensures that each actor is located in one memory unit only. The second constraint ensures that the sum of the sizes of all the actors assigned to the

scratchpad does not exceed the size of the scratchpad. We do not place a similar constraint for off-chip memory due to *Assumption 2, Section 3.3.*

4.2.3 Objective Function and Constraints for Data Buffer Allocation

The objective function for data buffers is somewhat more complicated. This is due to two reasons:

1. Data buffers are both read from and written to. (*Observation 5, 6. Section 3.4)*
2. Initial tokens in data buffer need to be considered. *(Observation 7, 8. Section 3.4)*

However, the key objective once again is to minimize the memory access time. The total memory access time for all accesses to data buffers in one complete iteration of a particular state is:

(Eq. 4.2)

$$\sum_{i=1}^{D}\Big\{M_{OffChip}(d_i)\Big(T_{OffChip_{Wr}}\cdot prod(d_i)\cdot F_{source}(d_i)+T_{OffChip_{Rd}}\cdot cons(d_i)\cdot F_{dest}(d_i)\Big)$$
$$+M_{SPM}(d_i)\Big(T_{SPM_{Wr}}\cdot prod(d_i)\cdot F_{source}(d_i)+T_{SPM_{Rd}}\cdot cons(d_i)\cdot F_{dest}(d_i)+T_{OffChip\rightarrow SPM}\cdot I(d_i)+T_{SPM\rightarrow OffChip}\cdot I(d_i)\Big)\Big\}$$

The first term specifies the time spent in memory accesses from off-chip memory (*Observation 5, 6. Section 3.4).* The first part of the second term computes the corresponding access time for buffers placed in the scratchpad. The second part of the second term specifies the time spent in moving the number of initial tokens from off-chip memory to scratchpad and the time spent in moving the tokens left in the buffer at the completion of the iteration to off-chip memory (*Observation 7, 8. Section 3.4).* The summation ensures that the memory access times for all data buffers are taken into consideration. This is the function that we seek to **minimize**. It should be noted that the entire function should be multiplied by S_{Token} in order for it to be an accurate expression.

However, since an overall multiplicative factor does not alter the minimization of the function we have decided to eliminate it from the minimization expression.

The constraints for data buffers are similar to constraints for the actors:

$$M_{OffChip}(d_i) + M_{SPM}(d_i) = 1 \quad (\forall i \in [1, D])$$

$$\sum_{i=1}^{A} M_{SPM}(d_i) \cdot S(d_i) \leq Size_{DataSPM}$$

4.2.4 Extensions to the ILP Formulation

The integer linear program formulated above can be easily extended and modified. In order to optimize the scratchpad allocation for minimum energy consumption, instead of minimum access time, the objective functions and variables have to be adjusted. The time variables shall have to be replaced with the corresponding energy consumption values:

$E_{OffChip_{Rd}}$ = Energy consumed to read a byte from OffChip memory in cycles

$E_{OffChip_{Wr}}$ = Energy consumed to write a byte to OffChip memory in cycles

$E_{SPM_{Rd}}$ = Energy consumed to read a byte from Scratchpad in cycles

$E_{SPM_{Wr}}$ = Energy consumed to write a byte to Scratchpad in cycles

$E_{OffChip \rightarrow SPM}$ = Energy consumed to move a byte from OffChip to Scratchpad in cycles

$E_{SPM \rightarrow OffChip}$ = Energy consumed to move a byte from Scratchpad to OffChip in cycles

The constraint functions shall remain unchanged. The objective functions shall remain identical except for replacing the time variables with the energy variables:

(Eq. 4.3, 4.4)

$$\sum_{i=1}^{A} \left\{ M_{OffChip}(a_i) \left(E_{OffChip_{Rd}} \cdot F(a_i) \cdot S(a_i) \right) + M_{SPM}(a_i) \left(E_{SPM_{Rd}} \cdot F(a_i) \cdot S(a_i) + E_{OffChip \rightarrow SPM} \cdot S(a_i) \right) \right\}$$

$$\sum_{i=1}^{D} \left\{ M_{OffChip}(d_i) \left(E_{OffChip_{Wr}} \cdot prod(d_i) \cdot F_{source}(d_i) + E_{OffChip_{Rd}} \cdot cons(d_i) \cdot F_{dest}(d_i) \right) \right.$$
$$\left. + M_{SPM}(d_i) \left(E_{SPM_{Wr}} \cdot prod(d_i) \cdot F_{source}(d_i) + E_{SPM_{Rd}} \cdot cons(d_i) \cdot F_{dest}(d_i) + E_{OffChip \rightarrow SPM} \cdot I(d_i) + E_{SPM \rightarrow OffChip} \cdot I(d_i) \right) \right\}$$

In the above fashion, one could also envision an optimization that takes both energy consumption and access time into consideration. For such a combined optimization, the energy/time variables shall have to be replaced with cost variables. The cost variables can be easily defined as weighted products of energy and time values, to proportionally account for both energy consumption and access time.

The integer linear program can also be extended very easily to allow optimization for multiple memory hierarchies. The current formulation considers a two-level hierarchy with a scratchpad and an off-chip memory only. However, various embedded processors have more than one level of scratchpad memory with variable access times and energy consumptions. For such a case, certain variables would have to be modified as such:

U = Number of memory units
$Size_j$ = Size of the jth memory unit in bytes $j \in [1,U]$
$T_{j_{Rd}}$ = Time to read a byte from the jth memory unit in cycles $j \in [1,U]$
$T_{j_{Wr}}$ = Time to write a byte to the jth memory unit in cycles $j \in [1,U]$
$T_{j \to Main}$ = Time to move a byte from the jth memory unit to main memory unit in cycles $j \in [1,U]$
$T_{Main \to j}$ = Time to move a byte from main memory to the jth memory unit in cycles $j \in [1,U]$

There shall have to be 0/1 integer linear variables for every memory unit as such:

$$M_j(a_i) = \begin{cases} 1 \text{ if actor } a_i \text{ is located in the } j\text{th memory unit } j \in [1,U] \\ 0 \text{ otherwise} \end{cases}$$

$$M_j(d_i) = \begin{cases} 1 \text{ if data buffer } \mathrm{d}_i \text{ is located in the } j\text{th memory unit } j \in [1,U] \\ 0 \text{ otherwise} \end{cases}$$

The objective functions would now have to include a double summation to account for all memory units in addition to the actors/buffers. The formulation also makes a simplifying assumption that all actor code and data buffers are moved between the main memory and an intermediate stage, but not in between different intermediate stages.

(Eq. 4.5, 4.6)

$$\sum_{j=1}^{U}\sum_{i=1}^{A}\left\{M_j(a_i)\left(E_{j_{Rd}}\cdot F(a_i)\cdot S(a_i)+E_{Main\to j}\cdot S(a_i)\right)\right\}$$

$$\sum_{j=1}^{U}\sum_{i=1}^{D}\left\{M_j(d_i)\left(E_{j_{Wr}}\cdot prod(d_i)\cdot F_{source}(d_i)+E_{j_{Rd}}\cdot cons(d_i)\cdot F_{dest}(d_i)+E_{Main\to j}\cdot I(d_i)+E_{j\to Main}\cdot I(d_i)\right)\right\}$$

Allowing for movement between intermediate stages would require further modifications to the objective functions. The constraint functions shall also have to be modified in the form:

$$\sum_{j=1}^{U} M_j(a_i)=1 \quad (\forall i\in[1,A])$$

$$\sum_{i=1}^{A} M_j(a_i)\cdot S(a_i)\le Size_j\ (\forall j\in[1,U])$$

4.3 Observations, Improvements and Drawbacks of the Allocation Algorithm

The scratchpad allocation algorithm provides an optimal memory allocation for a particular state of a HDF model. In a given state, the HDF model follows the exact semantics of an SDF model. Hence, a particular state of a HDF is essentially a pure SDF model. This allows us to make a key observation regarding the algorithm:

Observation 9: The scratchpad allocation algorithm when applied to a pure Synchronous Dataflow (SDF) model will generate the optimal static allocation for the model.

Thus our algorithm can also be used with SDF models. The allocation generated in such a case will be a static allocation, i.e. it will remain constant for the entire execution of the SDF model.

The overall quality of results of the above algorithm is greatly dependent on the memory requirements of the actor blocks and the data buffers. These memory requirements can be improved by performing schedule based optimizations as described in [22]. The size of the data buffers can also be reduced by using techniques such as modulo addressing etc. described in [22]. Hence, one could envision using these optimizations as a pre-processing step to our memory allocation algorithm in order to improve the quality of results.

The memory allocation algorithm has one drawback. It implicitly assumes that at the beginning of each state, all selected actors and buffers have to be copied into the scratchpad from the off-chip memory. The key reason for this is that the sequence of state changes in a HDF model cannot be determined *a priori*. Hence, we are unable to make any assumptions regarding the current allocation of the scratchpad memory at the beginning of each state. The implicit assumption may in reality make us account for time spent in moving an actor/buffer which is already in the scratchpad. In other words, some actors and buffers might already be present in the scratchpad at the start of a state and while there would be no actual time spent in bringing them in, we nonetheless account for the migration time in our optimization. Only a runtime algorithm shall be able to fully take into account the current allocation in order to determine the optimal assignment of actors and buffers to the scratchpad. However, as stated previously, runtime algorithms would prove prohibitively expensive.

It can however be observed that allowing multiple possible allocations for each state as compared to the current single allocation per state, should improve the overall performance. There are various possible ways of generating multiple allocations. In the next section we present a modification to the original scheme for generating multiple

allocations for a particular state. The allocation to be selected for a given state is determined by its predecessor state. Hence, we attempt to take some path information into account while generating multiple allocations. This primary purpose of this modification is to motivate the fact that it might be possible to improve performance by allowing multiple allocations per state based on different factors such as probability of reaching each state, path based criteria as observed from a trellis diagram such as Figure 3.1, etc. While a complete exploration of multiple allocations per state is beyond the scope of this report, it definitely serves as a promising direction for future work in this field.

4.4 Algorithm to Generate Multiple Allocations per State

In this section we present a modification to the original allocation algorithm which allows us to generate multiple allocations per state. It should be noted that this modification is aimed specifically for the allocation of actor code. This is because in the case of actors, a significant amount of code has to be moved to and from the off-chip memory to the scratchpad, every time an actor is brought into or evicted from the scratchpad. Allowing multiple allocations seeks to lower this migration cost. In the case of data buffers, there are no migration costs except when there are delay/initial tokens present in them. Hence, in the general case there is no benefit to be gained by aiming to reduce migration costs, since overall migration costs for data buffers are already negligible. As stated earlier, this modification is aimed at serving as a motivation for future exploration into multiple allocations per state. It does not generate allocations that provably minimize migration costs. It is more of a heuristic that is expected to lower migration costs in most general cases.

4.4.1 Formulation of Additional Variables

For this modification to the original algorithm we preserve all the variables and parameters introduced in Section 4.2. The following are the additional variables that are required for the modification:

$S =$ Number of valid states in the HDF model
$s_i =$ i th state, $i \in [1, S]$
$I(s_i) =$ Number of valid input paths into state s_i
$I_j(s_i) =$ The j th valid input path into state s_i, $j \in [1, I(s_i)]$
$Pd_j(s_i) =$ The predecessor state on the j th valid input path into state s_i, $j \in [1, I(s_i)]$
$\alpha(s_i) =$ Number of memory allocations for state s_i
$\alpha_1(s_i) =$ The initial memory allocation for state s_i
$\alpha_j(s_i) =$ The j th memory allocation for state s_i, $j \in [2, \alpha(s_i)]$

$I(s_i)$ represents the number of possible ways in which one can transition into state s_i.

The following is an additional variable needed for the modified objective function. $M_{OffChip}^{Predecessor}(a_i)$ is a 0/1 variable that identifies whether a particular actor was in off-chip memory in the predecessor state, i.e. the state from which we transitioned into the current state. It should be noted that this is not a 0/1 integer variable for solution by the linear program, but rather a variable that is set to the appropriate 0/1 value when composing the objective function.

$$M_{OffChip}^{Predecessor}(a_i) = \begin{cases} 1 \text{ if actor } a_i \text{ was located in the OffChip Memory in the predecessor state} \\ 0 \text{ otherwise} \end{cases}$$

4.4.2 Objective Function and Constraints for Actor Allocation

We are required to make a minor modification to the objective function as given below:

(Eq 4.7)

$$\sum_{i=1}^{A} \Big\{ M_{OffChip}(a_i)\Big(T_{OffChip_{Rd}} \cdot F(a_i) \cdot S(a_i)\Big) + M_{SPM}(a_i)\Big(T_{SPM_{Rd}} \cdot F(a_i) \cdot S(a_i) + M^{Predecessor}_{OffChip}(a_i).T_{OffChip \rightarrow SPM} \cdot S(a_i)\Big)\Big\}$$

As can be seen the only change that has been made is that the migration cost of moving an actor from off-chip to scratchpad has been augmented by $M^{Predecessor}_{OffChip}(a_i)$. This ensures that we account for the migration cost only if an actor was not present in the scratchpad in the predecessor state and actually needed to be moved in from off-chip for the current state transition.

The constraints for the objective function remain same as in Section 4.2.2.

4.4.3 Procedure for Generation of Multiple Allocations per State

The multiple allocations scheme generates $1+I(s_i)$ allocations per state, where $I(s_i)$ is the number of valid input paths into state s_i. Thus, $\alpha(s_i) = 1+I(s_i)$. The procedure for generating these allocations is as follows:

Step 1. For each state s_i generate initial allocation $\alpha_1(s_i)$ using the original objective function (Eq 4.1)

Step 2. For each state s_i

Step 2.1. For all input paths j, with predecessor state $Pd_j(s_i)$ generate:

The j^{th} allocation $\alpha_j(s_i)$ using the modified objective function (Eq 4.7) and setting $M^{Predecessor}_{OffChip}(a_i)$ according to $\alpha_1(Pd_j(s_i))$, the initial allocation for the predecessor state.

The above procedure generates $\alpha(s_i)$ allocations for each state.

4.4.4 Procedure for Choice of Appropriate Allocation during State Transition

In the prior subsection we have shown the procedure for generating multiple allocations for each state. In this section we present the method for choosing the appropriate allocation for a particular state when we enter the state during a state transition of the HDF model. In order to perform all the computations prior to runtime, we generate all valid predecessor-successor state pairs. A valid predecessor-successor state pair is a pair of states (S_i, S_j) where it is possible to transition directly from state S_i to state S_j. Thus, we are attempting to choose the appropriate allocation for a particular state given the knowledge of its predecessor state and the allocation for the predecessor state.

For each of the allocations $\alpha(S_i)$ for state S_i, we select the appropriate allocation $\alpha_{appr}(S_j)$ for state S_j, from the $\alpha(S_j)$ possible allocations for state S_j. The appropriate allocation $\alpha_{appr}(S_j)$ is the one for which the migration cost on transitioning from state S_i to S_j is the lowest of all the possible $\alpha(S_j)$ allocations. This procedure is repeated for all valid predecessor-successor state pairs.

The above information is stored as a map in which the predecessor state, the allocation for the predecessor state and the successor state serves as key and the appropriate allocation found by the method above serves as the value. At runtime when a state transition occurs, the predecessor state, its allocation and the successor state is used to index into the map to find the appropriate allocation for the successor state.

It should be noted that during the very first iteration of the HDF model, there is no predecessor state and the initial allocation $\alpha_1(S_i)$ is used.

4.5 Observations and Analysis of the Modified Allocation Algorithm

The modified allocation algorithm described above serves as a first step to motivate further exploration of ways to generate multiple allocations per state in HDF models. The algorithm is considerably more expensive both in terms of computation time and storage space requirements. For a worst case analysis, let us assume a fully connected system in which one can transition from one state to any of the other states including itself. Let the number of states in the system be N. There are N^2 valid predecessor-successor state pairs since the system is fully connected. Also, it is possible to transition into a particular state from all other states. Hence, there are N possible input paths into each state, i.e. N+1 memory allocations per state. Thus the total number of allocations that need to be generated is $N \cdot (N+1)$. Also, the total number of appropriate allocations that need to be stored is $N^2 \cdot (N+1)$, since there is one appropriate allocation for each possible allocation in the predecessor state. As can be seen even for moderately sized models the worst case scenario can prove prohibitively expensive. Nonetheless, it is unlikely from a practical standpoint to have a fully connected HDF model with a large number of states. Hence, it can be argued that the above algorithm would remain feasible for most practical HDF models. It should also be noted that all the computation is done prior to runtime. Hence, small variations in processing time may not be a major concern.

We can also clearly observe that the above algorithm does not generate a provably minimal set of allocations. It rather serves as a heuristic method of generating multiple allocations per state and selecting the one among them that would lower migration costs the most. However, the modification clearly shows that there might be many possible ways to take advantage of multiple allocations in improving the performance of the scratchpad memory.

5. Performance Analysis

In this chapter, we evaluate the performance of the allocation algorithm with respect to various parameters that affect its results. We also assess the scalability of the algorithm. The allocation algorithm was implemented in Ptolemy II, a Java-based framework for studying modeling, simulation and design of concurrent real-time systems [7]. The open-source linear programming system, LP Solve, was used as the solver for the integer programs [23]. To analyze the performance of the allocation algorithm, we applied it to the Adaptive Coding Model [11] developed in the HDF domain of Ptolemy II. The algorithm was also applied to a set of randomly generated HDF graphs to further study its performance.

In order to study the algorithm, the following memory characteristics were used. A two-level memory hierarchy was assumed with the Scratchpad Memory forming the first level. The SPM was assumed to have a 1-cycle read/write latency. The second level off-chip memory was assumed to have a 10-cycle read/write latency. DMA and pseudo-DMA mechanisms greatly speed up the transfer of data between the SPM and the off-chip memory, as in the Motorola MCORE processor [24]. Transfer times assuming DMA and pseudo-DMA mechanisms, were analyzed by Udayakumaran et al. in [17]. Basing ourselves on this analysis, the transfer times for data transfer between the SPM and the off-chip memory was assumed to be 2.5 cycles. We use caches as the comparison case to study the performance of our allocation scheme. The cache is assumed to be a fully-associative cache, with a 1-cycle hit and a 12-cycle miss latency. A Least Recently Used (LRU) replacement strategy is assumed for the cache. The cache is assumed to be a write-back cache with a write-allocate miss policy [8]. In order to better analyze the true

performance of our algorithm, the sizes of the cache and SPM were assumed to be varying percentages of the total code and data size of the model, rather than considering an absolute size. The off-chip memory is assumed to be large enough to hold all program data and code. Table 5.1 summarizes the memory characteristics considered for our experiments.

MEMORY CHARACTERISTIC		SPECIFICATION
SPM	**Read Latency**	1 cycle
	Write Latency	1 cycle
	Transfer Time (to/from OffChip)	2.5 cycles
Cache	**Hit Latency**	1 cycle
	Miss Latency	12 cycle
	Associativity	Fully Associative
	Replacement Strategy	Least Recently Used

Table 5.1 Memory Specification used for experiments

5.1 Adaptive Coding

In this section we present a thorough analysis of the allocation algorithm applied to the Adaptive Coding Model [11] in the HDF domain of Ptolemy II. The Adaptive Coding model demonstrates a wireless communication scenario in which the dataflow is switched between two encoders and decoders, with different consumption and production rates. Such a scenario can occur when one aims to preserve data quality in spite of varying levels of channel loss. A sophisticated coding scheme is chosen to reduce channel loss when signal strength is low and a simple scheme is used when signal strength is high. In the example, the model has two modes of Hamming Coding-Decoding, a (7,4) Hamming Code and a (3,1) Hamming Code. Switch, which produces the signal that chooses the coding scheme to be used, can be assumed to be the input from a performance detector, or a signal strength sensor. There are 4 possible states in this model of which only 2 states, the (7,4) Codec and the (3,1) Codec are relevant. The

states in which a (7,4) Coder is paired with the (3,1) Decoder and vice-versa are invalid and cannot be reached. The model is shown in Figure 5.1 and 5.2 in all its levels of hierarchy. Fig 5.2 shows the inside of Count Errors. The details of the Adaptive Coding model that are relevant to the Allocation Algorithm are summarized in Table 5.2. Figure 5.3 depicts the adaptive coding model as a HDF graph, with the actors simplified to Actor A, Actor B and so on, and production and consumption rates shown.

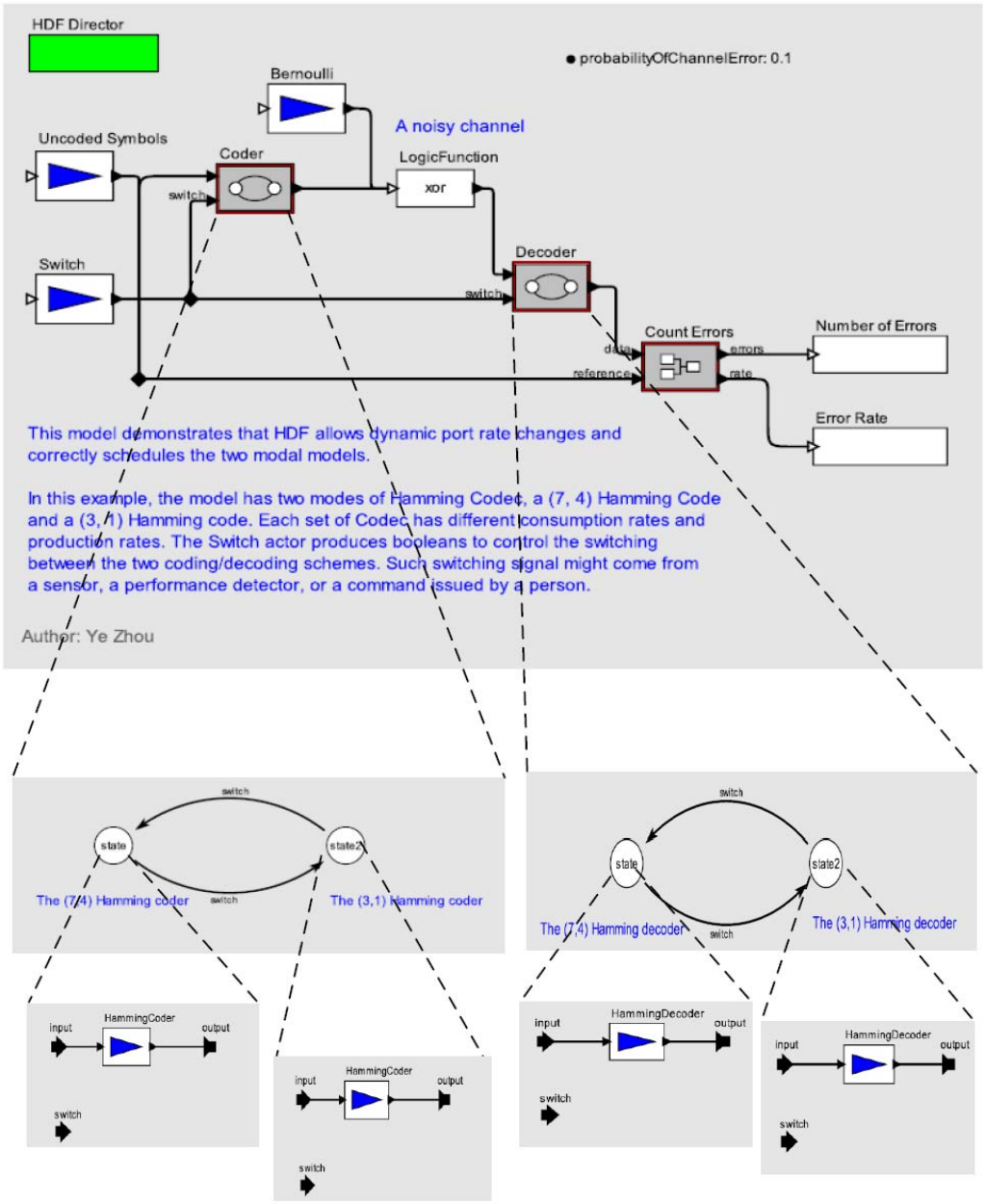

Figure 5.1 The Adaptive Coding Model

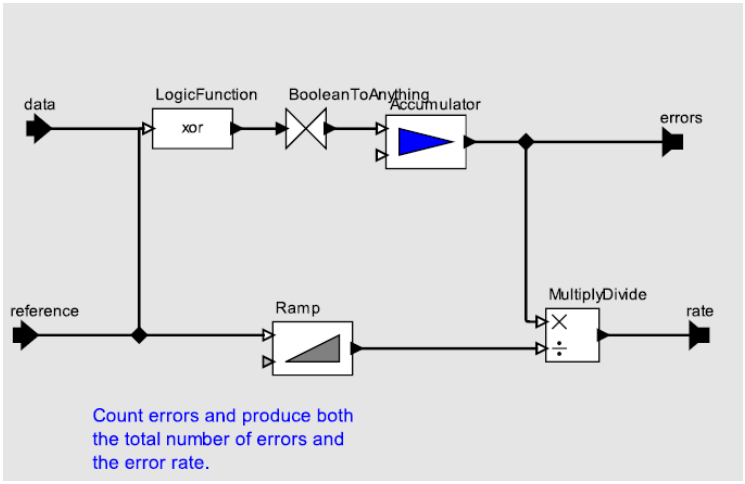

Figure 5.2 Expanded View of Count Errors

CHARACTERISTICS		VALUE
Actor	**Number of Actors**	15
	Total Size	112
Data Buffer	**Number of Buffers**	15
	Total Size in (7,4) State	63
	Total Size in (3,1) State	21
State	**Total Number of States**	4
	Number of Reachable States	2
	State 1	(7,4) Hamming Code-Decode
	State 2	(3,1) Hamming Code-Decode

Table 5.2 Relevant Characteristics of Adaptive Coding HDF Model

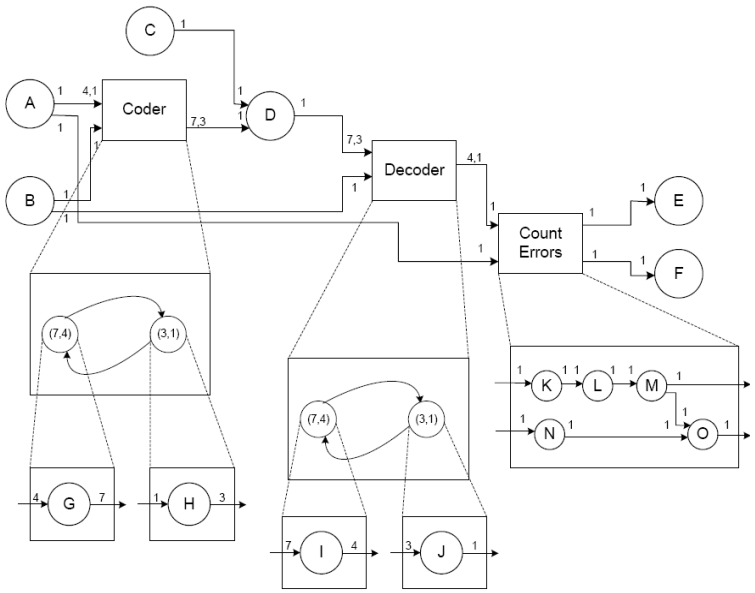

Figure 5.3 Adaptive Coding in HDF graph form

5.1.1 Performance Analysis for Actor Allocation for the Adaptive Coding Model

We first analyze the performance of the allocation algorithm for actor code. The performance for the allocation of data buffers is provided in the next subsection. In both State 1 and State 2, there are a total of thirteen actors. The schedules for the states are as follows:

State 1: (4A)(B)(G)(7C)(7D)(I)(4K)(4L)(4M)(4N)(4O)(4E)(4F)

State 2: (A)(B)(H)(3C)(3D)(J)(K)(L)(M)(N)(O)(E)(F)

The sizes of the actors A through O are 5, 5, 5, 10, 2, 2, 10, 10, 11, 11, 10, 6, 7, 10 and 8 units respectively. The Allocation Algorithm was used to generate the memory allocations for the two states.

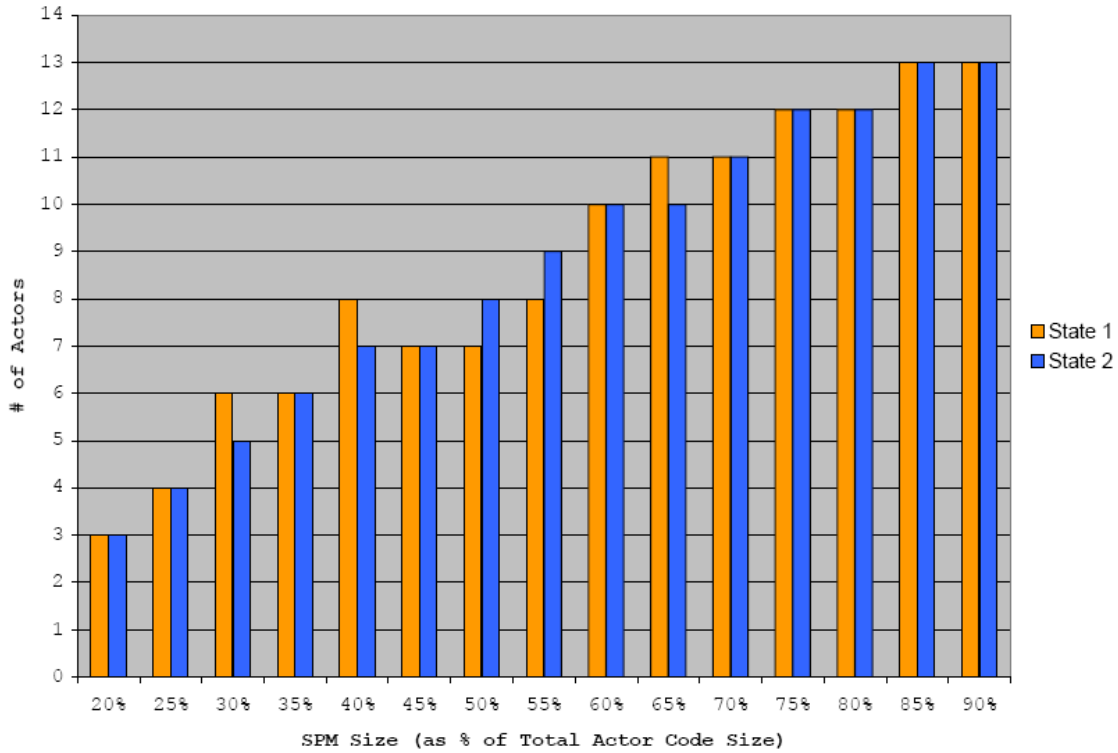

Figure 5.4 Number of Actors allocated to SPM for each state

Figure 5.4 shows the number of actors, out of the total 13, allocated to the SPM for varying SPM sizes. The size of the SPM is varied as a percentage of the total actor size. As expected the number of actors allocated to the SPM increases with increase in SPM size. Figure 5.5 and Figure 5.6 shows the number of memory accesses that occur from the SPM and Off-chip memory for varying SPM size, for State 1 and State 2 respectively. We can clearly see that the number of accesses to the SPM increases sharply with increase in SPM size. The rate of increase is higher than the rate at which the number of actors allocated to the SPM increase as seen in Figure 5.4. This clearly shows that the optimization achieves it desired purpose in allocating only such actors to the SPM that maximize the number of accesses to the SPM while minimizing net memory access time.

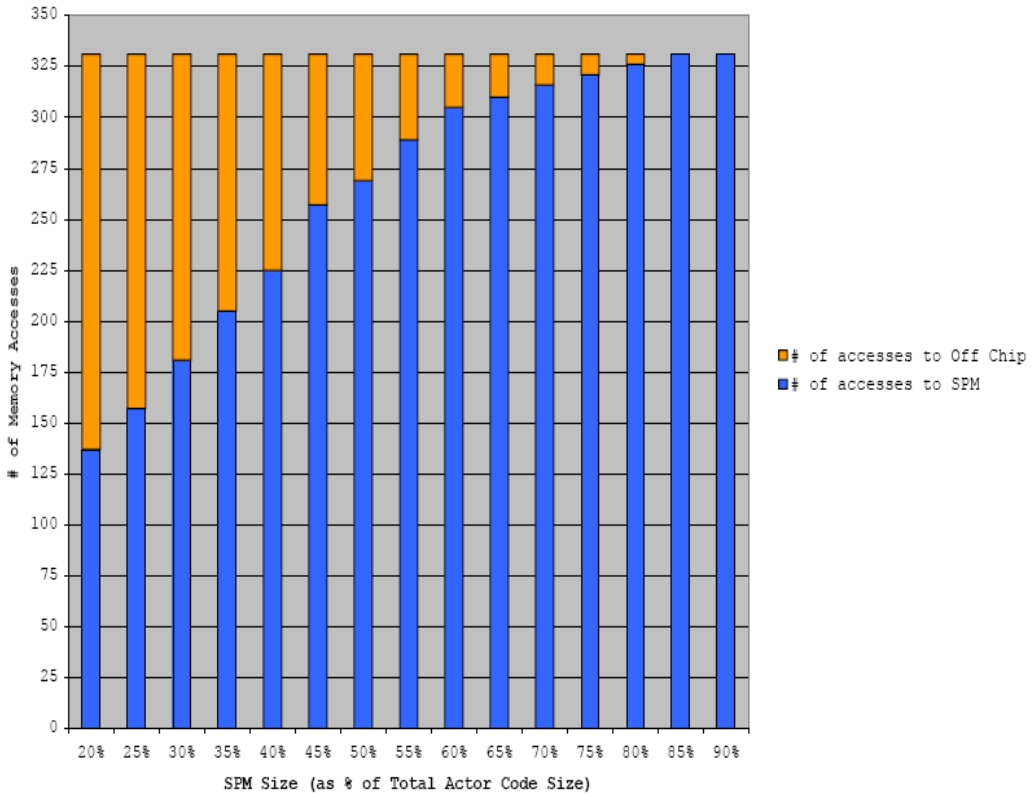

Figure 5.5 Number of Memory Accesses to SPM and Off-Chip for State 1 (7,4 Hamming Code)

Figure 5.7 highlights the trend shown in Figure 5.5 and Figure 5.6. It shows the percentage of memory accesses that hit the SPM with increasing SPM size. We can see that at a reasonable SPM size of 35-45% of total actor size, well over the 60-65% of all memory accesses hit the SPM. It can also be seen that the percentage of accesses to SPM is higher for State 1 than for State 2 for most SPM sizes. The reason for this is that in the schedule for State 1, several actors are fired multiple times. As a result, when a particular actor is allocated to the SPM, multiple firings of the actor ensure that a higher number of memory accesses hit the SPM. Since most actors in State 2 are fired a single time, there is reduced scope for the integer program to take advantage of multiple actor firings in computing the allocations. We can thus conclude that this aspect of the performance of the algorithm would be better for schedules in which actors are fired multiple times than for schedules in which most actors are fired a single time.

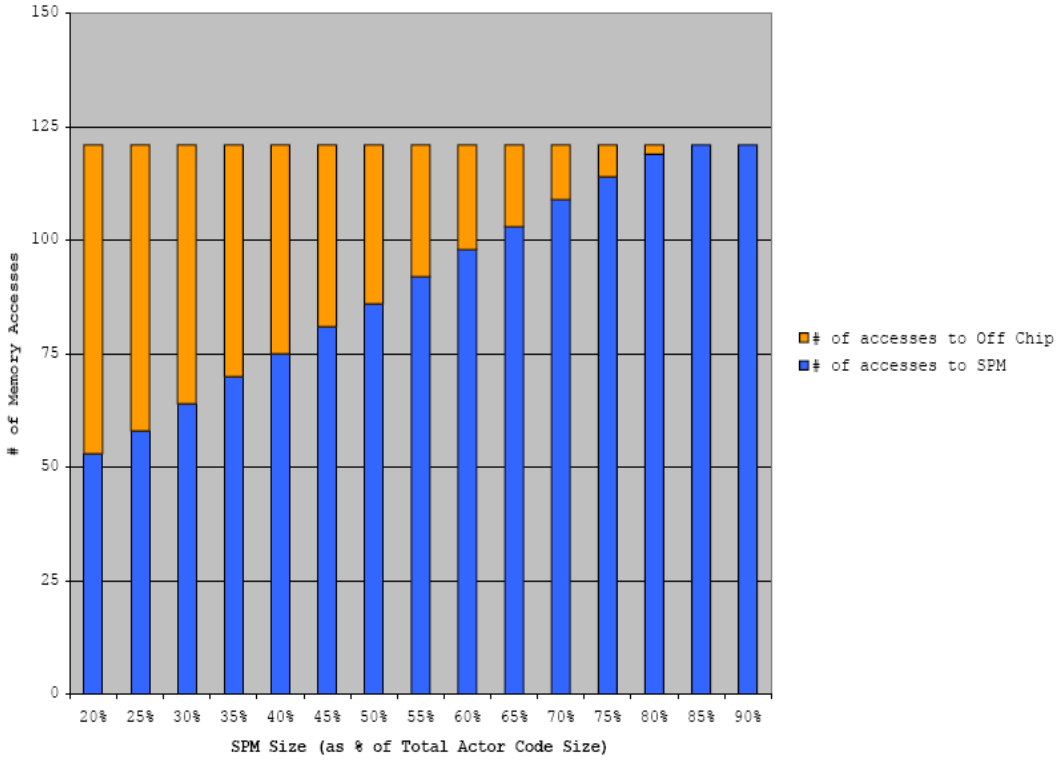

Figure 5.6 Number of Memory Accesses to SPM and Off-Chip for State 2 (3,1 Hamming Code)

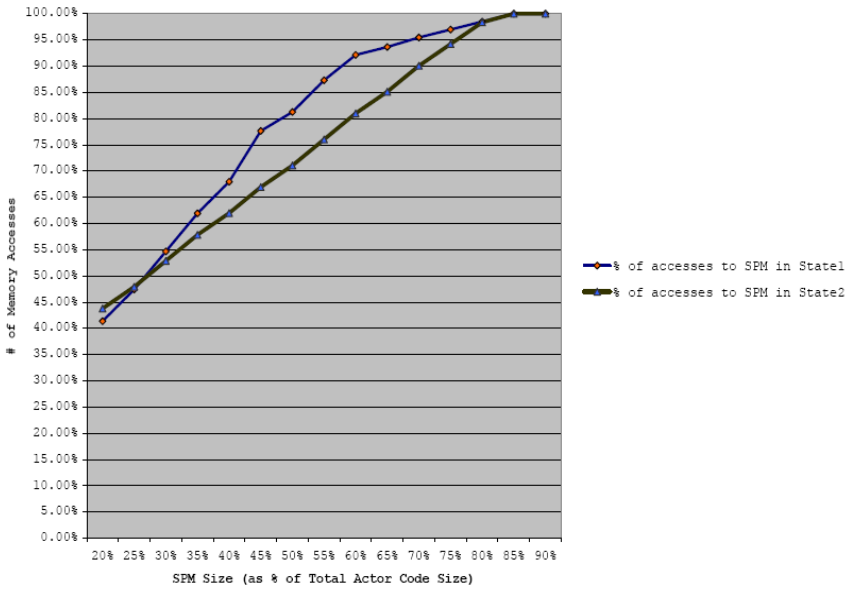

Figure 5.7 Percentage of Memory Accesses to SPM for State1 and State2

Figure 5.8 shows the total actor code memory access time for a single iteration of a particular state for both caches and SPM. It can be clearly observed that the memory access time for caches remains constant irrespective of the cache size. This is due to the fact that the only cache misses that we encounter are compulsory misses, i.e. the first reference to an actor code causes a cache miss. Thus the first firing of each actor causes cache misses, but all subsequent firings hit the cache. Since we do not encounter any conflict or capacity misses the total access time for caches remains constant irrespective of the actual cache size. It can be observed that the SPM allocation policy performs significantly better than caches. The memory access time for SPM for State 2 is lower than the cache for all memory sizes. However, for State 1 the SPM shows improvements over caches for sizes of about 40% and higher. This is because each actor is fired multiple times in States 1 hence the penalty for not being able to allocate such actors to the SPM is also significantly higher. It should be noted that while it might appear that the scratchpad allocation scheme also encounters compulsory misses especially during the first iteration, this is not entirely accurate. Our scheme ensures that the scratchpad

memory is preloaded with the appropriate actors prior to execution of an iteration of the HDF model. Hence such misses in which actors have to be brought into the fast memory during the course of an iteration, do not occur in the case of scratchpad memories.

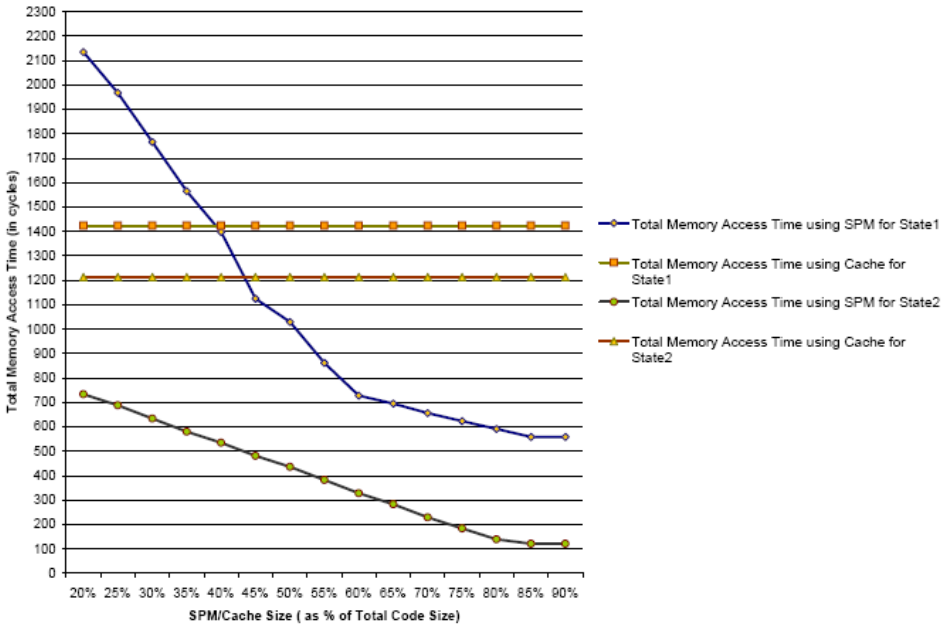

Figure 5.8 Total Memory Access Time for a single iteration of a particular state for SPM and cache configurations

One key point to note is that the total memory access time for caches is highly dependent on the firing schedule of the actors in a particular state. We have used a single appearance schedule for each state, in which each actor is fired in succession its required number of times. Consider the single appearance and an alternative schedule for State 1:

Single Appearance Schedule: (4A)(B)(G)(7C)(7D)(I)(4K)(4L)(4M)(4N)(4O)(4E)(4F)

Alternative Valid Schedule: (2A)(B)(3C)(A)(3C)(A)(C)(7D)(I)(4(KLMNOEF))

For the alternative schedule, the actor firings are not always in succession. As can be seen clearly, dependent on the size of the cache there might be significant conflict misses if executing this schedule. Consider the following portion of the schedule

(4(KLMNOEF)) with a cache large enough to store 4 actors at a time. Actors K, L, M, N, O, E, and F shall encounter compulsory misses on their first firing. By the time actor F completes its first firing, the actors present in the cache will be N, O, E, and F, assuming a LRU replacement policy. Hence, when actor K is fired a second time, it will cause a cache miss. The same shall happen for all the above actors for all four firings, leading to a large cache miss penalty. Thus we would encounter a cache trashing situation, in which a particular cache block is repeatedly evicted and then brought back into the cache again. It can also be seen that the single appearance schedule provides the best case access time for the use of a cache. Hence our choice of using single appearance schedules for performance comparison. The total memory access time for the SPM allocation algorithm is completely independent of the firing schedule. The time would be the same for both the schedules given above, for a particular SPM size. Hence, the SPM allocation algorithm provides us with much greater predictability than the use of a cache. Moreover, it is completely independent of the scheduling algorithm used.

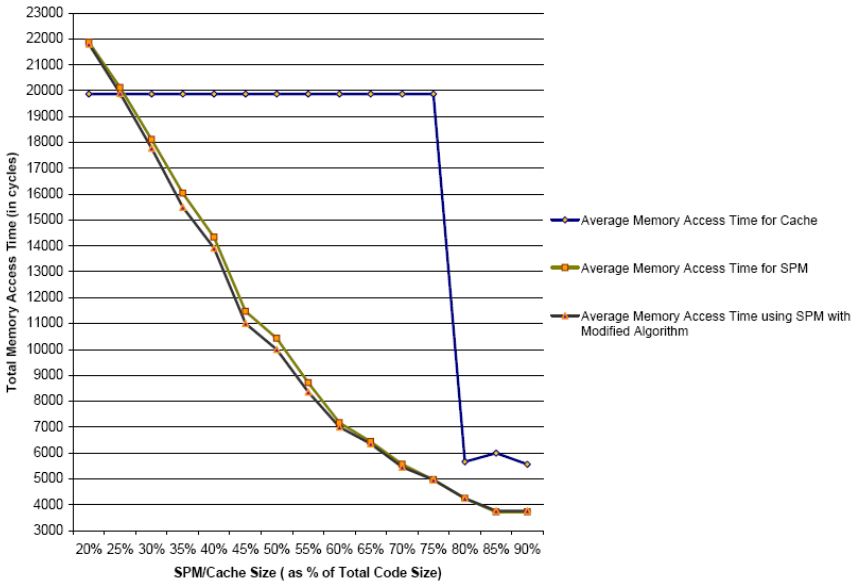

Figure 5.9 Average Memory Access Time for Actor Code for a fifteen iteration execution of the Adaptive Coding Model

Twenty-five execution traces of the Adaptive Coding Model were generated using the Ptolemy II [7] environment. Each of these traces was fifteen iterations long. The total memory access times for actor code access for these traces were computed and averaged over the executions to generate the average memory access time for a fifteen iteration long execution of this model. Figure 5.9 shows the comparative graph for the average memory access times using caches, the basic allocation algorithm and the multiple allocations per state modification to the original algorithm. It can be observed that both the SPM allocation algorithms outperform the cache for all memory sizes above 25%. The graph for the access time for the cache shows two distinct plateaus. The higher plateau indicates the situation where the cache encounters capacity misses across state transitions. When the HDF model transitions from one state to another, the actors from the predecessor state need to be evicted to make space for actors in the successor state. The lower plateau represents the situation where the cache is large enough to store actors that get reused across state transitions. Thus, when the model transitions from one state to the next the cache is large enough that none of the actors from the predecessor state need to be evicted. All actors that are common to both the predecessor and successor states do not cause any further cache misses. Hence, the only misses are caused by the compulsory misses on the initial firings of the actors and replacement misses for actors not common to both states.

We can also observe that for intermediate memory sizes, the multiple allocations per state perform better than the original allocation scheme. This is because the multiple allocation scheme reduces the number of actors that need to be transferred into the SPM across state transitions. Thus the time spent in migration of actors from off-chip memory to SPM is reduced. For small and large memory sizes the multiple allocation scheme fails

to outperform the original scheme. When the memory size is small, the number of actors assigned to off-chip memory is large. Hence, migration costs remain high due to the small number of actors that can be fit into the SPM, irrespective of the algorithm used. In other words, the SPM size is too small for the multiple allocation scheme to provide any reduction in migration cost. On the other hand, for large memory sizes, most actors are already in the SPM and hence migration costs are low already. Thus, the multiple allocation scheme is unable to extract any further reduction in migration costs for this example.

5.1.2 Performance Analysis for Data Buffer Allocation for the Adaptive Coding Model

The performance results for the allocation of data buffers to scratchpad memory show a trend that is very similar to that seen in the performance results for actor allocation. One key point to note is that the allocation of data buffers to the scratchpad involves no migration overhead during state transition. This is because there are no delay tokens to be preserved across state transitions in this example. More generally, the total number of delay tokens present in a HDF model is often quite small and hence migration costs can be considered negligible in comparison with the migration costs of actor allocation.

The trends seen in Figures 5.4, 5.5, 5.6 and 5.7 for actor allocation are very similar to the trends observed in data buffer allocation. Hence, for sake of brevity, the corresponding graphs have not been presented for data buffer allocation. Figure 5.10 shows the total memory access time for data buffer access for a single iteration of a particular state for both caches and SPM. It can be observed that the scratchpad allocation scheme performs better than caches for both State 1 and State 2. For State 1, performance improvements over caches are seen with memory sizes as small as 30% of

total data buffer size. For State 2, improvements are noted for all memory sizes. Data buffer access is a two step process involving the writing of tokens to the data buffer by the source actor and the reading of data tokens by the sink actor. Actor code access, on the other hand involves only reads. Hence, the cache access patterns are more complicated with potential for misses both during read and write. A general example of this cache access pattern can be found in Chapter 2 of [10]. Unlike the actor allocation scenario, both compulsory and capacity misses are encountered by caches in this situation. All initial writes to a data buffer encounters write misses, which are compulsory misses. When the sink actors attempt to read the tokens from the buffer, read misses may also be encountered. These read misses are capacity misses. This is the reason for the decrease in memory access time for caches with increase in cache size.

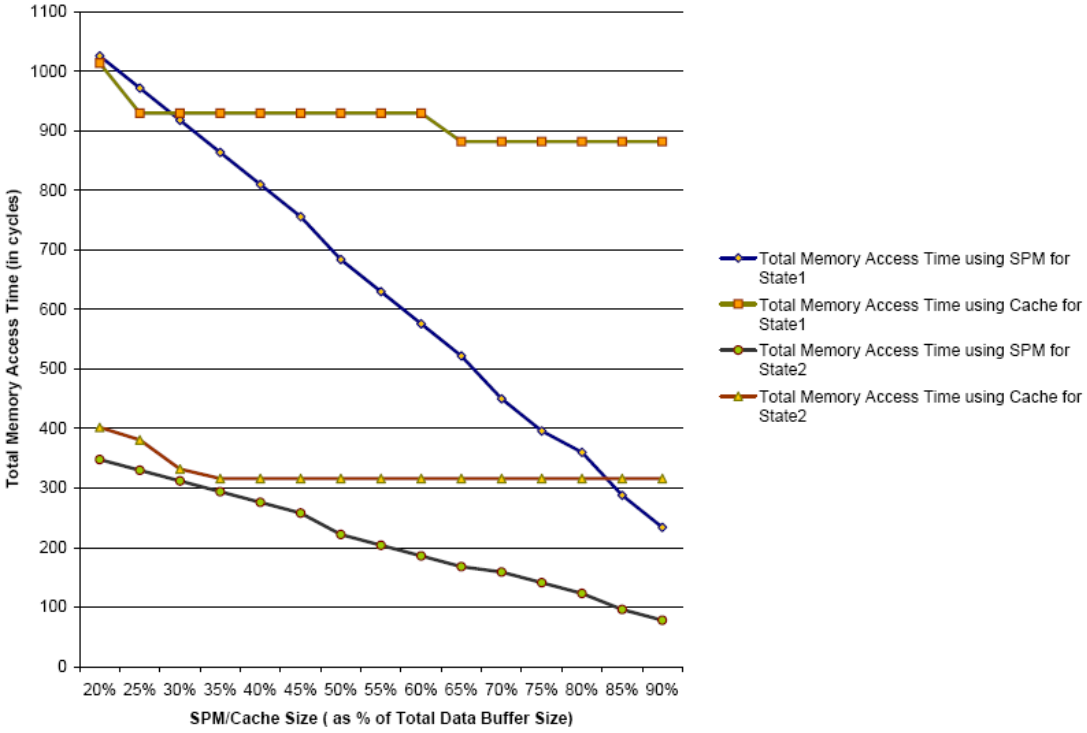

Figure 5.10 Total Memory Access Time for a single iteration of a particular state for SPM and cache configurations

The fact that cache misses may be encountered both on reads and writes, greatly increases the dependency of the cache access times on the scheduling algorithm and firing schedule for the HDF model. The cache access times for the very same HDF model scheduled using different algorithms might be vastly different. It should be noted that the scratchpad allocation scheme also fails to produce complete independence from the firing schedule, unlike the case for actor allocation. This is because the data buffer sizes themselves depend on the firings schedule. Different schedules lead to different data buffer sizes for each connecting channel between actors. This is a property of HDF/SDF itself. However, our scratchpad algorithm does offer access time predictability for a given schedule and the corresponding set of data buffer sizes.

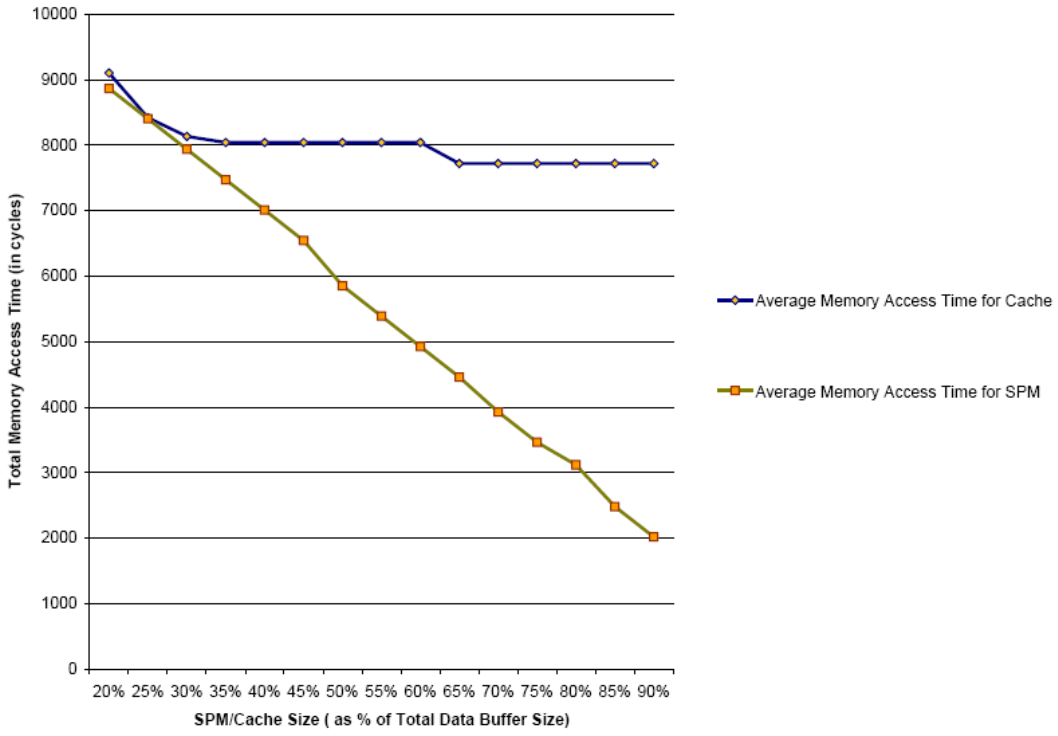

Figure 5.11 Average Memory Access Time for Data Buffer for a fifteen iteration execution of the Adaptive Coding Model

Figure 5.11 shows the comparative graph for average memory access times for data buffer access for a fifteen iteration execution of the Adaptive Coding Model, for

both cache and scratchpad allocation. The data for the above graph has been generated using the same procedure used to generate Figure 5.9. It can be observed that the scratchpad allocation scheme performs consistently better than caches for all memory sizes. It should be noted that unlike actor allocation, there is no migration cost involved in this case. Hence, the total memory access time for data buffer access for *N* iterations of a HDF model is essentially the sum of the per state memory access times of each state encountered during the iterations. This is also true for caches, since there is no preserved state to be maintained across state transitions, in the absence of delay tokens.

5.2 Randomly Generated HDF Graphs

To further analyze the performance of our algorithm, we applied our allocation scheme to a set of 50 randomly generated HDF graphs, and compared its performance to caches. In order to randomly generate the HDF graphs, we used several basic model templates which included the adaptive coding model, and the models shown in Figures 1.4 and 1.5. The production and consumption rates of the actors were chosen randomly from an interval of 1 to 10. Actor sizes we also chosen randomly between 5 and 40. We considered single appearance schedules for our analysis, since this provides the best cache performance for a LRU replacement policy, as has been discussed in Section 5.1.1. Memory sizes for both scratchpad and caches were set at 35% of the total actor/data buffer sizes. The procedure used to generate the average memory access times for comparison was identical to the procedure used for Figures 5.9 and 5.11.

For the actor allocation algorithm, the modified actor allocation algorithm and the data allocation algorithm, we calculated the percentage improvement in memory access time, i.e. the percentage reduction in memory access time as a result of selecting our

algorithm over caches. The percentages for improvements were computed over the memory access times provided by our algorithm. The percentages for deteriorations were calculated over the memory access times for caches.

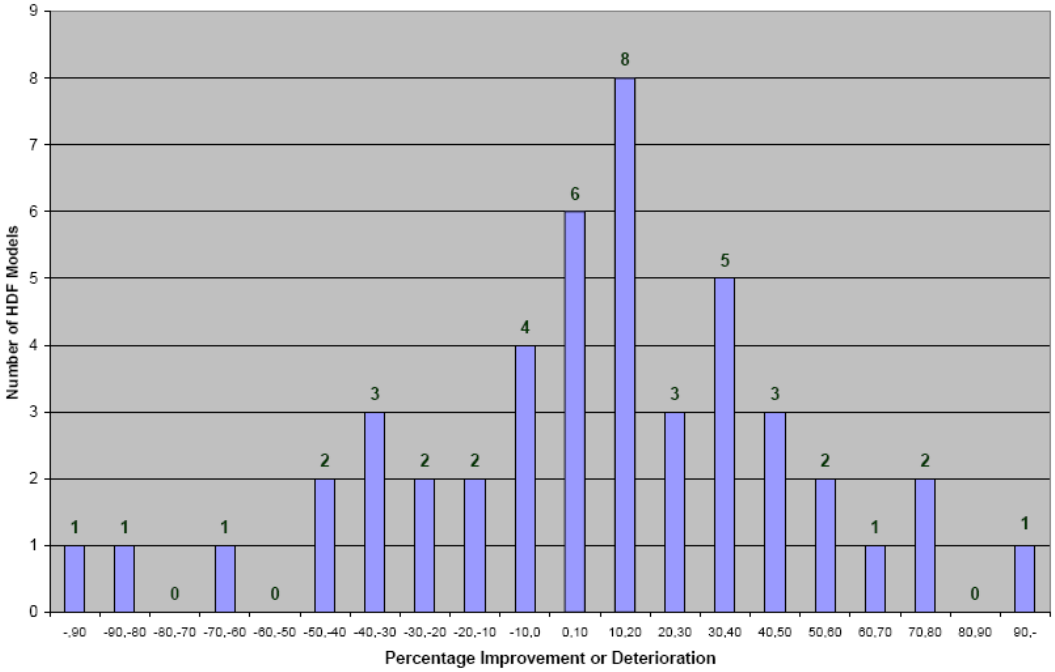

Figure 5.12 Performance Improvement/Deterioration of Actor Allocation to SPM versus cache for 50 randomly generated HDF graphs

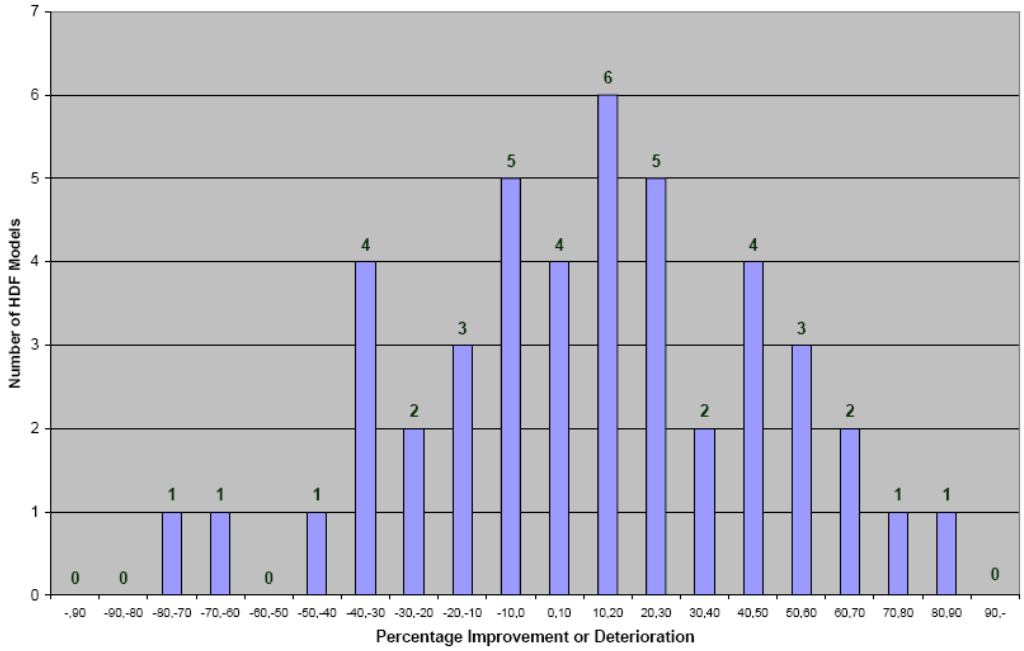

Figure 5.13 Performance Improvement/Deterioration of Actor Allocation to SPM using modified algorithm versus cache for 50 randomly generated HDF graphs

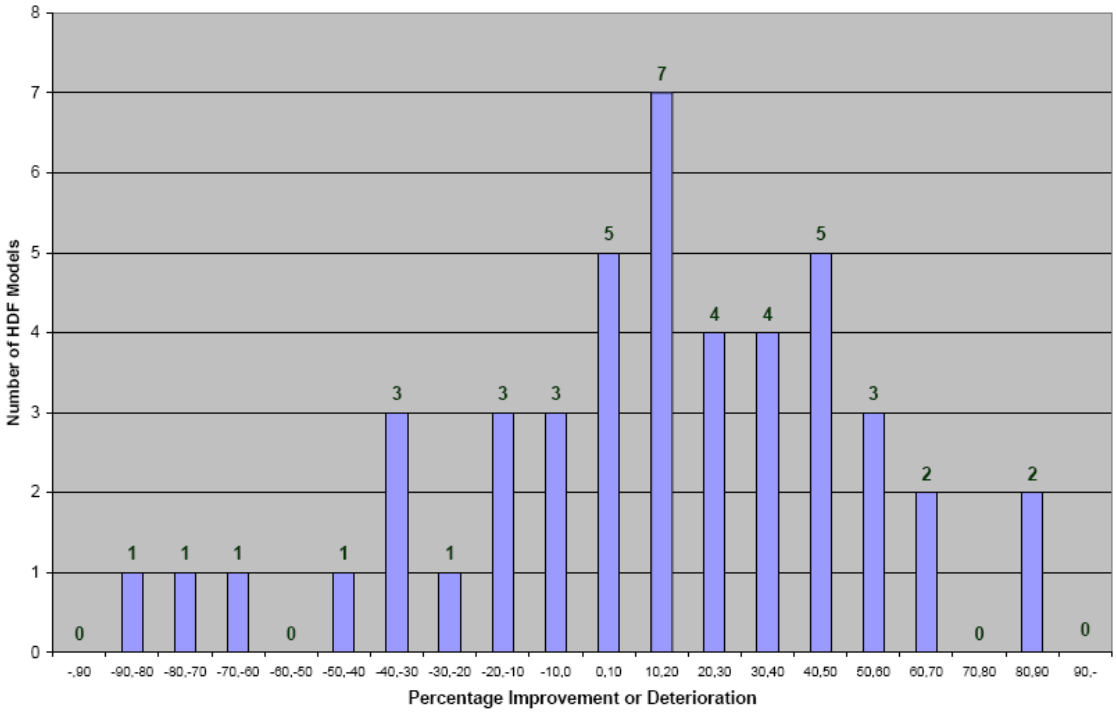

Figure 5.14 Performance Improvement/Deterioration of Data Buffer Allocation to SPM versus cache for 50 randomly generated HDF graphs

Figures 5.12, 5.13 and 5.14 show the performance improvement/deterioration for use of our algorithm for actor allocation, the modified algorithm for actor allocation and our algorithm for data allocation, respectively. There was no performance improvement or deterioration observed for 3, 5 and 4 of the 50 randomly generated HDF graphs for Figures 5.12, 5.13 and 5.14 respectively. For Figures 5.12, 5.13 and 5.14, 16, 17 and 14 of the graphs the showed a deterioration in performance, respectively. The average performance improvement for actor allocation was 13.43%. Using the modified algorithm resulted in a 15.64% average performance improvement. Thus a 2.21% advantage was noted in average performance by use of the modified multiple allocations per state algorithm. The data buffer allocation showed an average performance improvement of 17.24%. The fact that data buffer allocation performed better than actor allocation can be attributed to the potential for both capacity and compulsory cache misses as observed in

Section 5.1.2. It should be noted that the improvements observed here have been for single appearance schedules. As discussed earlier, use of alternative schedules would have potentially resulted in increased memory access times for caches, leading to greater improvements in comparative performance.

5.3 ILP Runtime Analysis

The most computationally expensive stage of our allocation scheme is the integer linear program. It forms the core of both the original allocation algorithm and the multiple allocations per state modification. Hence, it is critical to analyze the execution time of the ILP program and assess its feasibility for HDF models of varying sizes. In order to perform the execution time analysis, ILP programs of different sizes were randomly generated. The objective function and constraints were set up to represent actor/data buffer allocation problems of varying sizes. The generated ILP programs were solved using the open-source LP Solve package [23]. The computer used for the generating the execution times had a 1.86GHz Intel Pentium M processor with 1.49GB of RAM. Figure 5.15 shows the execution time graph for increasing linear program size on a per state basis.

As can be observed, even for models with 850 actors/data buffers, the execution time of the ILP solver is well below 0.5 seconds. It is only when we reach about a 1000 actors/data buffers, that a sharp exponential growth is observed. It can be safely assumed that most practical HDF models would be limited to at most a couple of hundred actors/data buffers. Thus, for all practical HDF models the execution times would be below 0.2 seconds. We can therefore conclude that the ILP solver execution time shall clearly be feasible and acceptable for all practical HDF models.

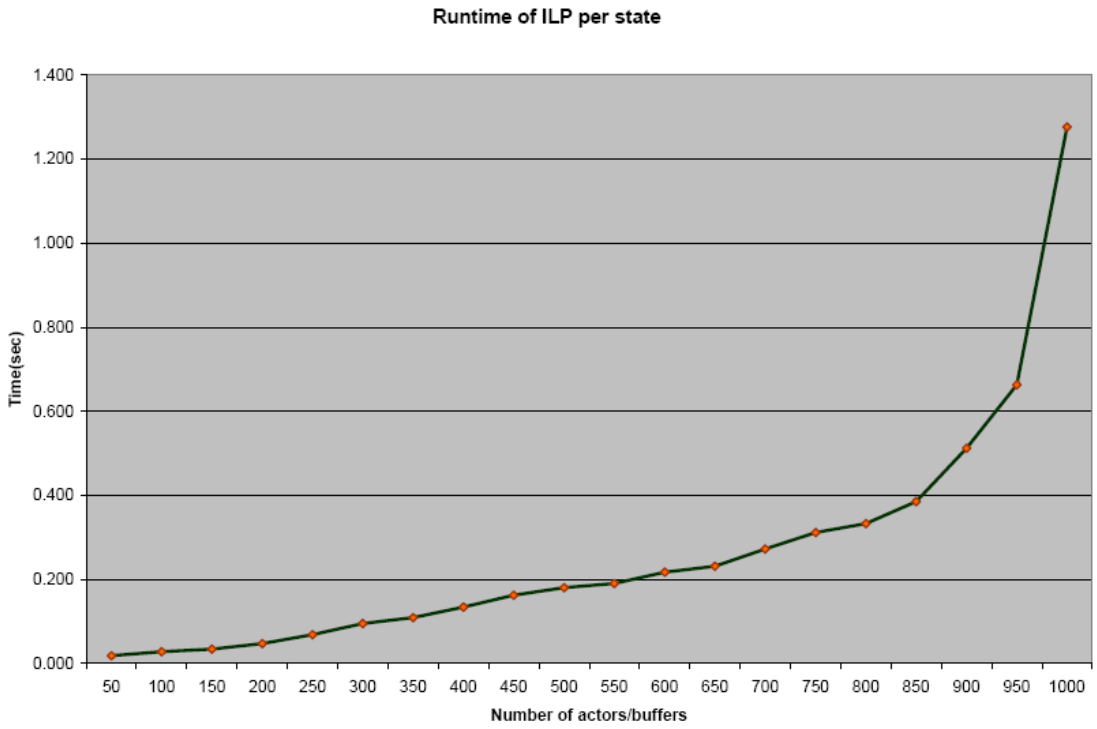

Figure 5.15 Execution Times for Linear Programs of Increasing Size

5.4 Worst Case Access Time

As stated earlier, the memory access time for the scratchpad allocation algorithm is independent of the scheduling algorithm used to generate the firing schedule for the HDF models. This lends a high degree of predictably to the scratchpad allocation scheme as compared to caches. In spite of the fact that the exact sequence of state transitions in the execution of a HDF model is not known *a priori*, it is possible to compute a strict upper bound on the total memory access time for the execution of a HDF model for a finite number of iterations. Given that the allocation for each state is generated by our algorithm, the memory access time for accessing the actor code for a particular state is:

$$M_{OffChip}(a_i)\left(T_{OffChip_{Rd}} \cdot F(a_i) \cdot S(a_i)\right) + M_{SPM}(a_i)\left(T_{SPM_{Rd}} \cdot F(a_i) \cdot S(a_i)\right)$$

Let $T(S)'$ be the greatest per state memory access time amongst all the states of the HDF model. The migration cost of moving an actor block into the scratchpad during a state transition can be computed simply by using *Observation 3, Sec. 3.4,* on all the actors that need to be moved for that particular state transition. Let $T_{migration}(S)'$ be the greatest migration cost for a single state transition amongst all possible state transitions. For an *N* iteration execution of a HDF model there are *N* states and *(N-1)* state transitions. Then clearly the upper bound on the total actor memory access time for an *N* iteration execution of a HDF model is:

$$N \bullet T(S)' + (N-1) \bullet T_{migration}(S)'$$

This is a strict upper bound that is not reachable. In other words, the actual actor memory access time shall always be less than the above expression. The reason for this is that, if a HDF model stayed in the state with the most expensive memory access time for its entire execution, then the migration cost would be zero at each state transition since the predecessor and successor states would always be the same. Thus the total actor memory access time is strictly less than the above expression.

The upper bound on the memory access time for data buffers can be computed in a similar fashion. The two bounds can be combined to form the upper bound for the overall memory access time of the model.

6. Conclusion and Future Work

In conclusion, we have developed an ILP based allocation algorithm that makes use of the coarse grained software architecture and execution flow information present in HDF block diagram programs to generate a state-wise optimal memory allocation for scratchpad memories. We have also provided generalizations of our allocation algorithm to allow for multiple memory hierarchies and energy based optimization. We have shown our method to perform better than caches in total memory access time. We have also shown that our method and the use of scratchpad memories offers greater predictability, independence from scheduling algorithms, and the ability to compute an upper bound on the memory access times, all of which are not possible or not useful for caches. We have established the feasibility of our linear programs with respect to solver runtimes. We have also provided a modification to our algorithm which generates multiple allocations per state. While not being truly optimal, the algorithm serves as a heuristic to lower memory access times for intermediate scratchpad sizes in most general cases. The modification also serves as a beacon for revealing a whole field of possible multiple allocations per state schemes that provide better performance.

The above project provides enormous scope for future exploration. Some possible directions of exploration are as follows:

1. Exploring the concept of partitioning the scratchpad memory into partitions that correspond to different levels in the hierarchy of the HDF model. Allocations schemes would have to aim at making the best use of these partitions to store appropriate actor and data buffers from different levels of the HDF model hierarchy without attempting to flatten the entire model.

2. Exploring the possibility of using profiling in computing the probabilities of the various possible state transitions. The probabilities thus computed would prove useful in attempting to formulate allocations schemes that generate multiple allocations for each state based on these transition probabilities.

3. Attempting to find methods that optimize memory allocations over a sequence of states rather than a single state as in the current case. Such methods might make use of the transition probabilities mentioned above to optimize over whole sections of the HDF execution trellis diagram, Figure 3.1, rather than for a single state. Another possibility would be perform window optimizations that generate optimal memory allocations for every n-tuple of states that have a high probability of occurring in a sequence.

4. Energy and area based optimization schemes provide yet another potential direction of future exploration.

5. Exploring allocation algorithms that are aimed specifically at optimizing data buffer allocation, while intimately accounting for the dependence of data buffer sizes on the scheduling algorithm and firing schedule.

6. Quasi Static Scheduling techniques and schedulability analysis of non deterministic branches introduced in [26] provides interesting possibilities to allow us to make optimization decisions across non deterministic state transition boundaries. The above schedulability analysis techniques have been applied to data dependent branches modeled as non-deterministic free choices in Petri nets. This is similar to our scenario where state transitions are data dependent. A suitable amalgamation of our memory allocations techniques with the concepts introduced in [26] might provide means of improving memory allocations by optimizing across state transitions.

Bibliography

[1] E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow”, *Proceedings of the IEEE*, vol. 75, no. 9, September, 1987.

[2] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Transactions on Computers*, vol. C-36, No. 1, Jan. 1987.

[3] A. Girault, B. Lee, and E. A. Lee, “Hierarchical Finite State Machines with Multiple Concurrency Models,” *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 18, No. 6, June 1999.

[4] E. A. Lee and T. M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995.

[5] P. Murthy, S. S. Bhattacharyya and E. A. Lee, “Joint Minimization of Code and Data for Synchronous Dataflow Programs,” *Journal of Formal Methods in System Design*, vol. 11, No. 1, July 1997.

[6] Shuvra S. Bhattacharyya, "Compiling Dataflow Programs for Digital Signal Processing," Ph.D. Thesis, *Tech. Report UCB/ERL 94/52*, Dept. of EECS, University of California, Berkeley, CA 94720, July 12, 1994.

[7] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.), "Heterogeneous Concurrent Modeling and Design in Java: Ptolemy II", Vol 1-3 *Technical Memorandum UCB/ERL M05/21 – M05/23*, University of California, Berkeley, CA USA 94720, July 15, 2005.

[8] J.L. Henessey and D.J. Patterson, "Memory Hierarchy Design," *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufmann Publishers, 2003.

[9] R. Banakar, S. Steinke, Bo-Sik Lee, M, Balakrishnan, P. Marwedel, "Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems," *Proceedings of CODES*, pp. 73-78, Estes Park, Colorado, May 2002.

[10] S. Kohli, "Cache Aware Scheduling for Synchronous Dataflow Programs," Master's Report, *Technical Memorandum UCB/ERL M04/03*, University of California, Berkeley, CA 94720, February 23, 2004.

[11] Ye Zhou, "Communication Systems Modeling in Ptolemy II," Master's Report, *Technical Memorandum No. UCB/ERL M03/53*, University of California, Berkeley, CA, 94720, USA, December 18, 2003.

[12] Y.-J. Chang, S.-J. Ruan, and F. Lai, "Design and analysis of low-power cache using two-level filter scheme", *IEEE Trans. Very Large Scale Integrated Systems*, 11(4):568–580, 2003.

[13] S. Wilton and N. Jouppi, “Cacti: An enhanced cache access and cycle time model”, *IEEE Journal of Solid-State Circuits*, May 1996.

[14] S. Steinke, L. Wehmeyer, B. S. Lee, P. Marwedel, “Assigning Program and Data Objects to Scratchpad for Energy Reduction”, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, March 2002.

[15] M. Verma, L. Wehmeyer, P. Marwedel, “Cache-Aware Scratchpad Allocation Algorithm”, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, February 2004.

[16] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, M. Olivieri, “A Post-Compiler Approach to Scratchpad Mapping of Code”, *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, September 2004.

[17] S. Udayakumaran and R. Barua, “Compiler-decided dynamic memory allocation for scratch-pad based embedded systems”, *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2003.

[18] G. Hallnor and S. K. Reinhardt, “A fully associative software-managed cache design”, *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[19] O. Avissar, R. Barua and D. Stewart, “An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems”, *ACM Transactions on Embedded Computing Systems,* Vol. 1, No. 1, pp. 6-26, November 2002.

[20] H. Falk and M. Verma, “Combined Data Partitioning and Loop Nest Splitting for Energy Consumption Minimization”, *Proceedings of Software and Compilers for Embedded Systems,* 8th International Workshop, SCOPES 2004, Amsterdam, The Netherlands, September 2-3, 2004.

[21] A. Dominguez, S. Udayakumaran and R. Barua, “Heap Data Allocation in Embedded Systems”, *Journal of Embedded Computing,* 1(4), 2005.

[22] S. S. Bhattacharyya. “Compiling Dataflow Programs for Digital Signal Processing,” PhD Thesis, *Technical Memorandum UCB/ERL 94/52*, University of California, Berkeley, CA 94720, July, 1994.

[23] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve: Open source (Mixed-Integer) Linear Programming System. *Version 5.1.0.0.,* May, 2004.

[24] M-CORE - MMC2001 Reference Manual. Motorola Corporation, 1998. (A 32-bit processor).

[25] D.Popov, E. Foutekova, I. Delchev, I. Krivulev, Z. Kochovski. Lectures PC 101 Seminar. Cache Memory: Implmentation and Design Techniques. Web Resource: *http://www.faculty.iu-bremen.de/birk/lectures/PC1012003/07cache/cache%20memory.htm*

[26] C. Liu, "Compile-Time Schedulability Analysis of Communicating Concurrent Programs," PhD Thesis, *Technical Report UCB/EECS-2006-94*, University of California, Berkeley, CA 94720, June 2006.