Cache Aware Scheduling for Synchronous Dataflow Programs[1]

by Sanjeev Kohli

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

[1] February 23, 2004
Technical Memorandum UCB/ERL M04/03
University of California, Berkeley, CA 94720

Abstract

The Synchronous Dataflow (SDF) model of computation [1] is an efficient and popular way to represent signal processing systems. In an SDF model, the amount of data produced and consumed by a data flow actor is specified *a priori* for each input and output. SDF specifications allow static generation of highly optimized schedules, which may be optimized according to one or more criteria, such as minimum buffer size, maximum throughput, maximum processor utilization, or minimum program memory. In this report, we analyze the effect of cache architecture on the execution time of an SDF schedule and develop a new heuristic approach to generating SDF schedules with reduced execution time for a particular cache architecture.

In this report, we consider the implementation of *well-ordered* SDF graphs on a single embedded Digital Signal Processor (DSP). We assume a simple Harvard memory architecture DSP with single-level caches and separate instruction and data-memory. In order to predict execution times, we propose a cache management policy for the data cache and argue that this policy outperforms traditional cache policies when executing SDF models. We also replace the instruction cache by a scratchpad memory with software-controlled replacement policy. Using our data cache and instruction scratchpad policies, we show that different schedules can have vastly different execution times for a given set of data cache and instruction scratchpad sizes. In addition, we show that existing scheduling techniques often create schedules that perform poorly with respect to cache usage.

In order to improve cache performance, an optimal cache-aware scheduler would minimize the total cache miss penalty by simultaneously considering both data and instruction miss penalties. Unfortunately, reducing data cache misses often increases instruction scratchpad misses and vice versa. In this report, we show that the number of schedules that must be considered increases exponentially according to the vectorization factor of the schedule. To address this complexity, we develop an SDF scheduling algorithm based on a greedy, cache-aware heuristic. We compare the resulting schedules with schedules generated by existing SDF scheduling schemes. The schedule generated by our algorithm poses an interesting problem of code generation. We also propose a solution to address this problem.

This work is highly applicable in the design of SDF systems that are implemented as Systems on Chip (SoC) with DSP cores.

Acknowledgements

I would like to thank my research advisor, Professor Edward A. Lee, whose support and guidance have made this report possible. His consistent encouragement and valuable advice helped me finish this report in a timely manner.

I would also like to thank all the members of the Ptolemy group for creating such a friendly and conducive research environment. In particular, I would like to thank Steve Neuendorffer for his valuable suggestions towards various aspects of this work.

Also, I am grateful to Krishnendu Chatterjee and Satrajit Chatterjee for various fruitful discussions we had during the course of this work.

I would also like to thank Professor Alberto Sangiovanni-Vincentelli for taking time out of his hectic schedule to serve as the second reader of this report.

Finally I would like to thank my family members and friends who motivated me to take up graduate studies and continue to be my pillar of strength. I dedicate this report to them.

Contents

1. Introduction
    - 1.1 Synchronous Dataflow (SDF) Graphs
        - 1.1.1 Chain-Structured SDF Graphs
        - 1.1.2 Well-Ordered SDF Graphs
    - 1.2 Scheduling Schemes for SDF Graphs
        - 1.2.1 Minimum Buffer Schedules
        - 1.2.2 Minimum Program Memory Schedules
        - 1.2.3 Minimum Program Memory – Minimum Buffer Schedules
        - 1.2.4 Single Appearance Minimum Activation Schedules
2. Motivation
3. Memory Architecture
    - 3.1 Data Cache and Instruction Scratchpad Policies
4. Cache Aware Scheduling – Complexity Analysis
5. Heuristic – Cache Aware Scheduling (CAS)
6. Performance Analysis
    - 6.1 Code Generation
    - 6.2 Extensions
7. Conclusions and Future Work

1. Introduction

1.1 Synchronous Dataflow (SDF) Graphs

Dataflow models of computation [2] represent concurrent programs through interconnected components called *actors*. The behavior of each actor consists of a sequence of invocations or *firings*, during which the actor can perform computation and communicate data *tokens* with other actors. Synchronous dataflow (SDF) [1] is a dataflow model of computation where actors communicate through FIFO queues and the number of data samples produced or consumed by each node on each invocation is specified *a priori*. An SDF model is represented by 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 communication buffer. In actor-oriented design [3], each node of an SDF graph can be hierarchically refined and incident edges on a node correspond to actor ports. Figure 1 shows an acyclic SDF graph containing 7 SDF actors, annotated with production and consumption rates.

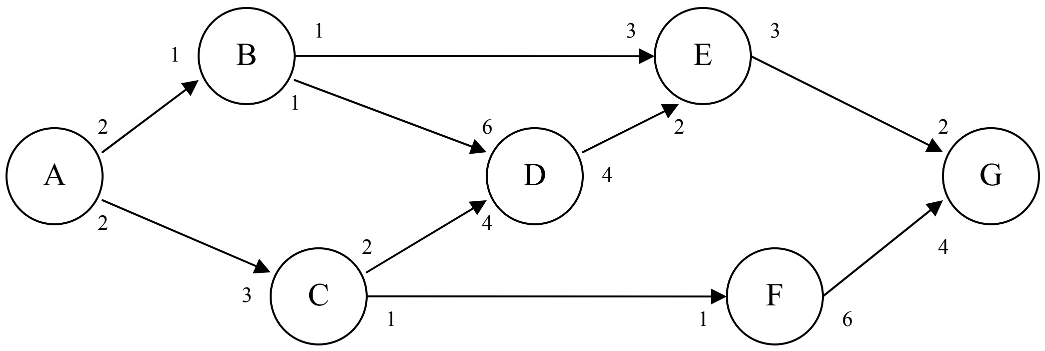

Figure 1 An SDF graph

Each edge in an SDF graph can also have an associated non-negative integer delay. Each unit of delay represents an initial token in a communication buffer. For clarity, we only consider SDF graphs with no edge delays, although SDF graphs with edge delays can be easily accommodated by the proposed scheduling algorithm.

In this report, we initially focus on the cache aware scheduling for a sub-class of well-ordered SDF graphs, called *chain-structured* SDF graphs. The results presented for chain-structured SDF graphs can be extended to the more general class of well-ordered graphs, but for clarity, we develop our techniques in the context of chain-structured graphs. The following subsections describe chain-structured and well-ordered graphs

1.1.1 Chain-Structured SDF Graphs

A chain structured graph is defined [4] as an m-vertex directed graph having (m-1) edges with orderings (v_1, v_2, …v_m) and (e_1, e_2, … e_{m-1}) for the vertices and edges, respectively, such that each e_i is directed from v_i to v_{i+1}. Figure 2 is an example of a chain-structured SDF graph.

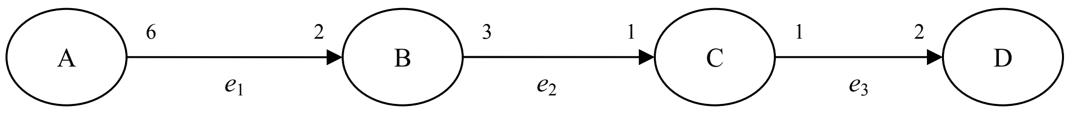

Figure 2 A chain-structured SDF graph

1.1.2 Well-Ordered SDF Graphs

A well-ordered graph is defined [4] as a directly connected graph that has only one ordering of the vertices such that for each edge *e*, the source vertex occurs earlier in the ordering than the sink vertex. Figure 3 is an example of a well-ordered graph.

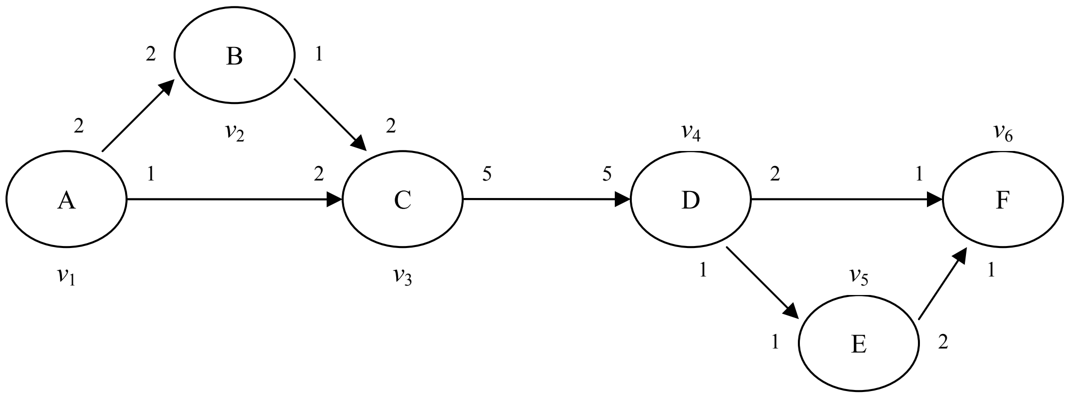

Figure 3 A well-ordered SDF graph

1.2 Scheduling Schemes for SDF Graphs

Lee and Messerschmitt [5] develop systematic techniques for basic compile-time scheduling of SDF graphs. An SDF schedule consists of a sequence of actor firings that can be executed forever without deadlock using a fixed amount of memory for communication. SDF schedules are based on a *minimal repetition vector* that determines the number of times each actor must fire. For example, the minimal repetition vector for the SDF graph shown in Figure 2 is $q^T = \{q_A, q_B, q_C, q_D\} = \{2, 6, 18, 9\}$, where q_A is the number of times actor A will fire. Although most schedules are based on the minimal repetition vector that fires each actor at least once, schedules can be constructed based on any positive integer multiple of the minimal repetitions vector. The multiplication factor *J*, also called a vectorization factor or blocking factor, can be chosen to exploit vector processing capabilities of DSPs, to reduce cache misses, or to generate efficient blocked schedules in multiprocessor environment [1]. In order to avoid confusion, we will always use q to represent the minimal repetition vector and explicitly show the vectorization factor of any schedule. The multiplication of minimal repetition vector with the vectorization factor results in a vector f which is called the *firing vector*.

A *periodic admissible schedule* or a *valid* schedule is a sequence of actor firings that fires each actor the number of times indicated in f, does not deadlock, and produces no net change in the number of tokens present on the edges [5]. We will represent schedules using the looped scheduling notation defined in [6]. Some example valid schedules for the model in Figure 2 are shown below.

S_1 = (A)(A)(B)(B)(B)(B)(B)(B)(C)(C)(C)(C)(C)(C)(C)(C)(C)(C)(C)(C)(C)(C)(C)(C)(C)
(C)(D)(D)(D)(D)(D)(D)(D)(D)(D)

S_2 = (2A)(6B)(18C)(9D)

S_3 = (2A)(6B)(9(2C)(D))

S_4 = (2(A)(3(B)(3C))(9D)

S_5 = (2(A)(2B)(3(2C)(D)))(2B)(6C)(3D)

S_6 = (A)(B)(2C)(D)(C)(B)(C)(D)(2C)(D)(B)(2C)(D)(C)(A)(B)(C)(D)(2C)(D)(B)(2C)(D)
(C)(B)(C)(D)(2C)(D)

S_7 = (2A)(3(2B)(3(2C)(D)))

Generally speaking, the above schedules represent different design tradeoffs and we would like to select an appropriate schedule for implementing a particular SDF graph based on the requirements of a particular implementation platform. For SDF graphs, completely exploring the design space is expensive, given the large number of schedules and complex implementation tradeoffs. Although some work has been done in automatic exploration of the design space, most research has focused on algorithms for generating optimal schedules based on certain implementation criteria. Optimal schedules can be generated for both single processor and multiprocessor environments [1]. However, we focus only on single processor schedules in this report. In the following subsections, we review some of the existing scheduling approaches.

1.2.1 Minimum Buffer Memory Schedules

One design criterion is often to reduce the memory usage of an SDF graph by choosing a schedule that reduces the memory buffer requirements. A *minimum buffer schedule* is a schedule whose maximum buffer requirement during execution is least among the set of all possible schedules. Bhattacharyya *et al* [6] explains how to calculate the buffer requirement of a given schedule and compute minimum buffer schedules. For the SDF graph shown in Figure 2, S_6 is one possible Minimum Buffer schedule. It has the buffer requirement of 8 units. In contrast, schedule S_5 requires 10 units of buffer memory and schedules S_1 and S_2 require 18 units of buffer memory. Table 1 shows the buffer activity profile [6] of the Minimum Buffer schedule S_6.

Buffer	A	B	2C	D	C	B	C	D	2C	D	B	2C	D	C	A
***e*₁**	6	4	4	4	4	2	2	2	2	2	0	0	0	0	6
***e*₂**	0	3	1	1	0	3	2	2	0	0	3	1	1	0	0
***e*₃**	0	0	2	0	1	1	2	0	2	0	0	2	0	1	1
Total	6	7	7	5	5	6	6	4	4	2	3	3	1	1	7

continued …..

Buffer	B	C	D	2C	D	B	2C	D	C	B	C	D	2C	D
***e*₁**	4	4	4	4	4	2	2	2	2	0	0	0	0	0
***e*₂**	3	2	2	0	0	3	1	1	0	3	2	2	0	0
***e*₃**	1	2	0	2	0	0	2	0	1	1	2	0	2	0
Total	8	8	6	6	4	5	5	3	3	4	4	2	2	0

Table 1 The buffer activity profile for the minimum buffer schedule of the SDF graph shown in figure 2

1.2.2 Minimum Program Memory Schedules

Another design criterion is to reduce the amount of program memory used to represent the SDF graph. Although there are several possibilities for generating code from a given SDF schedule, we will assume a simple code generation technique where each instance of an actor name in the schedule results in a copy of the actor code, and the repetition count of a schedule results in the generation of a loop [4] [6]. For instance, schedule S_5 is translated into the pseudo-code below.

```
for(i = 1 to 2) {
    A's code;
    for(j = 1 to 2) B's code;
    for(j = 1 to 3) {
        for(k = 1 to 2) C's code;
        D's code;
    }
}
for(i = 1 to 2) B's code;
for(i = 1 to 6) C's code;
for(i = 1 to 3) D's code;
```

The program requirements for this schedule is approximately *size*(*A*) + 2**size*(*B*) + 2**size*(*C*) + 2**size*(*D*), if we assume the actor's code is large relative to the loop code. The traditional technique works well for relatively simple schedules, or when actors are fine-grained, and contain relatively less instances of actors' names. Also, this technique allows specialization of each copy of an actor's code. An alternative is to replace each copy of the actor's code with a subroutine call to a single copy of the actor code. The use of subroutines is preferable for coarse-grained actors where the overhead of the jump is small relative to the cost of duplicated program code. From the point of view of cache usage, code generation of schedules using subroutine calls will usually result in better cache utilization as well.

A *minimum program memory schedule* is defined as a schedule that has the least program memory requirement among all possible admissible schedules. This schedule is often called a *single appearance* schedule, since if we ignore loop overhead, each actor can only appear once in the minimum program memory schedule. Bhattacharyya *et al* [6] give an algorithm for computing single appearance schedules. For the graph in Figure 2, S_2, S_3, and S_4 are all Minimum Program Memory schedules.

1.2.3 Minimum Program Memory – Minimum Buffer Schedules

In general, there are often many minimum buffer schedules and minimum program memory schedules. However, minimum buffer schedules are not generally minimum program memory schedules, and minimizing either the buffer memory or program memory individually rarely results in a schedule with minimal total memory requirements. To address this difficulty, Murthy *et al* [4] develop techniques for finding the minimum program memory schedule with the minimum buffer memory requirement. This schedule is called the *minimum program memory-minimum buffer schedule* (MPM-MBS). The MPM-MBS for our example is schedule S_7. In examining this execution of this schedule closely, we see that it generally reduces data-memory usage and results in good data locality of reference. Both of these properties are likely to result in a data cache that performs well. In the schedule this is achieved by switching rapidly between different actors. However, although this execution pattern is likely to perform well with respect to data caches, it is likely to generate many instruction cache misses.

1.2.4 Single Appearance Minimum Activation Schedules

Alternatively, we can consider schedules that perform well with respect to instruction caches and poor with respect to data caches. Sebastian Ritz *et al* [7] characterize a switch from firing one actor to firing another in a schedule as an actor *activation*. They give an algorithm for finding the *single appearance-minimum activation schedule* (SAMAS) for a given SDF graph. Intuitively, this schedule is likely to provide good instruction locality of reference since it repeatedly executes the same actor's code, as long as that code fits into the instruction cache. For the SDF graph of Figure 2, schedule S_2 is the SAMAS. In general, the SAMAS requires higher buffer size compared to some other classes of schedules; however, the buffer requirement can be kept reasonable when it can be shared by different signals of a block diagram [8].

Although the previous two scheduling approaches do not explicitly address cache miss penalties, we can use them to develop some intuition with respect to the interaction between memory architecture and SDF schedules. An MPM-MBS strategy that activates a new actor as soon as possible to continue processing the same piece of data generally results in good data locality, while an SAMAS strategy that activates a new actor as late as possible results in good instruction locality. However, since the execution time of a schedule depends on minimizing the total cache miss penalty, minimizing data miss penalty or instruction miss penalty alone is unlikely to result in good overall cache performance. In the next section, we show the significance of joint optimization of instruction and data cache performance. The rest of the report will develop a heuristic scheduling algorithm that attempts to reduce the total cache miss penalty by considering both data and instruction cache effects.

2. Motivation

This section highlights the importance of cache analysis as an important aspect of schedule generation. Existing SDF scheduling schemes do not take processor memory hierarchy into account during schedule generation, even though the memory accesses constitute significantly to the overall execution time of an SDF graph. The memory access time, for a given SDF graph, is greatly influenced by the caches (instruction and data) present in the underlying hardware. They play an important role as the cache misses are costly, and they can significantly increase the overall execution time. Figure 4 shows that the cache access time and memory access time have a ratio of 1:100, highlighting the importance of reducing memory accesses.

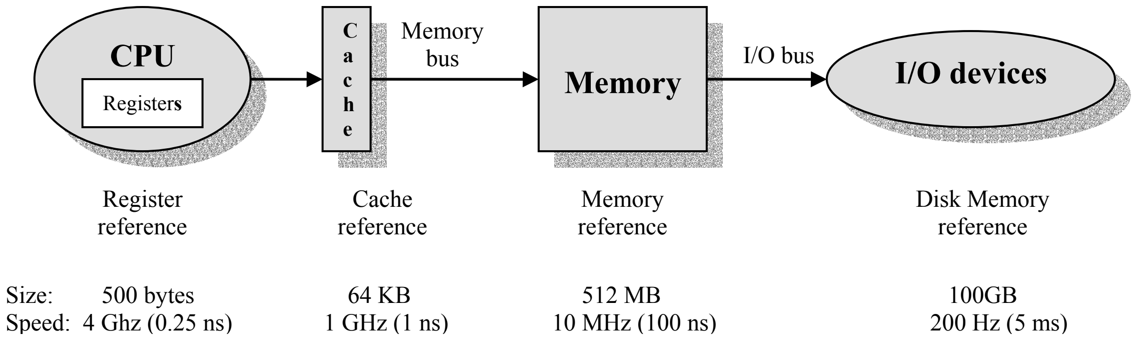

Figure 4 The levels in a typical memory hierarchy in embedded and desktop computers [9].

A cache is a fast memory, i.e. collection of data blocks, in which frequently used data elements are stored to make program execution faster. Caches are either hardware-controlled or software-controlled. A *hardware-controlled cache* is a collection of data blocks in which the controlling hardware decides which memory address maps to which cache block. There are several types of hardware-controlled caches, such as *direct mapped*, *set associative* and *fully associative* [9]. The blocks in a cache are marked either valid or invalid depending on their contents. A cache block is marked valid if it contains useful data else it is marked invalid. When the hardware controller decides to place the contents of a memory location, say M_i, in a cache block, it first checks if that cache block is invalid. If it is, the controller puts the contents of M_i in it; however, if it contains useful data, the controller writes that data back in the main memory before replacing it with the contents of M_i. The hardware controller implements one of the several replacement policies like *Least Recently Used* (LRU) and *Least Recently Replaced* (LRR) [9]. The *software assisted cache*, also known as scratchpad memory, is discussed in the next section.

Since the SDF model of computation has been used heavily in the DSP design environment, we consider a common DSP Processor memory architecture, shown in Figure 5, as the hardware used to execute SDF graphs. The caches are hardware controlled and are fully associative in this analysis, which is the best caching strategy in terms of performance. In this analysis, we also assume that the time taken to write and read from the caches is negligible. In reality, it is not negligible; however, this assumption does not affect the analysis as the data is routed through the caches for all schedules, adding a constant to the overall execution time for each schedule.

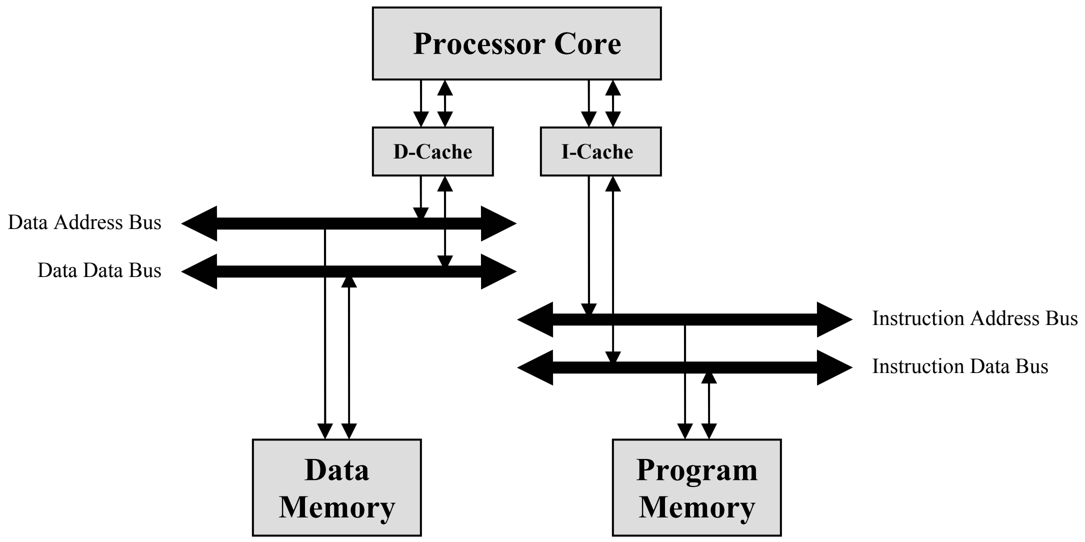

Figure 5 A Harvard Architecture. The processor core can simultaneously access the two memory banks using two independent sets of buses.

We now present an example that highlights the inefficiency of existing scheduling schemes, in terms of cache misses, and motivate the emergence of memory hierarchy as an important scheduling criterion.

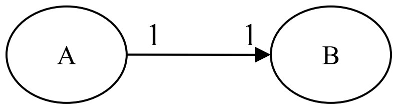

Figure 6 An SDF graph

Consider the simple SDF graph shown in Figure 6. In this SDF model, actor *A* produces one token (one unit of data) per firing and actor *B* consumes one token per firing. The repetition vector for this graph is $q^T = \{q_A, q_B) = \{1, 1\}$ and the only admissible schedule, for a single iteration of this graph is (AB). However, for 100 iterations of this graph, a large number of periodic admissible schedules can be generated. As described in the previous section, the two interesting schedules out of these schedules are the schedule that jointly minimizes the program memory and the buffer requirement, and the schedule that minimizes the actor activations. For the above graph, the MPM-MBS is (100(A)(B)). It has a buffer requirement of 1 unit, let us call it *S'*. The SAMAS for this graph is (100A)(100B). It has a buffer requirement of 100 units, let us call it *S''*. Neglecting the looping overhead, *S''* needs same program memory as *S'*; however, it has a significantly higher buffer requirement.

In both *S'* and *S''*, *A* and *B* are fired 100 times, however, these schedules have different execution times. This is attributed to the difference in the order of actor firings in *S'* and *S''* that significantly affects the total memory accesses time. Let us consider a simple case first. We assume that the programs (codes) of both *A* and *B* are stored in the instruction cache (I-Cache), and the size of data cache (D-Cache) is 25 units. Also, the data cache is a *write back* cache [9], i.e. the data is written only to the block of cache and the modified cache block is written to main memory only when it is being replaced. Since both actors' programs reside in the I-Cache, there is no instruction miss penalty associated with any schedule. So, in this case, the total cache miss penalty for a schedule is equal to the data miss penalty for that schedule. Also, we assume that the time taken to read and write one unit of data (one token) from data or program-memory is t_R and t_W units respectively. In this case, the schedule *S''* will result in a total of 100 data reads and 100 data writes to the data-memory, i.e. a total cache miss penalty of $100(t_R + t_W)$. In comparison to *S'* (zero cache miss penalty), this is an increase of the total execution time by $100(t_R + t_W)$ units. Let us examine this step by step.

The first 25 firings of *A* will fill the data cache completely; each subsequent firing of *A* will result in write back of one unit of data causing 75 evictions (data-memory writes) in all for 100 firings of *A*. At this point, the D-Cache has data produced by the last 25 firings of *A*; however, when *B* starts firing, it requires the tokens produced by *A* in the same order as they were produced. Hence for first 25 firings of *B*, there will be 25 evictions from the D-Cache and 25 reads from the data-memory. The subsequent 75 firings of *B* will cause 75 reads from the data-memory making it a total of 100 reads and 100 writes to the data-memory. In comparison, the Minimum Program Memory-Minimum Buffer schedule *S'* = (100(AB)) does not cause any data cache misses because it has a buffer requirement of only 1 unit while the D-Cache has a space of 25 units; hence its total cache miss penalty is zero. This analysis shows that the schedule *S'* executes in the fastest possible time. As a matter of fact, all schedules of the form ((x_1(AB))(x_2(AB))..(x_n(AB))) where $x_i \leq 25$ for all $i \in \{1,2,..,n\}$ and $\sum x_i = 100$ will have same execution time as *S'*, if looping overhead is negligible, because they do not evict any valid data from the D-Cache.

In the above case, we assumed that both actors' programs reside in the I-Cache; this allowed us to switch back and forth among them without any additional penalty. Now consider a slightly more complex case where only one of the programs of *A* or *B* can reside in the I-Cache at any instant, i.e. the combined size of the two is more than the I-Cache size. Let the I-Cache size, the program size of *A* and the program size of *B* be 15 units each and the I-Cache be empty initially. Let the D-Cache size be the same as the previous case, i.e. 25 units.

In this case, the schedule *S''* will result in a total of 130 data reads and 100 data writes, i.e. a total cache miss penalty of ($130t_R + 100t_W$), out of which 30 reads are attributed to the reading of 15 units each of the program of *A* and *B* from the program-memory, and the rest are the same as discussed in the previous case. Also, in this case, the schedule *S'* will result in a total cache miss penalty of $100(30t_R)$ which is entirely attributed to the reading of the programs of actors *A* and *B* repeatedly. This analysis shows that for the current set of assumptions *S''* is more efficient that *S'*. On careful analysis, we can see that a schedule *S''* = 4((25A)(25B)) is the best schedule in this case as the cache miss penalty associated with it is only $4(30t_R)$. It uses the knowledge of I-Cache and D-Cache sizes as well as the program sizes of the two actors.

The above example shows that because the two schedules *S'* and S'' have different buffer requirement and different firing order, the associated data and instruction cache miss penalties are different depending on the data and instruction cache sizes. We saw that schedule *S'* minimizes the data miss penalty while schedule *S''* minimizes the instruction miss penalty, however, none of them is an obvious choice under all circumstances. This clearly illustrates the importance of the joint analysis of data and instruction cache behavior during schedule generation for SDF graphs and serves as the motivation for this work.

3. Memory Architecture

This section motivates the replacement of traditional replacement policies for hardware-controlled cache by a new cache management policy that utilizes the characteristics of SDF programs. Hardware-controlled cache (HCC) memories with traditional replacement policies offer statistical speed up in general purpose computing but they are not deterministic. Analysis of such hardware-controlled caches is cumbersome and often impossible [10]. Due to the unpredictability of hardware-controlled caches, they are often disabled in embedded system design. In this section we illustrate how we can make the execution of SDF programs faster and deterministic by using a new cache management scheme for data caching instead of the existing replacement policies such as Least Recently Used (LRU) or Least Recently Replaced (LRR).

For chain-structured SDF graphs, the data produced by the actors is read only once. Hence, there is no advantage of replacing any valid block from the data cache to accommodate another data. This forms the basis of our new *cache management policy* for data cache (D-Cache). Under this policy, the modified hardware controller maps a new address to the D-Cache only if an invalid block is present in the D-Cache. Also, any valid block is marked invalid by the cache controller once the data stored in it is read. This cache management policy is better for SDF programs than all existing hardware replacement policies because whenever the data cache is full and a valid block of data is evicted to accommodate another data block, *at least* one write and read from data-memory needs to be performed. However, with the proposed cache management policy, *exactly* one write and read from the data-memory is performed. We now present an example to illustrate this further.

Consider the simple SDF graph shown in Figure 6. Let us analyze the data cache behavior for the Single Appearance Minimum Activation Schedule *S''* = (100A)(100B), defined in the previous section, for both new and existing cache policies. Let the HCC be a fully associative cache of size 50 units. Let the existing replacement policy implemented by HCC be the Least Recently Used (LRU) replacement policy [9]. Under the proposed policy, the controller maps a new address to the cache if an invalid block is present in the HCC. Table 2 shows the number of reads and writes to the data-memory (DM) for both LRU policy and the proposed policy.

	Existing Policy - LRU		**Proposed (new) Policy**	
	Reads-DM	Writes-DM	Reads-DM	Writes-DM
First 50 firings of A	0	0	0	0
Last 50 firings of A	0	50	0	50
First 50 firings of B	50	50	0	0
Last 50 firings of B	50	0	50	0
Total	**100**	**100**	**50**	**50**

Table 2 Reads and writes to data-memory for LRU and the proposed policy while executing the schedule *S''* = (100A)(100B).

The first fifty firings of *A* fill the HCC completely under both policies. However, the HCC evicts these first fifty data units, before they are consumed, under the LRU policy to accommodate the data produced by the last fifty firings of *A* whereas the hardware controller writes that data directly to the data-memory under the proposed policy. During the first fifty firings of *B*, the contents of HCC are again evicted under the LRU policy. It is done to load the data produced by first fifty firings of *A* in the HCC as *B* consumes data in the same order as produced by *A* resulting in fifty unnecessary writes and reads from the data-memory. This example shows that the proposed data cache management policy is a much better choice, for chain-structured SDF graphs, for data caching.

Another interesting observation about traditional cache replacement policies is that they unnecessarily write all data to data-memory; however, this is not a problem for the proposed policy as the hardware controller knows which data are valid and writes only that data to data-memory if the data cache gets full. In the current example, the first fifty units of data produced by *A* are never written back to the data-memory if the data cache implements the proposed policy. This leads to faster execution of the SDF graph. Also, the reduction in memory accesses saves power, as there are fewer bus transactions, and forms another argument in favor of the proposed policy.

An alternative to the hardware-controlled cache is the *scratchpad memory*. A scratchpad (SPM) is a fast compiler-managed (software assisted) SRAM. In comparison, scratchpads offer better real-time guarantees relative to hardware-controlled caches and have significantly lower overheads in energy consumption, area and overall runtime, even with a simple allocation scheme [11]. As the scratchpad is software assisted, its analysis is simple and different allocation schemes can be defined for different types of programs to get better performance by exploiting their characteristics. M. Balakrishnan *et al* [11] have pointed out the advantages of having scratchpad memories in place of traditional instruction caches in embedded system designs. However, using scratchpads for data caching in SDF programs opens an interesting problem in code generation. Since, a scratchpad memory has its own physical address space, unlike the hardware controlled cache, multiple specialized copies of each actor's program need to be generated to account for retrieval of data from the data scratchpad and data-memory. This will result in higher instruction cache (or scratchpad) usage and an increase in instruction miss penalty. For this reason, we do not consider scratchpad memory for data caching in this work and leave the exploration of scratchpad memory for data caching to future work.

Based on the above analysis, and the results shown by M. Balakrishnan *et al* [11], we modify our memory architecture accordingly as shown in Figure 7. We assume that the compiler contains the mechanism to transfer data between the instruction scratchpad (I-Scratchpad) and the program-memory. We also assume that the data cache is fully associative in this architecture, which is expensive to implement.

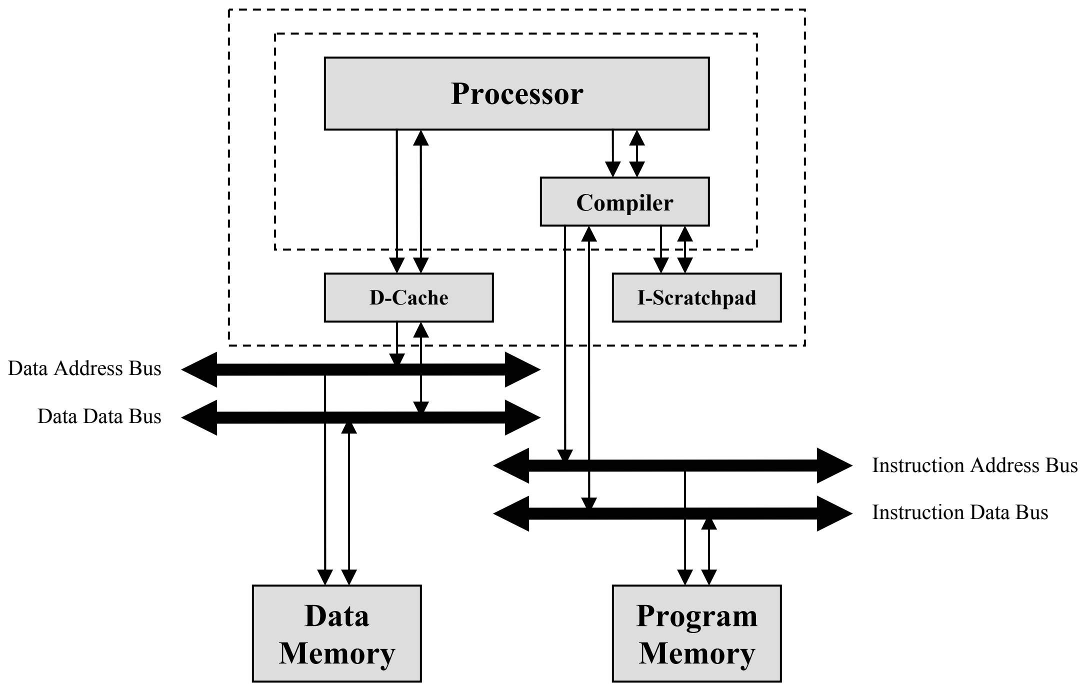

Figure 7 Modified memory architecture: the data cache implements the proposed cache management policy and the compiler controls the instruction scratchpad.

Embedded DSP systems have hard real time deadlines as they react with the environment at the speed of the environment [3]. The worst case execution time (WCET) of an SDF graph *G* for a given firing vector *f* is given by the following equation.

$$WCET(G) = \sum_{i=1}^{n} f_i.WCET(A_i) + \text{Cache Miss Penalty}$$

In general, the WCET for a random program is undecidable, however, for certain actors it can be calculated by doing the structural (path) analysis of their programs. Thus, the total WCET for certain SDF programs, having actors with deterministic WCET, can be determined if we can calculate the cache miss penalty at compile time. As discussed above, analysis of hardware-controlled cache with traditional replacement policies is very difficult and often impossible. However, the proposed data cache management policy and analysis of instruction scratchpad can provide accurate cache miss penalty information at compile time.

3.1 Data Cache and Instruction Scratchpad Policies

In this subsection, we formally list the cache management policy for the data cache and the eviction policy for the instruction scratchpad.

Data Cache Management Policy: The hardware controller maps a new address to the D-Cache only if an invalid block is present in the D-Cache. Upon retrieval of data from a valid block, it is marked invalid. This policy performs well because the data produced by the actors is read only once in SDF graphs.

Instruction Scratchpad Eviction Policy: For a given SDF graph, the sum of actors' program sizes can be greater than the I-Scratchpad size i.e. the I-Scratchpad might not be able to contain all the actors' programs. During the execution of such an SDF graph, a situation can arise where the I-Scratchpad can not accommodate the program of the actor to be executed next without evicting the program of one or more of the actors currently residing in it. Let $I = \{A_{i1}, A_{i2}, ..., A_{ik}\}$ be the set of actors that are currently residing in the I-Scratchpad in the decreasing order of their sizes i.e. $size(A_{i1}) \geq size(A_{i2}) \geq ... \geq size(A_{ik})$. Let the actor whose program has to be loaded to the I-Scratchpad be A_m and let *freeSpace* be the space available in I-Scratchpad at this instant. We propose the following policy, listed as ISEPfunction, to make the eviction choice.

ISEPfunction:

```
Input:     I, the set of actors that currently reside in I-Scratchpad and the
           actor $A_m$ whose program has to be loaded in the I-Scratchpad
Output:    O, the set of actors whose programs have to be evicted
Step1.     $O = \phi$
Step2.     If $A_m \leq freeSpace + A_{i1}$ then evict $A_{it}$ such that
           $freeSpace + A_{it} \geq A_m > freeSpace + A_{i(t+1)}$. $O = O \cup \{A_{it}\}$
Step3.     Else evict $A_{i1}$ and update I to $I \setminus A_{i1}$ and increase the freeSpace to
           $freeSpace + size(A_{i1})$ and iterate through step 2
```

4. Cache Aware Scheduling – Complexity Analysis

In previous sections, we motivated the analysis of memory hierarchy during schedule generation and modified the traditional memory architecture to support deterministic cache analysis. In this section, we analyze the complexity associated with generating the schedule that has the minimum cache miss penalty for a given chain-structured SDF graph, shown in Figure 8.

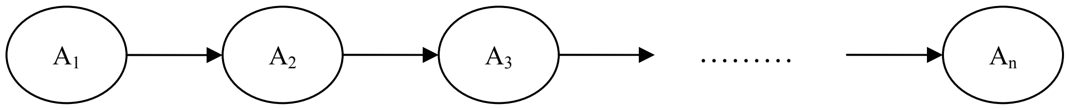

Figure 8 A chain-structured SDF graph with *n* actors.

Cache miss penalty (CMP) for a schedule is defined as the sum of data miss penalty (DMP) and instruction miss penalty (IMP) for the given schedule where the data miss penalty/instruction miss penalty for the given schedule is defined as the sum of data miss penalty/instruction miss penalty incurred at each schedule element. To analyze the complexity associated with generating the minimum cache miss penalty schedule, we need to analyze the complexity associated with the following two tasks:

a) Finding the minimum cache miss penalty for a given schedule.

b) Exhaustive search for the minimum cache miss penalty schedule among all periodic admissible schedules for a given vectorization factor.

Let us first analyze the complexity of finding the minimum cache miss penalty for a given periodic admissible schedule. Let the firing vector of the SDF graph be $f = J \times q$

where J is the vectorization factor and q is the repetition vector. Let the given periodic admissible schedule be represented as a list of schedule elements e.g. $S = ((x_{11}A_1)(x_{21}A_2)...(x_{ij}A_i)...(x_{np}A_n))$, where x_{ij} is the number of firings of actor A_i during its jth activation. Let the total number of elements in the schedule S be k. The sum of the firings for all activations of an actor A_i is f_i, for all $i \in \{1,2,..,n\}$ i.e.

$$\sum_j x_{ij} = f_i$$

Theoretically, it is possible to calculate the minimum cache miss penalty by incrementally analyzing the effect of each schedule element on both the D-Cache and the I-Scratchpad. For a given schedule S, the penalty associated with minimum D-Cache misses, the minimum data miss penalty for S, can be easily calculated as the D-Cache analysis is trivial. It takes linear time in terms of the total number, k, of schedule elements in S. It is fast because the D-Cache controller maps a new address to the D-Cache only if there is free space in it. However the calculation of penalty associated with minimum instruction reads, the minimum instruction miss penalty for S, is not so straightforward. Whenever we need to make space for an actor by evicting one or more actors that currently reside in the I-Scratchpad, there can be many eviction choices (evict an actor or a set of actors) to explore. To the best of the authors' knowledge, there is no polynomial time algorithm to find the best eviction choice. In worst case, n-1 actors can be present in the I-Scratchpad at an instant, resulting in a total of $2^{(n-1)} - 1$ eviction choices. However, some of these choices dominate others. In worst case, the maximum choices that are not dominated can be $2 \times \begin{pmatrix} n-1 \\ (n-1)/2 \end{pmatrix}$. Hence the complexity to find the minimum instruction miss penalty for a k length schedule is $O(\binom{n-1}{(n-1)/2}^k)$, which is exponential in terms of the number of schedule elements and combinatorial in terms of the number of actors in the graph. This highlights the need for an I-Scratchpad eviction policy that we defined in section 3.1.

Now let us analyze the complexity of an exhaustive search for the minimum cache miss penalty schedule among all periodic admissible schedules for a given vectorization factor, assuming that we have a faster way to calculate the cache miss penalty for a given schedule.

Theorem 1: *The number of periodic admissible schedules for a chain-structured SDF graph is at least exponential in terms of the vectorization factor.*

Proof: Any arbitrary chain-structured SDF graph can be represented as a chain-structured SDF graph with two nodes. Let us consider the n actor SDF graph shown in Figure 8. We collapse the nodes $A_2, A_3, .., A_n$ to construct a single node called $A_{(2,..,n)}$, resulting in a two actor chain-structured SDF graph shown in Figure 9.

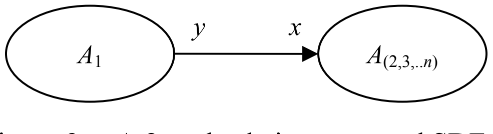

Figure 9 A 2-node chain-structured SDF graph.

Let the first node, A_1, be denoted A and the collapsed node, $A_{(2,..,n)}$, be denoted B. A schedule for one iteration of the above SDF graph is $S = (x\mathrm{A})(y\mathrm{B})$. In the simplest case, x

and *y* are 1 respectively and the SDF graph reduces to the graph shown in Figure 6, and the schedule *S* becomes (AB). By proving that the number of periodic admissible schedules is exponential for this case, we effectively prove that it is at least exponential for any arbitrary chain-structured graph. Let us analyze the case for a vectorization factor *J*. Let *M*(*J*) represent the set of all periodic admissible schedules for the vectorization factor *J* and let the cardinality of this set be |*M*(*J*)|.

We can see that the union of the two sets *P* and *Q* is a subset of the set *M*(*J*) (i.e. $P \cup Q \subseteq M(J)$) where,

$$\begin{aligned} P &= \{AeB : e \in M(J-1)\} \\ Q &= \{eAB : e \in M(J-1)\} \end{aligned}$$

Also we know that all elements of any set *M*(*J'*) end with a '*B*' in order to meet the precedence criterion. Using this fact, we can see that the two sets *P* and *Q* are disjoint; i.e. they do not have any common elements because the penultimate element of all schedules in *P* is always *B* while for elements in *Q*, it is always *A*. The cardinality of both *P* and *Q* is |*M*(*J*-1)|. Thus,

$$|M(J)| \geq 2 \times |M(J-1)| \qquad (1)$$

On expanding equation (1), we get,

$$\left.\begin{aligned} &|M(J-1)| \geq 2 \times |M(J-2)| \\ &|M(J-2)| \geq 2 \times |M(J-3)| \\ &\vdots \\ &|M(2)| \geq 2 \times |M(1)| \end{aligned}\right] \qquad (2)$$

where |*M*(1)| = 1 as *M*(1) contains only one element, i.e. (AB). Combining equation (1) and the set of equations (2), we get,

$$\left|M(J)\right| \geq 2^{J-1} \quad (3)$$

which proves the theorem.

Since the number of admissible schedules is exponential and calculating the instruction miss penalty for any given schedule, by a brute force search, is also exponential, exhaustive search will have exponential complexity. To the best of authors' knowledge, no polynomial time procedure is known that finds the optimum schedule. In the following section, we develop a greedy algorithm based heuristic that generates a cache aware schedule that minimizes the cache miss penalty. The results of this algorithm are sub-optimal; however, it is an efficient heuristic as shown in section 6.

5. Heuristic - Cache Aware Scheduling (CAS)

This algorithm tries to minimize the total memory accesses by utilizing the data cache and instruction scratchpad information, hence the name *cache aware scheduling*. Correspondingly, the schedule generated by this algorithm is called the Cache Aware Schedule. The main idea behind this algorithm is to choose between continuing to fire an actor, when the D-Cache gets full by its earlier firings, or to start firing its successors. The decision is made by comparing the data miss penalty that will be incurred if we continue to fire the actor, and the instruction miss penalty that will be incurred if we stop firing the current actor and start firing its successors.

Let t_R and t_W be the time taken to read and write one unit of data (one token) from the data or program-memory. There is a small fixed cost of evicting an actor's code from the I-Scratchpad due to the operations performed by the compiler; we assume it to be negligible in this analysis. Let the current actor be A_i. Let α_i be the maximum possible number of firings of A_i at the current instant and β_i be the number of firings after which the D-Cache gets full ($\beta_i < \alpha_i$). Let the data miss penalty incurred on firing the actor A_i α_i times be denoted as DMP(A_i, α_i, β_i). Let the instruction miss penalty incurred on stopping the firings of A_i after β_i firings be denoted as IMP(A_i, α_i, β_i). If DMP(A_i, α_i, β_i) is greater than IMP(A_i, α_i, β_i), then the actor A_i stops firing after β_i firings and its successor starts firing else it continues to fire A_i the complete α_i times.

DMP(A_i, α_i, β_i) is defined as the sum of data miss penalty incurred by α_i firings of A_i as well as the data miss penalty incurred due to the firings of A_i's successors that can be fired as a result of α_i firings of A_i. The number of enabled firings for a successor A_k due to α_i firings of A_i is

$$\alpha_{ik} = \left\lfloor \frac{\alpha_{i(k-1)} \times p_{k-1}}{c_k} \right\rfloor$$

where $\alpha_{i(k-1)} = \left\lfloor \frac{\alpha_{i(k-2)} \times p_{k-2}}{c_{k-1}} \right\rfloor$ and $\alpha_{ii} = \alpha_i$ i.e. $\alpha_{i(i+1)} = \left\lfloor \frac{\alpha_i \times p_i}{c_i} \right\rfloor$. The production and consumption rates of actor A_k are denoted by p_k and c_k respectively. The data miss penalty incurred by α_i firings of only A_i is $(\alpha_i - \beta_i) \times p_i \times (t_R + t_W)$.

IMP(A_i, α_i, β_i) is defined as the sum of the instruction miss penalty incurred in loading the program, to the I-Scratchpad, of each successor of A_i that is fireable at least once as a result of β_i firings of A_i. To calculate IMP(A_i, α_i, β_i), we need to iterate through all the successors of the actor A_i in order; IMP(A_i, α_i, β_i) is initially set to zero.

While iterating through the successor list of A_i, if a successor is present in the I-Scratchpad at the time of its analysis, we simply proceed to the next successor without adding anything to IMP(A_i, α_i, β_i); otherwise we analyze the cost (or penalty) of loading this successor's program in the I-Scratchpad and add that cost to IMP(A_i, α_i, β_i). The actor to be evicted from the I-Scratchpad to accommodate this successor is determined by the instruction scratchpad eviction policy defined in section 3.1. Let the successor whose program has to be loaded is A_j. The set of actors to be evicted to load A_j's code is given by O = ISEPfunction(I, A_j) where I is the set of actors currently residing in the I-

Scratchpad and O is the output of the ISEPfunction, i.e. the instruction scratchpad eviction policy. The penalty to be added to IMP(A_i, α_i, β_i) while analyzing the successor A_j is calculated as per the following function:

```
while( $O \neq \phi$ )
{
        1. Let $A_k \in O$.
        2. If $A_k$ is a predecessor of $A_i$, no penalty is added.
        3. Else, if $A_k$ is a successor of $A_i$, a penalty of $(\lceil \alpha_i / \beta_i \rceil - 1) \times size(A_k) \times t_R$
           is added to IMP($A_i$, $\alpha_i$, $\beta_i$).
        4. Else, if $A_k$ is $A_i$, a penalty of $(\lceil \alpha_i / \beta_i \rceil - 1) \times (size(A_k) + size(A_j)) \times t_R$ is
           added to IMP($A_i$, $\alpha_i$, $\beta_i$).
        5. $O = O \setminus A_k$
}
```

The intuition behind these rules is that if A_i was completely fired in a single activation, i.e. α_i times, then in the best case, we would have brought each of its fireable successors only once to the I-Scratchpad and had fired it completely. However, when we stop firing A_i after β_i firings, then each fireable successor of A_i has to be activated at least $\lceil \alpha_i / \beta_i \rceil$ times and we need to account for these extra activations.

In the case where D-Cache gets full during the firing of an actor and its immediate successor is not fireable, then we can not compare DMP(A_i, α_i, β_i) and IMP(A_i, α_i, β_i). In such a situation, we fire the current actor the maximum number of times, i.e. α_i times. If the immediate successor is still not fireable, we return to the nearest predecessor that is fireable. However, if the immediate successor is fireable, then it is fired.

The cache aware scheduling (CAS) algorithm is defined in the following steps.

Step1. Start with the first actor, call it the current actor

Step2. Fire the current actor till either

(a) The D-Cache gets full. Go to Step 3

(b) Or all possible firings of current actor have taken place. If its immediate successor is fireable, make it the current actor, else make its nearest fireable predecessor the current actor. Go to Step 2

Step3. Calculate the DMP(A_c, α_c, β_c) and IMP(A_c, α_c, β_c) where A_c is the current actor, α_c is the maximum possible firings for A_c at this instant and β_c is the number of firings after which D-Cache got full

- If DMP(A_c, α_c, β_c) > IMP(A_c, α_c, β_c), stop firing the current actor
  - If its immediate successor is fireable, make it the current actor. Go to Step 2
  - Else fire the current actor maximum possible times. If its immediate successor is fireable now, make it the current actor, else make its nearest fireable predecessor the current actor. Go to Step 2
- If DMP(A_c, α_c, β_c) $\leq$ IMP(A_c, α_c, β_c), fire the current actor maximum possible times. If its immediate successor is fireable, make it the current actor, else make its nearest fireable predecessor the current actor. Go to Step 2

If the current actor, its immediate successor and any of its predecessors are not fireable, the algorithm terminates. This happens only after the last firing of the last actor in the chain. The worst case complexity of the CAS heuristic for the given n actor chain-structured SDF graph, with a firing vector $f = J \times q$, is $O(totalfirings \times n^2 \log n)$ where *totalfirings* is the sum of firings of all actors i.e.

$$totalfirings = \sum_{i=1}^{n} f_i$$

The minimum number of firings of an actor per schedule element can be 1, so, in the worst case, we need to make *totalfirings* greedy decisions. Also the calculation of IMP for an actor can take a maximum of $n^2 \log n$ steps, hence the complexity.

6. Performance Analysis

In this section, we evaluate the performance of CAS heuristic quantitatively with respect to various parameters that affect its performance. We have implemented the CAS algorithm in Ptolemy, a heterogeneous platform for software prototyping [12]. To analyze the performance of CAS heuristic, we apply it to the CD to DAT benchmark [13] and a set of randomly generated chain-structured SDF graphs. We compare the performance of Cache Aware Schedule with the Minimum Program Memory–Minimum Buffer Schedule (MPM-MBS) [4] and the Single Appearance Minimum Activation Schedule (SAMAS) [6]. Since it is difficult to find the minimum instruction miss penalty for a given schedule, as discussed in section 4, we calculate the instruction miss penalty for the Minimum Program Memory-Minimum Buffer Schedules using the *Instruction Scratchpad Eviction Policy*, defined in section 3.1. For Single Appearance Multiple Activation Schedules, we find the optimum cache miss penalty.

Consider the Compact Disc to Digital Audio Tape (CD-DAT) benchmark shown in Figure 10. Digital Audio tape (DAT) technology operates at a sampling rate of 48 kHz, while compact disk (CD) players operate at a sampling rate of 44.1 kHz. The chain-structured SDF graph of Figure 10 converts the sampling rate from 44.1 kHz to 48 kHz. The repetition vector for this graph is $q^{T} = \{147, 147, 98, 28, 32, 160\}$. The D-Cache size is taken as 204 units which is 20% of the total data produced (1021 units) in one iteration by the CD-DAT benchmark. The size of I-Scratchpad is taken as 100 units. The program sizes for the actors are generated randomly with an upper bound equal to I-Scratchpad size. The randomly generated program sizes of actors *A*, *B*, *C*, *D*, *E* and *F* are 27, 35, 41, 29, 84 and 12 units. The vectorization factor for this graph is taken to be 1.

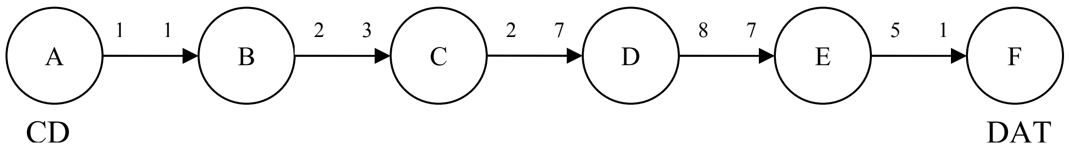

Figure 10 Multi-stage implementation of a CD to DAT sample rate change system

The various schedules for CD-DAT graph shown in Figure 10 are:

CAS: (147A)(57B)(38C)(10D)(11E)(55F)(90B)(60C)(18D)(21E)(105F)

SAMAS: (147A)(147B)(98C)(28D)(32E) (160F)

MPM-MBS: (7(7((3(1A)(1B))(2C)))(4D))(32(1E)(5F))

Figure 11 shows the data, instruction and cache miss penalty associated with the different schedules for the CD to DAT benchmark shown in Figure 10. We can see that CAS is efficient in comparison to SAMAS and MPM-MBS, which is attributed to the fact that CAS heuristic utilizes the size information of both data cache and instruction scratchpad. Figure 11 shows that the CAS performs approximately 7% better in comparison to SAMAS which is the better of the two existing schedules (SAMAS and MPM-MBS) for this case.

To further evaluate the performance of CAS heuristic, we apply it to a set of 100 randomly generated chain-structured SDF graphs and compare its performance with the SAMAS and MPM-MBS schedules. The production and consumption rates of actors in these graphs are chosen randomly from the interval 1 to 10. For each of these graphs, we

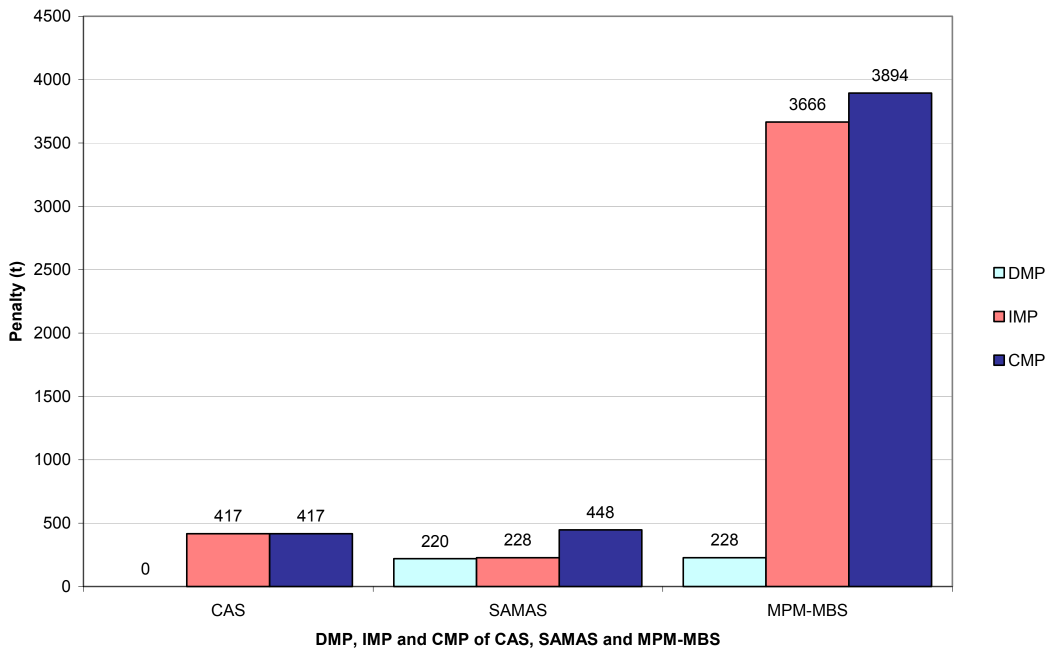

Figure 11 DMP, IMP and CMP of CAS, SAMAS and MPM-MBS Schedules for the CD to DAT sample rate change system.

randomly select the I-Scratchpad size between 50 and 200 units. The code sizes of actors in each of these graphs are chosen randomly from the interval 1 to the randomly generated I-Scratchpad size. The D-Cache size associated with each of these graphs is selected to be 20% of the total data produced by each graph in a single iteration (vectorization factor of 1).

We calculate the cache miss penalty (CMP) associated with each of the three schedules (CAS, MPM-MBS and SAMAS) for each randomly generated graph and compute the percentage improvement in performance, i.e. percentage reduction in CMP, as a result of selecting CAS. We compare the CMP of the CAS with the minimum of the MPM-MBS and SAMAS CMPs for each graph. In cases where CAS results in performance improvement, we calculate the percentage improvement with respect to the best existing schedule, i.e. the existing schedule with minimum CMP. In cases where CAS results in performance deterioration, i.e. increase in CMP, we calculate the percentage deterioration with respect to the CMP of CAS. Figure 12 shows the percentage improvement or deterioration of CAS over the existing scheduling schemes (SAMAS and MPM-MBS). The vectorization factor for this analysis is taken to be 1.

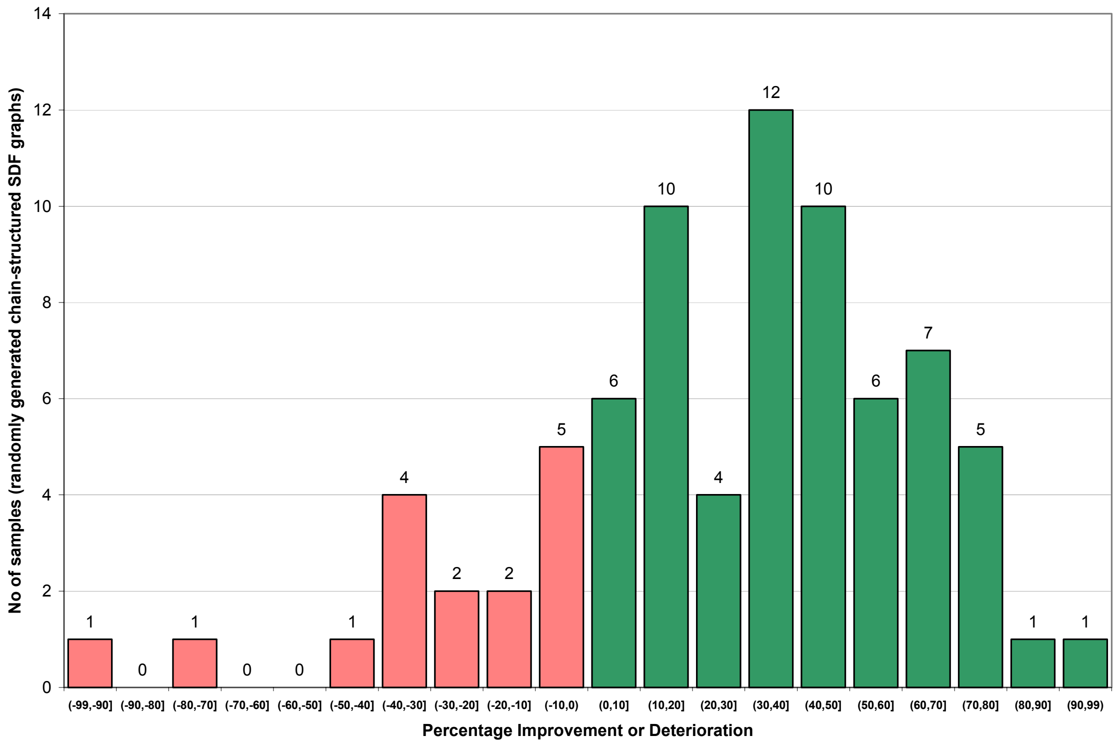

Figure 12 Percentage Improvement or Deterioration of CAS versus existing scheduling schemes for a set of 100 randomly generated chain-structured graphs.

The total number of samples shown in Figure 12 is 78 as choosing CAS results in no performance improvement or deterioration in 22 of the 100 randomly generated graphs. Out of the 78 samples shown above, selection of CAS results in performance improvement in 62 samples with an average improvement of 40.06%. Selection of CAS results in performance deterioration in 16 of the 78 samples shown above with an average deterioration of 27.60%. Overall, for the 100 randomly generated graphs, selection of CAS results in an average performance improvement of 20.43%.

We also calculate the overall performance improvement in CMP of the CAS algorithm as compared to the minimum of SAMAS and MPM-MBS, with respect to the vectorization factor varying from 1 to 20 for the same set of 100 randomly generated graphs. The D-Cache and I-Scratchpad sizes remain fixed all throughout the experiment. Figure 13 shows the results of this experiment.

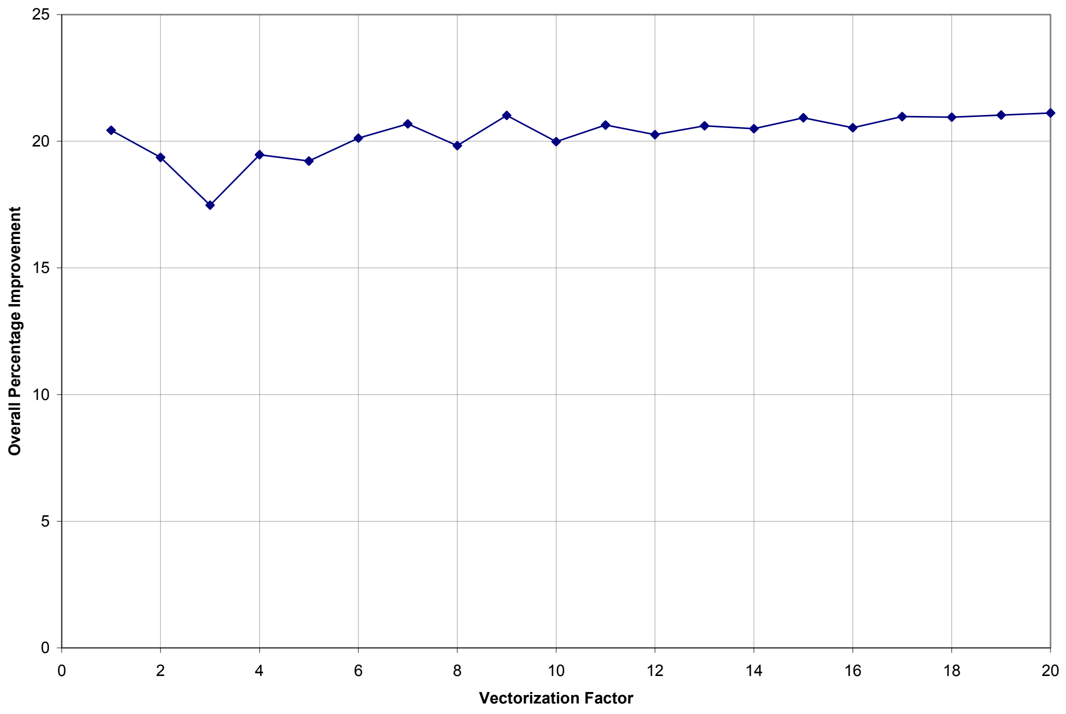

Figure 13 Overall Percentage Improvement of CAS versus existing scheduling schemes for the set of 100 randomly generated graphs for varying vectorization vector.

Increase in vectorization factor implies increase in total data produced. Since the D-Cache size remains fixed, the percentage of D-Cache size with respect to total data produced decreases. Figure 13 illustrates that the overall performance improvement still remains almost the same.

We also analyze the performance of CAS-heuristic with respect to varying the D-Cache and I-Scratchpad sizes. We consider a randomly generated SDF graph shown in Figure 14 for our analysis. The randomly generated program sizes of actors *A*, *B*, *C* and *D* are 32, 7, 47 and 31 respectively. The repetition vector for this graph is $q^T = \{90, 45, 40, 72\}$. Let the vectorization factor be 1, i.e. the firing vector is same as the repetition vector.

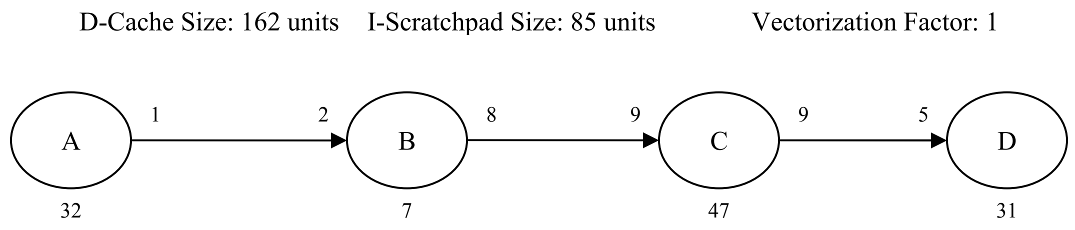

Figure 14 A randomly generated chain-structured SDF graph.

Figure 15 shows the effect of varying I-Scratchpad size on the CMPs of the three schedules. We vary the I-Scratchpad size from 50-90 units keeping the D-Cache size and vectorization factor fixed at 162 units and 1 respectively. In this graph, we observe that increasing I-Scratchpad size decreases the CMP for MPM-MBS and CAS schedules. This decrease is a result of decrease in the IMP of these two schedules, however, the decrease is not linear because an actor's full program has to be loaded or evicted at once from the I-Scratchpad instead of an instruction. The IMP of MPM-MBS schedule stays constant until the I-Scratchpad size is 75 units because the programs of actors *C* and *D* cannot exist together in the I-Scratchpad, which results in program swaps.

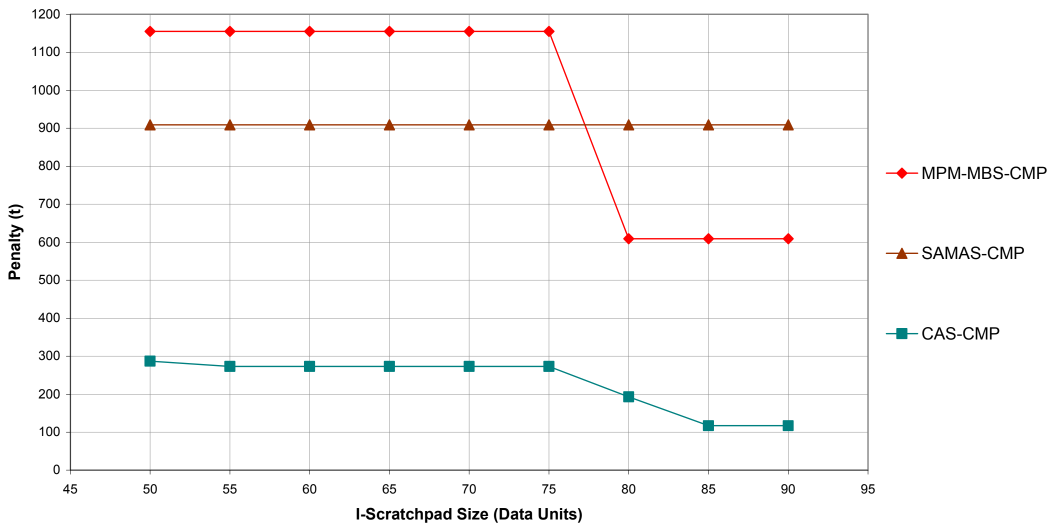

Figure 15 CMP vs. I-Scratchpad size.
(Vectorization Factor: 1, D-Cache Size: 162 units (20% of total data produced))

Once the I-Scratchpad becomes large enough (80 units) to accommodate both of their programs, the IMP for MPM-MBS reduces to minimum, thereby reducing its CMP. Similarly the IMP of CAS schedule decreases whenever the I-Scratchpad gets large enough to accommodate more actors reducing its CMP.

The IMP of the SAMAS schedule stays constant because every actor gets activated only once. Also, the increase in I-Scratchpad size has no influence on the DMP of SAMAS and MPM-MBS schedules because their DMPs depend only on the size of D-Cache, which remains fixed. Thus, the CMP of SAMAS schedule stays constant and the CMP of MPM-MBS schedule varies as per its IMP variations. In this case, the DMP of CAS schedule stays constant (zero) as a result of scheduling decisions taken by the CAS scheduler; hence its CMP changes as its IMP changes.

Figure 16 shows the effect of varying D-Cache size on the CMPs of the three schedules. We vary the D-Cache size from 10-200 units keeping the I-Scratchpad size and vectorization factor fixed at 85 units and 1 respectively.

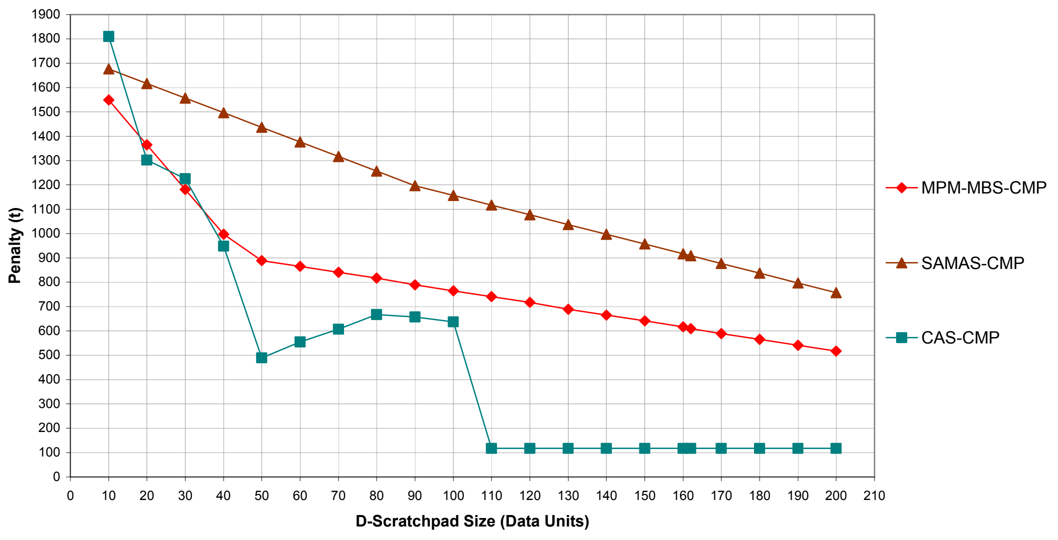

Figure 16 CMP vs. D-Cache size.
(Vectorization Factor: 1, I-Scratchpad Size: 85 units)

In the above graph, we observe that increasing D-Cache size generally results in a decreasing CMP for MPM-MBS and SAMAS schedules. The decrease can be considered as piecewise linear, and the slope changes at points where the data produced by an actor in one activation fits in the D-Cache. In comparison, the CMP of CAS schedule does not always decrease on increasing D-Cache size as it is a heuristic and it takes scheduling decisions based on the comparison of predicted DMP and IMP, which can be incorrect. This phenomenon is evident from the CMP curve of CAS schedule between D-Cache sizes 50 to 80 units. For CAS schedule, the increase in CMP between D-Cache sizes 50-80 is attributed to its DMP which increases due to the behavior of actor *B*. Between D-Cache sizes 50-80 units, the number of firings in first activation of actor *A* go up linearly.

This results in an increase in the number of firings in first activation of actor *B*, whose production rate is much higher than the consumption rate. It causes more data to be produced in the first activation of actor *B* resulting in higher data misses. Eventually, the DMP for CAS schedule goes to minimum, i.e. 0, at 110 units, when there is enough space in D-Cache to fit all the data produced by actor *A* and the data produced by a few firings of actor *B*.

The MBMPS and SAMAS schedules do not depend on D-Cache or I-Scratchpad sizes; hence there is no change in their IMP on increasing D-Cache size. However, the CAS schedule's IMP decreases with an increase in D-Cache size because it produces better schedules that take D-Cache and I-Scratchpad sizes into account. Eventually it becomes equal to the IMP of SAMAS schedule, which is the minimum. In general, the IMP of CAS schedule does not necessarily decrease with an increase in D-Cache size as the scheduler can take a decision that increases IMP but decreases DMP by a greater amount to reduce overall CMP. Figure 16 also highlights that CAS can perform worse (D-Cache size 10 and 30 units) because it is a heuristic. However, in general, the CAS schedule performs better than the other two schedules in terms of CMP and reduces the CMP to minimum when the D-Cache size equals 110 units. In comparison, the MPM-MBS and SAMAS schedules need a much bigger D-Cache size to attain the minimum CMP.

In general, the CAS algorithm trades off DMP for IMP and vice versa, to reduce the CMP. Figures 17, 18 and 19 highlight this characteristic of the CAS heuristic.

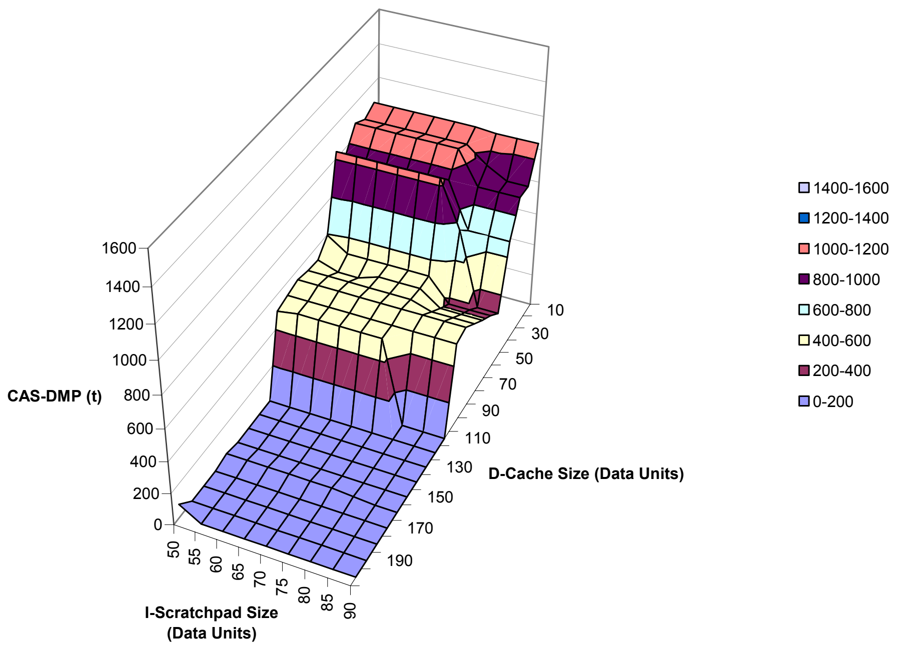

Figure 17 CAS-DMP vs. D-Cache and I-Scratchpad sizes.
(Vectorization Factor: 1)

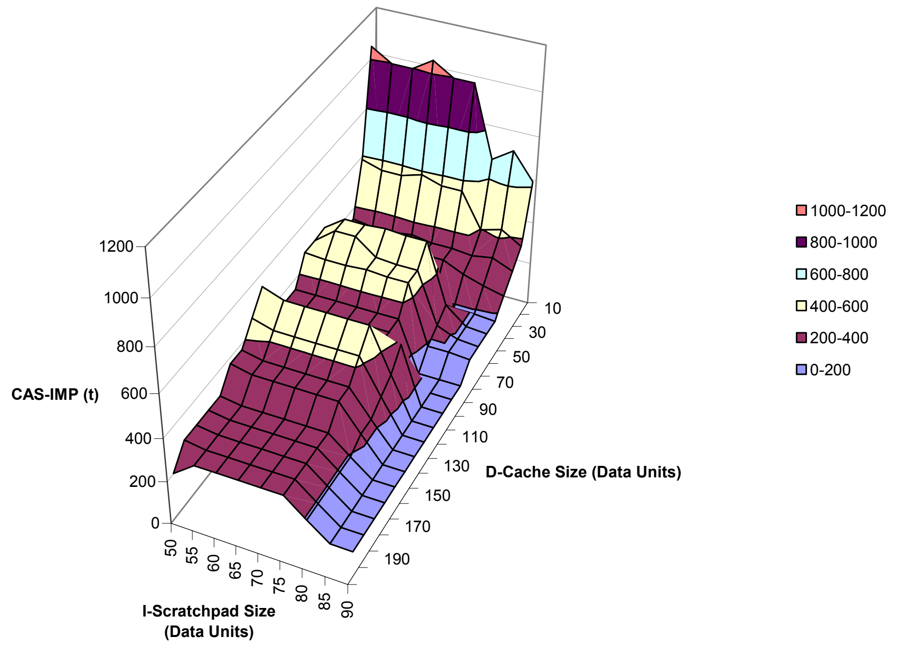

Figure 18 CAS-IMP vs. D-Cache and I-Scratchpad sizes.
(Vectorization Factor: 1)

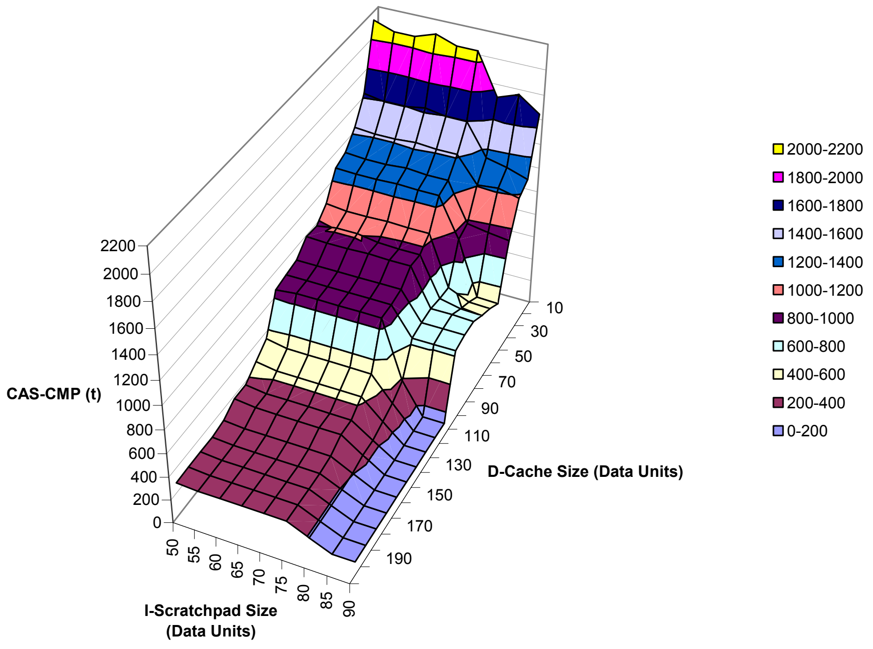

Figure 19 CAS-CMP vs. D-Cache and I-Scratchpad sizes.
(Vectorization Factor: 1)

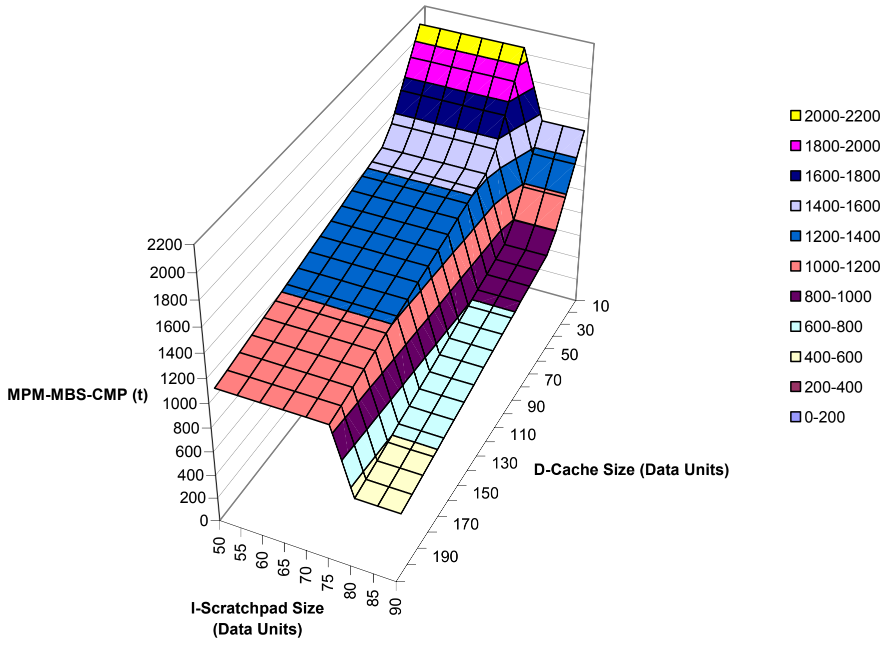

Figure 20 MPM-MBS-CMP vs. D-Cache and I-Scratchpad sizes.
(Vectorization Factor: 1)

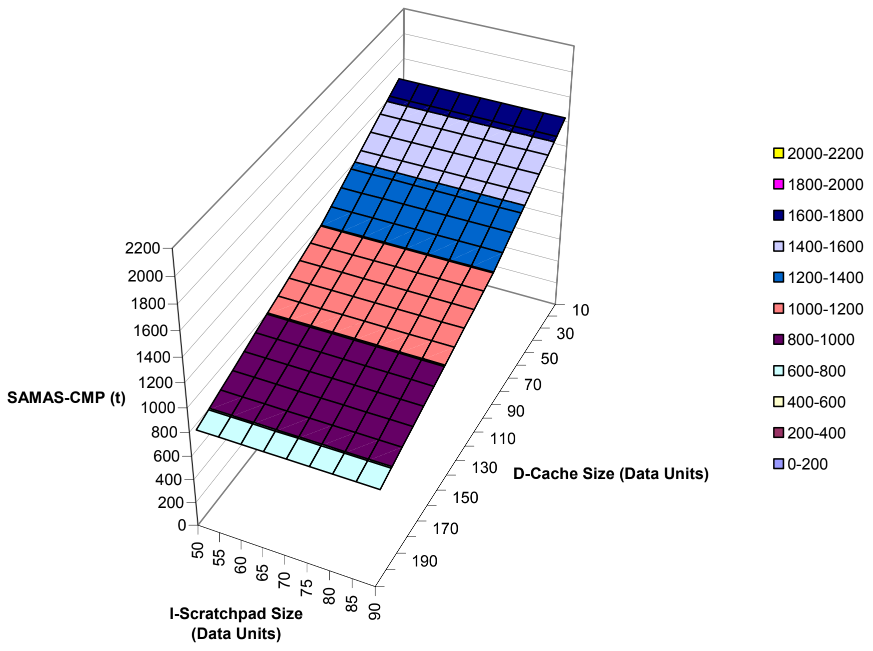

Figure 21 SAMAS-CMP vs. D-Cache and I-Scratchpad sizes. (Vectorization Factor: 1)

Figure 19, 20 and 21 show the CMPs associated with the three schedules for the SDF graph shown in Figure 14 for a variation of 10-200 units in D-Cache size and a variation of 50-90 units in the I-Scratchpad size. The total number of I-Scratchpad and D-Cache pairs is 180. Out of these 180 instances, CAS performs better than the minimum of SAMAS and MPM-MBS schedules 165 times with an average improvement in CMP of 43.56%. The average deterioration in the other 15 instances is 13.46% and the overall improvement of CAS is 38.81%.

6.1 Code Generation

The schedule generated by the proposed greedy algorithm (CAS) is not guaranteed to be a single appearance schedule nor an efficiently looped schedule. For example, the schedule generated by CAS heuristic for the SDF graph shown in Figure 14 with D-Cache and I-Scratchpad sizes as 462 and 85 units is as follows:

(450A)(3B)(2C)(3D)(2B)(2C)(4D)(3B)(3C)(5D)(5B)(4C)(7D)(5B)(5C)
(9D)(8B)(7C)(13D)(11B)(9C)(16D)(13B)(12C)(22D)(18B)(16C)(29D)
(25B)(22C)(39D)(32B)(29C)(52D)(43B)(38C)(69D)(57B)(51C)(92D)

In general, the CAS schedule can have multiple irregular invocations of one or more actors; hence we need to address the issue of code generation for the Cache Aware Schedule. The trivial approach of code generation where each instance of an actor name in the schedule results in a copy of the actor code is highly inefficient for the Cache Aware Schedule in terms of program memory requirement. It can increase the code size tremendously if we have large grain actors in the graph. To avoid such a situation, we suggest the following two-step process to generate code.

- Run a pattern-matching algorithm to find trivial patterns. This will reduce the number of elements in the schedule, thus reducing the code size.
- Generate code for each actor exactly once and make calls to it from the main scheduling loop in the sequence defined by the Cache Aware Schedule. In case of multiple invocations of an actor, in a schedule element, use loops.

Let us consider an example of the above. Consider a chain-structured graph with three actors *A*, *B*, and *C*. Let the generated Cache Aware Schedule be (4A)(2(B2C))(2A)(B)(2C). The main scheduling loop for this schedule will look like this:

```
for(i = 1 to 4) call A;
for(i = 1 to 2) {
        call B;
        for(j = 1 to 2) call C;
}
for(i = 1 to 2) call A;
call B;
for(i = 1 to 2) call C;
```

Some researchers argue that these so-called subroutine calls (to actors' codes) are expensive in terms of the overhead involved in making these calls. However, we think that they should not be viewed as subroutine calls. The main scheduling loop does not perform any computation; hence only the program counter needs to be saved, which is not all that expensive considering the program-memory savings achieved by this procedure. In fact unconditional jumps (JMP) can be used in the main scheduling loop to access the actors' program code. Before such a jump, the PC can be incremented and saved in a register, say *R* that is not used by any actors' code. At the end of each actor's code, instead of the return command, jump to *R* command can be appended. This will result in the execution of three extra assembly instructions per execution of an actor, which is negligible overhead in case of large grain actors.

Also, code generation of schedules using subroutine calls results in better instruction scratchpad utilization as only one copy of each actor's code needs to be loaded in the instruction scratchpad. In comparison, the traditional technique will load a new copy of an actor's code every time it gets activated.

6.2 Extensions

The CAS algorithm proposed in section 5 can be easily extended to well-ordered SDF graphs. In well-ordered SDF graphs, the token production rate of an actor with more than 1 outgoing arc is considered to be the sum of token production rates at each outgoing arc. Similarly, the token consumption rate of actors with more than 1 incoming arc is the sum of token consumption rates at each incoming arc. The rest of the algorithm including the data miss penalty and instruction miss penalty calculation remains the same as CAS for chain-structured graphs.

7. Conclusions and Future Work

In conclusion, a novel scheduling algorithm, named cache aware scheduling, has been presented to minimize the cache miss penalty. Data cache management policy and instruction scratchpad eviction policy to support the CAS algorithm and address the issue of unpredictable cache behavior have also been presented. A solution to address the code generation problem has also been presented.

The schedules generated by the proposed algorithm are shown to be highly efficient in comparison with the Minimum Program Memory-Minimum Buffer schedules and the Single Appearance Minimum Activation schedules. The proposed algorithm is only applicable to well-ordered SDF graphs in single processor environment. Cache aware scheduling of arbitrary acyclic and cyclic SDF graphs will be considered in future. The future work also includes development of multiprocessor cache aware scheduling scheme. Also, the exploration of scratchpad memory for data caching will be considered in future.

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 T. M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995.

[3] E. A. Lee, “Embedded Software,” *Advances in Computers*, vol. 56, Academic Press, London 2002.

[4] 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.

[5] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Signal Processing,” *IEEE Transactions on Computers*, vol. C-36, (no.1):24-35, January, 1987.

[6] S. S. Bhattacharyya and E. A. Lee, “Scheduling synchronous dataflow graphs for efficient looping,” *Journal of VLSI signal Processing*, vol. 6, December 1993.

[7] S. Ritz, M. Pankert, V. Zivojnovic and H. Meyr, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs," *Technical Report* IS2/DSP93.1a, Aachen University of Technology, Germany, January 1993.

[8] S. Ritz, M. Pankert and H. Meyr, "High level software synthesis for signal processing systems," in *Proceedings of the Intl. Conf on Application-Specific Array Processors*, pp. 679-693, Prentice Hall, IEEE Computer Society, 1992.

[9] Henessy and Patterson, "Memory Hierarchy Design," *Computer Architecture*, Third Edition, Morgan Kaufmann Publishers, 2003.

[10] B. L. Jacob and S. S. Bhattacharyya, "Real-time Memory Management: Compile-Time Techniques and Run-Time Mechanisms that Enable the Use of Caches in Real-Time Systems," Technical Report, *Institute for Advanced Computer Studies*, University of Maryland at College Park, September 2000.

[11] 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.

[12] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng (eds.), "Heterogeneous Concurrent Modeling and Design in Java", Vol 1–3, *Technical*

Memorandum No. UCB/ERL M03/27 – MO3/29, University of California, Berkeley, CA USA 94720, July 16, 2003.

[13] T. M. Parks, “Bounded Scheduling of Process Networks,” *Technical Report UCB/ERL-95-105*, Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, CA USA 94720, December 1995.