A Scratchpad Memory Allocation Scheme for Dataflow Models

Shamik Bandyopadhyay
Thomas Huining Feng
Hiren D. Patel
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-104
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-104.html

August 25, 2008

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards \#0720882 (CSR-EHS: PRET) and \#0720841 (CSR-CPS)), the U. S. Army Research Office (ARO \#W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI \#FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

A Scratchpad Memory Allocation Scheme for Dataflow Models

Shamik Bandyopadhyay
EECS, UC Berkeley
shamikba@microsoft.com

Thomas Huining Feng
EECS, UC Berkeley
tfeng@eecs.berkeley.edu

Hiren D. Patel
EECS, UC Berkeley
hiren@eecs.berkeley.edu

Edward A. Lee
EECS, UC Berkeley
eal@eecs.berkeley.edu

August 25, 2008

Abstract

Scratchpad memories are alternatives to caches in real-time embedded processors. They provide better timing predictability and lower energy consumption. However, program code and data must be explicitly moved in the memory hierarchy. Current practice either leaves it up to the programmer to manually manage the memory or to use low-level compiler techniques to create an allocation schedule. In this paper, we show how to leverage the structure and semantics of a dataflow model to make optimal use of scratchpads. We assume the heterochronous dataflow model of computation (or its special cases). To show feasibility of the approach, we formulate an ILP problem to minimize the memory access times. We provide performance comparisons between our memory allocation scheme and caches with LRU replacement policy.

1 Introduction

The use of on-chip memory in a processor's memory hierarchy is critical in bridging the processor-memory gap. Caches are the common on-chip memory solution for traditional architectures. However, for real-time embedded processors, on-chip memories in the form of scratchpad memories are often favored over caches. The reasons to opt for scratchpads are low power and energy consumption, and a higher degree of predictability in program execution times [4, 35, 22, 30]. These advantages are in part because unlike caches, scratchpads do not require hardware policies for determining when and what data needs to be moved in and evicted from the on-chip memory. Instead, scratchpads offer a software managed on-chip memory solution. This reduces the comparator logic thus reducing energy and power consumption. It also provides better analyzability of

a program's execution times, thus improving the predictability of its execution times. But at the same time, scratchpads put the burden on the programmer to schedule memory transfers from/to the off-chip. In efforts to reduce this burden, automated memory allocation schemes are typically employed to schedule transfers to/from scratchpads.

There are already many such allocation schemes for scratchpads [2, 24, 33, 23, 37, 26]. Most existing memory allocation schemes use compiler-based analysis on C/C++ programs to develop the allocation strategy. This has the advantages of reducing the programmer's burden, better managing large programs, and easier porting to different target architectures. However, extracting the semantics of C/C++ programs through static analysis is in general a very difficult problem. This results in strict programming guidelines and conservative estimations during static analysis. We find this to be a notable limitation in most compiler-based memory allocation schemes.

Instead of using low-level compiler-based analysis to extract the structure and semantics of a program, we propose using a higher-level dataflow model of computation (MoC) that exposes the structure and semantics of a model for analysis, which is difficult to perform via static analysis of C/C++. We use the heterochronous dataflow (HDF) model of computation [16, 39] in Ptolemy II [11] as a means to specify the program. The HDF MoC is an extension to the synchronous dataflow (SDF) MoC [20, 5] with finite state machines (FSM). It is well suited for implementing control protocols and adaptive algorithms that require dataflow rate variation during execution [39]. In HDF semantics, changes in the FSM's state correspond to changes in the dataflow network and the dataflow rates of the SDF. These state changes only occur at the end of an iteration. We use these points of state changes to direct our memory allocation scheme.

Certain special cases of HDF are used in domain-specific scenarios for embedded systems development. LabVIEW (from National Instruments), for example, uses structured dataflow, and is widely used in instrumentation systems. SDF [20, 5] and cyclo-static dataflow (CSDF) [10] are used for signal processing applications, for example in the products Advanced Design Systems (ADS) from Agilent, SPW from CoWare, and its successor, Signal Processing Designer, as well as several applications aimed at audio, image, and video processing. Structured dataflow and SDF have recently been elaborated into the language StreamIT, used for programming multicore systems [31]. Because HDF is a generalization of these MoCs, our methods could apply in principle to all with minimal adaptation.

Our dataflow specification uses *actor-oriented* [19] design principles, where "actors" (components that are in charge of their own actions) communicate by exchanging messages. The streams of messages in the form of data tokens are called signals in the dataflow. Actors execute in response to the available data in their input signals. These actors and signals connecting them define the structure of the HDF model.

We exploit the structure and semantics of HDF models to gather information about the temporal pattern and access frequencies of memory references and

requirements of the computation blocks. Our memory allocation scheme uses this information to create a schedule for program code and data.

For us, program code and data are synonymous to actor code and data buffers in an HDF model.[1] Our memory allocation scheme uses an integer linear programming approach for both the actor code and data buffer memory for the HDF model to schedule transfers to the scratchpad memory. Our optimization criterion is to minimize the memory access time cost. However, we also show the flexibility of our formulation by changing the optimization criterion to low energy consumption. As a bonus, we show that our memory allocation scheme has a higher degree of timing predictability when compared to caches. We compare our memory allocation scheme with the use of caches with least recently used (LRU) replacement policy.

1.1 Main Contributions

Our contributions in this paper are: 1) to present a scratchpad memory allocation scheme that uses the structure and semantics of an HDF model to make optimal use of the scratchpads, 2) to show the flexibility of this allocation scheme by extending it to support multiple levels of memory in the memory hierarchy, 3) to easily change the optimization metric between memory access time and energy consumption and 4) to perform memory allocations based on history information. We also present experimental results obtained from using our memory allocation scheme.

1.2 Organization

We present background on the synchronous dataflow and heterochronous dataflow models of computation and brief descriptions of caches and scratchpad memories in Section 2. Section 3 discusses related work in using scratchpads and allocation algorithms. We formulate our problem by describing our assumptions and observations in Section 4. In Section 5, we describe our dynamic scratchpad allocation scheme. We explore this scheme to show optimization strategies based on the actor and data code memory access time and energy costs. In addition, we extend our optimizations to take multiple allocations based on execution history for each HDF state into account in Section 5.4. Our experimental results are presented in Section 6 and in Section 7 we conclude.

2 Background

2.1 Synchronous Dataflow

A dataflow model of computation [21] represents concurrent programs through interconnected blocks called actors. Each such block represents a function or

[1] Without loss of generality, we assume that actor state is modeled as buffers in a feedback loop.

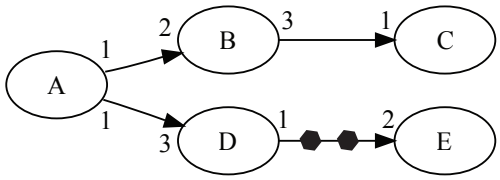


Figure 1: An SDF model shown as a directed graph

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. To model actors with state, we assume an input port and an output port for the state, where the output is connected directly back to the input via a buffer that is initialized with the initial state.

Synchronous dataflow (SDF) [20], 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*. These numbers are known as the *production rate* and the *consumption rate*, separately. They together give the *rate signature* of an actor. The memory required by executing an SDF model consists of the memory for actor code and that for data buffers. An example of SDF model is shown in Figure 1 [39].

Each data 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 shows two units of delay on the buffer connecting actors D and E. An iteration of an SDF model is a sequence of firings of the actors that returns the FIFO buffers to their original sizes. A *periodic admissible schedule* or *valid schedule* for an SDF model is a sequence of actor firings such that deadlock does not occur and no net change in the number of tokens present on the edges is produced [20]. 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 model.

2.2 Heterochronous Dataflow (HDF)

SDF is a very robust and well studied model of computation that allows for static schedule and deadlock-free execution. However, one key limitation of SDF models is that the rate signatures of actors must be fixed and defined *a priori*. For this reason, SDF models prove unsuitable for implementing control protocols and adaptive algorithms that require dataflow rate variation during execution [39]. Heterochronous dataflow (HDF) is a model of computation that significantly increases the expressiveness of SDF and makes it suitable for implementing variable rate models.

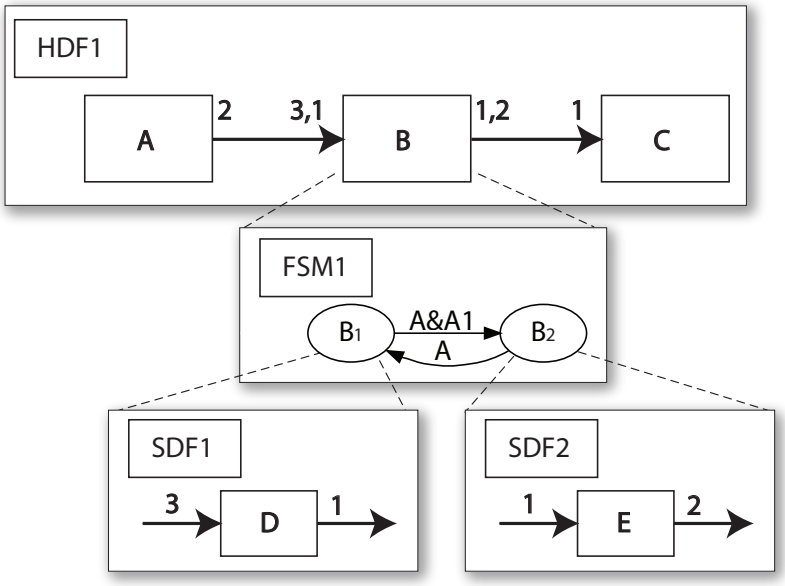


Figure 2: A simple HDF model

The HDF model of computation was originally introduced by Girault et al in [16]. In simple terms, HDF is a heterogeneous composition of finite state machines (FSM) and SDF, in which the state changes correspond to the changes in dataflow networks and dataflow rates. 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 [16]. A composite actor (a composition of other actors) in HDF is composed of an FSM, whose individual states 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 considered disconnected from the system.

Each state of HDF is identified by 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 schedule for that state. Hence, each state of the HDF model corresponds to a different schedule for the system. Changes in rate signatures or state changes are restricted to occur only at the end of an iteration of the HDF model.

Figure 2 [16] shows a simple HDF model. 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 (global) states:

$$S_1 = AB_1C$$
$$S_2 = AB_2C$$

In state S_1, B consumes three tokens and produces one. This leads to the schedule $[A, A, A, B, B, C, C]$. In state S_2, B consumes one and produces 2, leading to the schedule $[A, B, B, C, C, C, C]$. In general, HDF models can contain multiple composite actors at the same level and can have an arbitrary depth of hierarchy [39]. It is possible to statically pre-compute all schedules for all reachable states of the HDF model, assuming all state machines are finite. This avoids the overhead of computing a schedule every time after a state transition.

Structured dataflow, as used in LabVIEW and StreamIT, borrows ideas from structured programming to create nested constructs that be modeled by HDF. Specifically, these nested constructs express conditional processing of tokens and iteration (both manifest and data dependent). CSDF can also be modeled by HDF by constraining the guards on mode transitions so that the state machines cycle periodically. Parameterized SDF [8] can also be modeled by HDF, but the state machines are no longer finite state, so the usefulness of this is questionable.

2.3 Caches

A *cache* is a fast on-chip memory, in which frequently used data elements are stored to make program execution faster. It takes advantage of the *principle of locality* [18], 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. 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 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. Similarly, the total area of the cache is also the sum of the areas of each of the aforementioned components. 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.

2.4 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. 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 as for caches. The simplicity of the hardware architecture also greatly lowers the power consumption and area of a SPM [4].

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 an 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 that 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. 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 Related Work

In the context of embedded systems, SPMs have proven to be the better choice for on-chip memory architecture in terms of predictability, but with the increase in the use of embedded processors for mobile applications, power and energy consumption has become a critical limiting factor. The power and energy consumption of SPMs are significantly lower than that of caches [6]. Scratchpad memories are also significantly smaller in area than caches of similar capacity. In [3], 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 [28].

Since SPMs are managed by the program, the responsibility of scheduling the memory allocation is often passed onto the programmers. This however, is cumbersome and thus, several automatic scratchpad allocation algorithms have been developed to address this issue. For example, Panda et al [25] present a partition algorithm that uses an intersecting lifetime time criterion for deciding which arrays to allocate on SPMs and off-chip. This metric exposes the possible cache conflicts between accesses to arrays and aims to minimize the cache conflict penalties. Steinke et al [29], Avissar et al [2] and Xue et al [38] address similar problems but focusing on energy reduction. Puaut et al compare locked caches to scratchpad memories [27] with respect to instruction code. Their experiments show that SPMs suffer from fragmentation and propose splitting of the basic blocks.

Software caching [17] is another method, 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 [28, 34, 1, 2]. 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 [32, 15, 14].

Whitham and Audsley [36, 37] use trace instruction scratchpads to reduce execution times for real-time architectures. They propose algorithms to identify traces that minimize the average case execution by parallelizing the frequently executed code blocks. Their approach uses a static allocation scheme where input programs are specified in C. Furthermore, trace scratchpads are only used for the instruction memory.

Milidonis et al describe a decoupled processor architecture that only uses SPMs in its memory hierarchy [23]. They dedicate an *Access* processor to perform transfers between various levels of the memory hierarchy and from the L1 SPM to directly the register file. The memory allocation is supported via the compiler that requires a certain amount of static analysis to be done. The *Execute* processor is the traditional integer and floating unit, which interacts with the Access processor via interrupts and handshake protocol.

In order to make good use 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, these algorithms are hampered by incomplete semantic information about program structure and dynamic behavior. HDF models have much more exploitable information than C/C++ programs. We show here how to exploit that information.

4 Problem Definition

We address the problem of formulating an automatic allocation scheme for mapping the key memory requirements of HDF models to the scratchpad memory. This scheme presents a high level, coarse granularity scratchpad allocation for HDF models. Since the HDF model of computation is an extension to the SDF model of computation, we consider code memory (actor code) and data memory (buffer data) as the key memory requirements [9].

4.1 Structure of an HDF Model

We use Ptolemy II's graphical interface and the HDF domain to specify an HDF model. In this domain, actors are connected via signals. This makes it straightforward to identify actor code and the corresponding data buffers in the HDF model. We assume that actors and data buffers are atomic units so that an actor is entirely allocated to the scratchpad or not at all. This is true for data buffers as well.

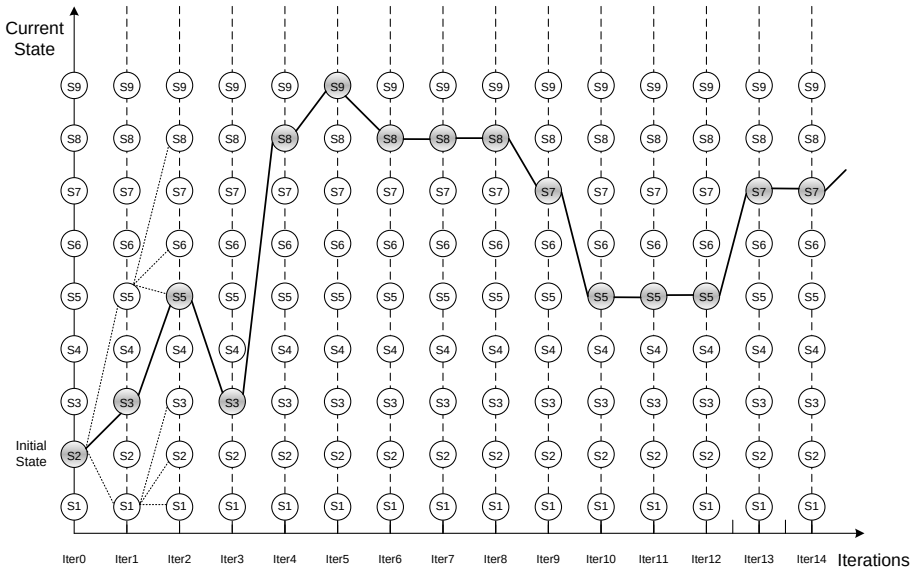


Figure 3: Trellis Diagram for the Execution of an HDF Model

4.2 Temporal Characteristics of Memory Accesses in an HDF Model

As mentioned earlier, HDF models can potentially change state at the end of a global iteration. For the duration of a particular iteration, the state, the rate signatures and the schedule remain fixed. Thus the entire execution of an HDF model can be viewed as a path through a Trellis diagram. For example, Figure 3 shows the first 14 iterations of an HDF model with 9 possible states. The model starts in initial state S_2 and then transitions to the shaded state at the end of each iteration. For the duration of any particular iteration, the model stays in the shaded state. 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. By virtue of the semantics of HDF, the actual path of execution is data dependent and not known a priori.

An important observation here is 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 are usually different from the access patterns for another state. Thus, memory accesses follow a particular trend for the duration of one iteration and then potentially change to a different trend at the next iteration.

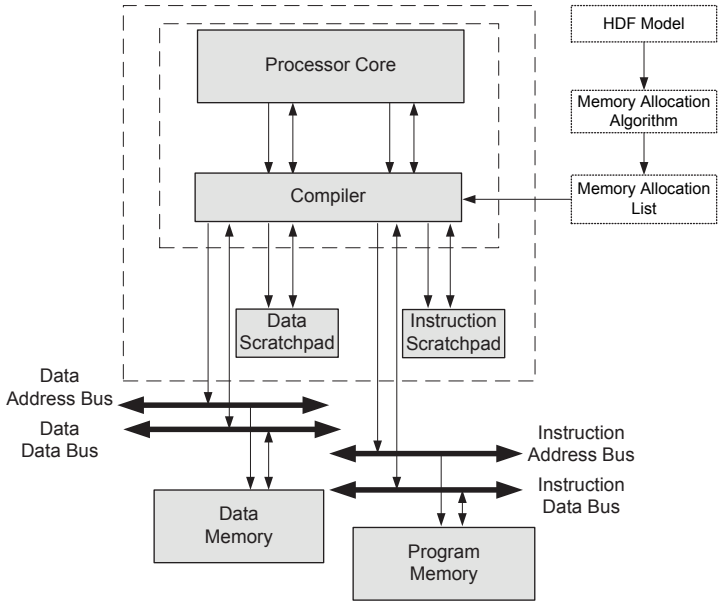


Figure 4: Modified memory architecture: the instruction and data scratchpad are controlled by the compiler based on the allocations generated by the memory allocation algorithm

4.3 Assumptions

Assumption 1: Memory architecture

We assume a Harvard architecture for the memory system. We consider the code memory and data memory to be present in separate independent memory banks with independent buses, as depicted in Figure 4. 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*. The software controller accesses this list between iterations to decide whether the items of code memory and data memory should be stored in the scratchpad for the next iteration.

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

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 large enough to accommodate the entire code and data memory requirements of the HDF model.

Assumption 3: Actors are black-boxes.

Atomic actors are considered opaque. Opaque actors do not have any refinements within them. 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 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. The code block for an actor is either allocated entirely in the scratchpad or is not allocated in the scratchpad at all. The same applies for data buffers.

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

In order to make 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 may contain branches that cause the actual number of accesses to differ. Extensive profiling or static analysis of the actor code could be used (imprecisely) to guage the number of accesses per invocation. Instead, we simplify the scenario by assuming the code size of an actor to be representative of the number of accesses to its code memory in an invocation. The code size is a parameter in our algorithms and can be easily replaced by the results of profiling.

Assumption 6: An allocation scheme remains fixed in each state of the HDF model; allocation schemes are switched only at state changes.

Our algorithm computes an optimal allocation scheme for each reachable state of the HDF model. Based on this assumption, we claim our schemes to be optimal.

An improvement that we will explore in the near future is to relax this assumption by allowing changes of allocation schemes within a state. This potential improvement on the one hand opens up opportunities for further optimization, but on the other hand complicates the allocation algorithm. Its effectiveness heavily depends on the applications. For HDF models in which state changes are frequent and the overhead of moving data in and out of the scratchpad is relatively high, allowing changes of allocation scheme in a state may even degrade performance.

Assumption 7: Firing schedule

We further assume for any reachable state of the HDF model, a single appearance firing schedule [20], which includes the firing count and firing order of actors. The firing order does not affect the performance of our memory allocation scheme. However, a single appearance schedule supplies caches with the best possible schedule. This allows us to compare the performance between caches with best possible schedules and SPMs with our memory allocation scheme.

4.4 Observations

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

Given *Assumption 5*, for an actor A_i, code size C_i represents the number of code memory accesses for a single invocation of an actor. In a given state, if A_i is scheduled to fire f_i times, then

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

Observation 2: The code memory is read-only.

The actor code is not self-modifiable. Hence, the accesses to it is read-only.

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

In order to move the code block of an actor from off-chip memory to the scratchpad, 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.

Because the off-chip memory contains all contents of the scratchpad, there is no need to write back the code when an actor is evicted from the scratchpad.

Observation 5: The total number of write accesses to the data memory for a data buffer in a particular schedule is the product of the production rate and the number of 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,

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

Observation 6: The total number of read accesses to the data memory for a data buffer in a particular schedule is the product of the consumption rate and the number of 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,

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

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

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

$$\text{Cost of moving a data buffer to scratchpad} = T_{Offchip \rightarrow SPW} \cdot I_i$$

Observation 8: The cost of removing a data buffer from scratchpad memory is the product of the unit migration cost from the scratchpad memory to the off-chip memory and the number of tokens 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 be written back to the off-chip memory upon evicting a data buffer from scratchpad.

$$\text{Cost of evicting a data buffer from scratchpad} = T_{SPW \rightarrow Offchip} \cdot I_i$$

5 Dynamic Scratchpad Allocation Scheme

In this section we develop an allocation technique to map the actor code and data buffers of HDF models to the scratchpad memory. We select state changes as the points at which allocations of the scratchpad memory are altered. An allocation is fixed for the duration of a state. For each state, we statically compute an optimal allocation scheme.

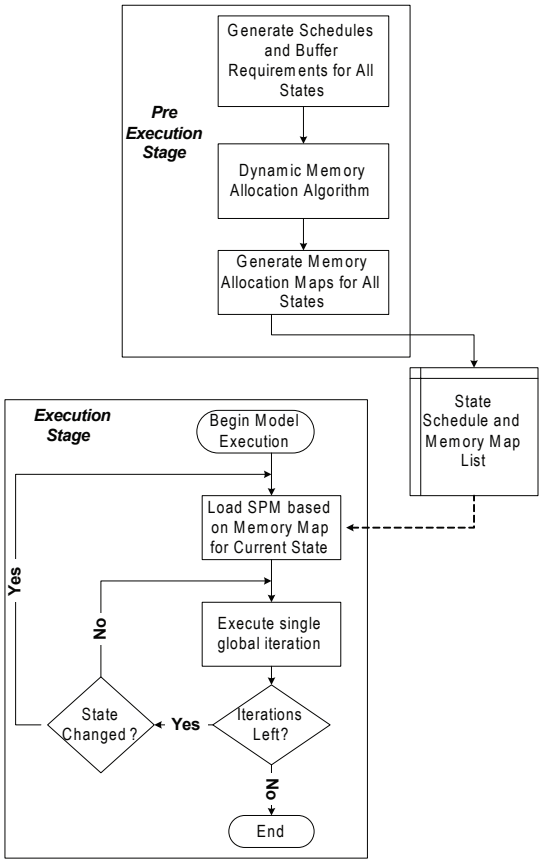


Figure 5: Overall structure of memory allocation scheme

5.1 Overall Structure of Allocation Scheme

Figure 5 shows the overall organization of the dynamic allocation scheme. The scheme is separated into two 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 memory requirements for all states are generated. They are supplied as inputs to the memory allocation algorithm. The algorithm generates a memory map for each state that identifies the actors and buffers to be placed in the scratchpad.

During the execution of the HDF model, state transitions are identified at the end of each iteration. Immediately after a state transition, the scratchpad memory is loaded with the actor code and data buffers specified by the memory map for the new state. Only those actor code and data buffers that are not already in the scratchpad memory are loaded. If necessary, tokens residing in buffers in the old state are evicted.

5.2 Memory Allocation Algorithm

The memory allocation algorithm generates the list of actors and data buffers to be placed in the scratchpad for each state based on the supplied schedule, rate signatures and memory requirements. The general problem of optimal data allocation is known to be *NP*-complete. We formulate this problem as an integer linear programming (ILP) problem.

5.2.1 Formulation of Variables

We formulate the variables for the ILP problem in Table 1. Among them, the following are the 0/1 Boolean variables formulated to represent the locations of actor code and data buffers.

$$M_{Offchip}(a_i) = \begin{cases} 1 & \text{if actor } a_i \text{ is in off-chip only} \\ 0 & \text{otherwise} \end{cases}$$

$$M_{SPW}(a_i) = \begin{cases} 1 & \text{if actor } a_i \text{ is in scratchpad} \\ 0 & \text{otherwise} \end{cases}$$

$$M_{Offchip}(d_i) = \begin{cases} 1 & \text{if data buffer } d_i \text{ is in off-chip only} \\ 0 & \text{otherwise} \end{cases}$$

$$M_{SPW}(d_i) = \begin{cases} 1 & \text{if data buffer } d_i \text{ is in scratchpad} \\ 0 & \text{otherwise} \end{cases}$$

Note that the above are two pairs of complementary variables.

$$M_{Offchip}(a_i) = 1 - M_{SPW}(a_i)$$
$$M_{Offchip}(d_i) = 1 - M_{SPW}(d_i)$$

Variable	Meaning
U	Number of memory units
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
D	Number of data buffers
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
$F_{dest}(d_i)$	Number of firings of the destination actor of d_i
$Size_{DataSPM}$	Size of scratchpad for data in bytes
$Size_{InstrSPM}$	Size of scratchpad for actor code in bytes
$T_{Offchip\,Rd}$	Time to read a byte from off-chip memory in cycles
$T_{Offchip\,Wr}$	Time to write a byte to off-chip memory in cycles
$T_{SPW\,Rd}$	Time to read a byte from scratchpad in cycles
$T_{SPW\,Wr}$	Time to write a byte to scratchpad in cycles
$T_{Offchip \rightarrow SPW}$	Time to move a byte from off-chip to scratchpad in cycles
$T_{SPW \rightarrow Offchip}$	Time to move a byte from scratchpad to off-chip in cycles
$M_{Offchip}(a_i)$	Whether a_i is in off-chip only
$M_{SPW}(a_i)$	Whether a_i is in scratchpad
$M_{Offchip}(d_i)$	Whether d_i is in off-chip only
$M_{SPW}(d_i)$	Whether d_i is in scratchpad

Table 1: Variables in the integer linear programming formulation

5.2.2 Objective Function and Constraints for Actor Allocation

The objective of our allocation algorithm is to generate a cost-optimal allocation. We consider memory access time as the cost criterion. We will therefore seek to find the allocation that minimizes the total access time in each state.

We formulate two separate ILP problems: one to optimize allocation of actor code and the other to optimize allocation of data buffers. Because we have assumed that the scratchpad memory for actor code and data buffers is separated, these two problems can be solved independently. (Combining the two problems into one helps to remove this assumption.)

The access time for accessing actor code can be computed with the following formula:

$$\begin{aligned} Obj_1 = \sum_{i=1}^{A} \Big(& M_{Offchip}(a_i)\big(T_{Offchip_{Rd}} \cdot F(a_i) \cdot S(a_i)\big) \\ & + M_{SPW}(a_i)\big(T_{SPW_{Rd}} \cdot F(a_i) \cdot S(a_i) \\ & + T_{Offchip \rightarrow SPW} \cdot S(a_i)\big)\Big) \end{aligned} \tag{1}$$

Obj_1 is the objective function that we minimize in the first ILP problem. The first term specifies the time spent in memory reads from off-chip memory. (*Observation 1, 2*) 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*)

The constraints are: $\forall i \in [1, A]$,

$$M_{Offchip}(a_i) + M_{SPW}(a_i) = 1$$

$$\sum_{i-1}^{A} \big(M_{SPW}(a_i) \cdot S(a_i)\big) \leq Size_{InstrSPM}$$

The first set of constraints ensures that each actor is located either in both the scratchpad and off-chip memory, or in off-chip memory only. The second set of constraints ensures that the sum of the sizes of all the actors assigned to the scratchpad does not exceed the size of the scratchpad. (Due to *Assumption 2*, there is no limit on off-chip memory.)

5.2.3 Objective Function and Constraints for Data Buffer Allocation

The objective function for data buffers is more complicated for two reasons:

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

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

$$\begin{aligned} Obj_2 = \sum_{i=1}^{D} \Big(& M_{Offchip}(d_i)\big(T_{Offchip\ Wr} \cdot prod(d_i) \cdot F_{source}(d_i) \\ & + T_{Offchip\ Rd} \cdot cons(d_i) \cdot F_{dest}(d_i)\big) \\ & + M_{SPW}(d_i)\big(T_{SPW\ Wr} \cdot prod(d_i) \cdot F_{source}(d_i) \\ & + T_{SPW\ Rd} \cdot cons(d_i) \cdot F_{dest}(d_i) \\ & + T_{Offchip \rightarrow SPW} \cdot I(d_i) + T_{SPW \rightarrow Offchip} \cdot I(d_i)\big)\Big) \end{aligned}$$

Obj_2 is the objective function we minimize for the second ILP problem. 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 solution of the ILP problem, we decide to eliminate it from the expression. The first term specifies the time spent in memory accesses from off-chip memory. (*Observation 5, 6*) 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*) The summation ensures that the memory access times for all data buffers are taken into consideration.

The constraints for data buffers are similar to constraints for actor code allocation: $\forall i \in [1, D]$,

$$M_{Offchip}(d_i) + M_{SPW}(d_i) = 1$$

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

5.2.4 Extension to the ILP Formulation for Multiple Memories in the Hierarchies

The ILP problems can be extended to allow optimization for multiple memories in the hierarchy. 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. In that case, certain variables would have to be modified as in Table 2.

Variable	Number of memory units
$Size_j$	Size of the j-th memory unit in bytes ($j \in [1, U]$)
${T_j}_{Rd}$	Time to read a byte from the j-th memory unit in cycles ($j \in [1, U]$)
${T_j}_{Wr}$	Time to write a byte to the j-th memory unit in cycles ($j \in [1, U]$)
$T_{j \to Main}$	Time to move a byte from the j-th memory unit to main memory unit in cycles ($j \in [1, U]$)
$T_{Main \to j}$	Time to move a byte from main memory unit to the j-th memory unit in cycles ($j \in [1, U]$)

Table 2: Modified variables for memory hierarchies

There have to be Boolean variables for every memory unit:

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

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

The objective functions would now have to include a double summation to account for all memory units.

$$Obj_1' = \sum_{j=1}^{U} \sum_{i=1}^{A} \Big(M_j(a_i) \big({T_j}_{Rd} \cdot F(a_i) \cdot S(a_i) + T_{Main \to j} \cdot S(a_i) \big) \Big)$$

$$Obj_2' = \sum_{j=1}^{U} \sum_{i=1}^{D} \Big(M_j(d_i) \big({T_j}_{Wr} \cdot prod(d_i) \cdot F_{source}(d_i) + {T_j}_{Rd} \cdot cons(d_i) \cdot F_{dest}(d_i) + T_{Main \to j} \cdot I(d_i) + T_{j \to Main} \cdot I(d_i) \big) \Big)$$

The constraints are also modified as follows: $\forall a_i \in [1, A]$,

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

Variable	Meaning
$E_{Offchip\,Rd}$	Energy consumed to read a byte from off-chip memory in cycles
$E_{Offchip\,Wr}$	Energy consumed to write a byte to off-chip memory in cycles
$E_{SPW\,Rd}$	Energy consumed to read a byte from scratchpad in cycles
$E_{SPW\,Rd}$	Energy consumed to write a byte to scratchpad in cycles
$E_{Offchip \rightarrow SPW}$	Energy consumed to move a byte from off-chip to scratchpad in cycles
$E_{SPW \rightarrow Offchip}$	Energy consumed to move a byte from scratchpad to off-chip in cycles

Table 3: Energy cost variables

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

5.2.5 Extension to the ILP Formulation for Energy Consumption

The ILP problem can also be extended to minimize the energy consumed for migrating code or data from off-chip memory to scratchpad and back. We replace the time variables in Table 1 with the corresponding energy consumption variables, as shown in Table 3. The constraint functions remain unchanged. The objective functions become:

$$\begin{aligned} Obj_1'' = \sum_{i=1}^{A} \Big(& M_{Offchip}(a_i)\big(E_{Offchip\,Rd} \cdot F(a_i) \cdot S(a_i)\big) \\ & + M_{SPW}(a_i)\big(E_{SPW\,Rd} \cdot F(a_i) \cdot S(a_i) \\ & + E_{Offchip \rightarrow SPW} \cdot S(a_i)\big)\Big) \end{aligned}$$

$$\begin{aligned} Obj_2'' = \sum_{i=1}^{D} \Big(& M_{Offchip}(d_i)\big(E_{Offchip\,Wr} \cdot prod(d_i) \cdot F_{source}(d_i) \\ & + E_{Offchip\,Rd} \cdot cons(d_i) \cdot F_{dest}(d_i)\big) \\ & + M_{SPW}(d_i)\big(E_{SPW\,Wr} \cdot prod(d_i) \cdot F_{source}(d_i) \\ & + E_{SPW\,Rd} \cdot cons(d_i) \cdot F_{dest}(d_i) \\ & + E_{Offchip \rightarrow SPW} \cdot I(d_i) \\ & + E_{SPW \rightarrow Offchip} \cdot I(d_i)\big)\Big) \end{aligned}$$

For a combined optimization on both energy consumption and access time discussed previously, the energy or time variables can be replaced with cost variables. The cost variables are defined as weighted products of energy and time values to proportionally account for both energy consumption and access time.

5.3 Limitations and Improvements of the Allocation Algorithm

The quality of optimization results is dependent on the memory requirements of the actor code and data buffers. These memory requirements can be improved by performing schedule based optimizations as discussed in [9]. The size of data buffers can also be reduced by using techniques such as modulo addressing [9]. Hence, one could envision using these optimizations as a pre-processing step to our memory allocation algorithm in order to improve the quality of the results.

We make a simplification in our approach by not considering the actor code and data buffers that are in the scratchpad prior to a state change. Therefore, our formulation of the optimization problem includes in the cost the time to load all the values needed by the next iteration. An improved version would take into account the data that are already in the scratchpad and not require to load them again. However, to come up with such a solution, we need to consider the execution history of the HDF model, which is not know *a priori*. This requires a run-time algorithm that gives rise to expensive overhead, and hence is not practical.

Considering execution history in the allocation, as compared to the current allocation considering only the state itself, should improve the overall performance. In the next section we present a modification to the original scheme for generating allocations for a particular state taking into account a finite number of states in the execution history before that state is entered. The primary purpose of this modification is 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), etc. While a complete exploration of multiple allocations per state is beyond the scope of this paper, it definitely serves as a promising direction for future work in this field.

5.4 Memory Allocation with Execution History

In the previous section, we consider memory allocation for each state separately. According to our experiment in the next section, that mechanism based on individual states yields satisfactory result and outperforms the approach based on caches.

In this section we present a modification of the original allocation algorithm, which allows us to generate allocations based on finite execution histories. 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

Variable	Meaning
S	Number of valid states in the HDF model
s_i	i-th state ($i \in [1, S]$)
$I(s_i)$	Number of states with transitions into state s_i
$P_j(s_i)$	The j-th state with a transition 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 ($j \in [2, \alpha(s_i)]$)
$M_{Offchip}^{Predecessor}(a_i, P_j(s_i))$	Whether a_i was in off-chip in the predecessor state $P_j(s_i)$

Table 4: Additional variables for multiple allocations

the off-chip memory to the scratchpad, every time an actor is brought into or evicted from the scratchpad. By taking into account the execution histories, we seek to lower this migration cost.

The reason for considering execution history is that the cost of migrating actor code between states can be reduced if the code is in the scratchpad memory in the predecessor state. In that case, there is no need to evict the code and to load it again. This, of course, requires considering the predecessor state in the optimization of the current state. If there are S states in the HDF model, then the possible number of combinations of predecessor state and current state is at most $S \times S$.

We could further extend this by considering history of at most k steps before the current state is entered. We call this the k-lookback optimization, where $k \geq 0$. The number of combinations is S^{k+1} in general. We will illustrate this approach in this section for $k = 1$.

5.4.1 Formulation of Additional Variables

For this modification, we preserve all the variables and parameters introduced in Section 5.2.1. The additional variables are shown in Table 4.

$M_{Offchip}^{Predecessor}(a_i, P_j(s_i))$ is a Boolean variable that identifies whether a particular actor was in off-chip memory in the predecessor state. Note that this is not a variable for solution by the linear program, but rather a variable that is

set to the appropriate 0 or 1 value when composing the objective function.

$$M_{Offchip}^{Predecessor}(a_i, P_j(s_i)) = \begin{cases} 1 & \text{if actor } a_i \text{ was in off-chip in the} \\ & \text{predecessor state } P_j(s_i) \\ 0 & otherwise \end{cases}$$

5.4.2 Objective Function and Constraints for Actor Allocation

For any state s_i and predecessor state $P_j(s_i)$, we optimize the allocation with this modified objective function:

$$\begin{aligned} Obj''' = \sum_{i=1}^{A} \Big(& M_{Offchip}(a_i)\big(T_{Offchip} \cdot F(a_i) \cdot S(a_i)\big) \\ & + M_{SPW}(a_i)\big(T_{SPW_{Rd}} \cdot F(a_i) \cdot S(a_i) \\ & + M_{Offchip}^{Predecessor}(a_i, P_j(s_i)) \cdot T_{Offchip \rightarrow SPW} \\ & \cdot S(a_i)\big)\Big) \end{aligned} \tag{2}$$

The only change is that the migration cost of moving an actor from off-chip to scratchpad has been augmented by $M_{Offchip}^{Predecessor}(a_i, P_j(s_i))$. 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 5.2.2.

5.4.3 Procedure for Generating Allocations for All Predecessors

This allocation scheme generates $\alpha(s_i) = 1 + I(s_i)$ allocations for state s_i. These include an allocation for that state without considering any predecessor (initial allocation), and $I(s_i)$ allocations for the $I(s_i)$ predecessor states. The procedure for generating these allocations is as follows:

1. For each state s_i, generate initial allocation using the original objective function (Eq. 1), and obtain the optimal allocation $\alpha_1(s_i)$.

2. For each state s_i and each $P_j(s_i)$ (where $1 \leq j \leq I(s_i)$), generate the $(j+1)$-th allocation for s_i, $\alpha_{j+1}(s_i)$, using the modified objective function (Eq. 2) and setting $M_{Offchip}^{Predecessor}(a_i, P_j(s_i))$ according to $\alpha_1(P_j(s_i))$, which was computed in step 1.

With this procedure, we generate all the $\alpha(s_i) = 1 + I(s_i)$ allocation schemes for each state s_i statically. These allocation schemes are stored in a table that can be looked up at run time. The indices are numbers in $[1, \alpha(s_i)]$ that refer to the predecessor states, and the entries are the allocation to take effect on a transition into s_i. For example, when the system transitions into s_i from predecessor state $P_j(s_i)$, the entry associated with index $j+1$ in s_i's allocation

MEMORY CHARACTERISTIC		SPECIFICATION
SPW	Read Latency	1 cycle
	Write Latency	1 cycle
	Transfer Time (to or from off-chip)	2.5 cycles
Cache	Hit Latency	1 cycle
	Miss Latency	12 cycle
	Associativity	Fully Associative
	Replacement Strategy	Least Recently Used

Table 5: Memory specification used for experiments

table is fetched. According to that allocation, the actor code is loaded only if it is not in the scratchpad previously.

5.5 Observations and Analysis of the Modified Allocation Algorithm

The modified allocation algorithm is a first step to motivate further exploration of generating 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 Trellis diagram in which one can transition from one state to any state. Let the number of states in the system be S. There are at most S^2 valid predecessor-successor state pairs. There are at most S possible input paths into each state, i.e. $S + 1$ memory allocations per state. Thus the total number of allocations that need to be generated is no more than $S \cdot (S + 1)$. Also, the total number of allocations that need to be stored is $S^{k+1} \cdot S \cdot (S + 1)$ in general, since one allocation is generated for each possible allocation for the predecessor state. This worst case scenario can prove prohibitively expensive. Fortunately, it is unlikely to have a fully connected HDF model in practice. Hence, it is arguable that the above algorithm is still feasible.

6 Performance Analysis

We evaluate the performance of our allocation algorithm and compare it with a cache 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 [11], a Java-based framework for studying modeling, simulation and design of concurrent real-time systems. The open-source linear programming system, LP Solve [7], was used as the solver for the ILP problems. We applied our algorithm to the adaptive coding model [39], as well as a set of randomly generated HDF models.

Table 5 summarizes the memory characteristics considered for our experiments. The scratchpad memory forms the first level, which has a 1-cycle read/write latency. We assume a two-level memory hierarchy in our experiments. The second level off-chip memory has a 10-cycle read/write latency. Direct memory access (DMA) and pseudo-DMA mechanisms greatly speed up data transfer between the scratchpad and the off-chip memory, as in the Motorola MCORE processor [13]. Transfer times assuming DMA and pseudo-DMA mechanisms, have been analyzed by Udayakumaran et al. in [32]. Basing on this result, the time for data transfer between the scratchpad and the off-chip memory is assumed to be 2.5 cycles. We use cache with LRU as the comparison case. The cache is assumed to be a fully-associative cache, with a 1-cycle hit and a 12-cycle miss latency. An LRU replacement strategy is assumed for the cache. The cache is assumed to be a write-back cache with a write-allocate miss policy [18]. In order to better analyze the true performance of our algorithm, the sizes of the cache and the scratchpad 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 code and data.

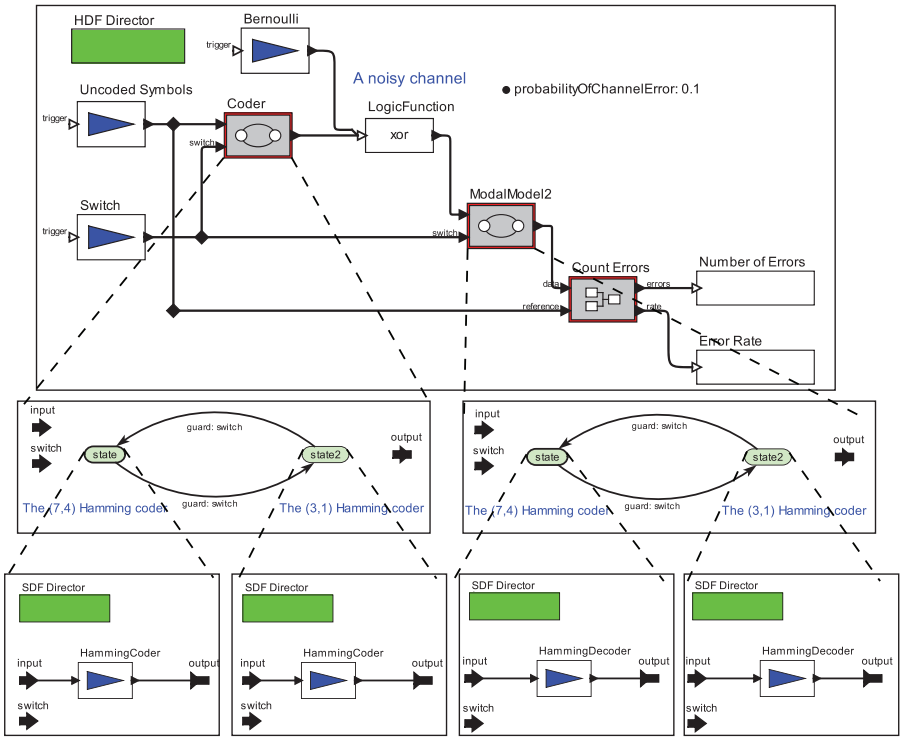
6.1 Adaptive Coding Example

The adaptive coding model [39] 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 6(a) and 6(b) in all its levels of hierarchy. Figure 6(b) shows the inside of the CountErrors actor. The details of the adaptive coding model that are relevant to the allocation algorithm are summarized in Table 6.

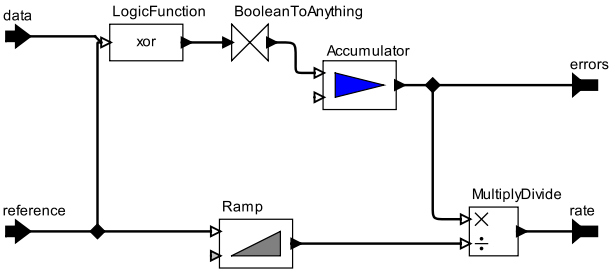
6.1.1 Restricting to Single Appearance Schedules

Note that the total memory access time for cache is highly dependent on the firing schedule of the actors in a state. We use 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:

For the alternative schedule, the actor firings are not always in succession. Dependent on the size of the cache there might be significant conflict misses



(a) Expanded toplevel model



(b) Expanded view of the CountErrors actor

Figure 6: Adaptive Coding Model

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 6: Relevant characteristics of adaptive coding HDF model

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))

when 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 scratchpad allocation algorithm is completely independent of the firing schedule. The time would be the same for both the schedules given above. Therefore, the scratchpad allocation algorithm provides us with better predictability than the use of a

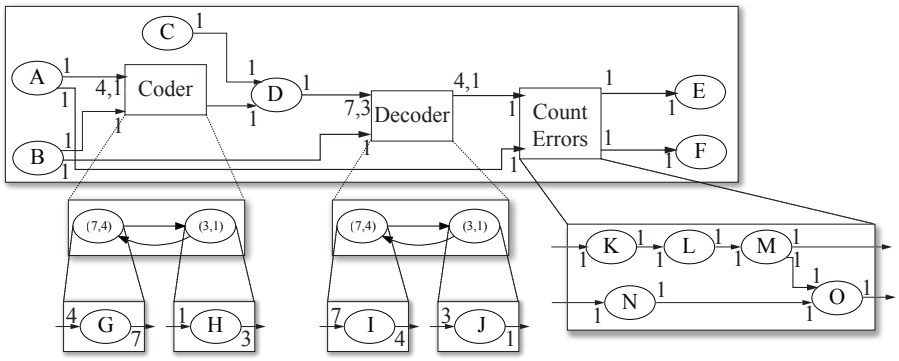


Figure 7: Adaptive Coding Model with Consumption and Production rates and Actor labels

cache.

6.1.2 Performance Analysis for Actor Allocation for the Adaptive Coding Model

We first analyze the performance of the allocation algorithm for actor code. Figure 6 shows the adaptive coding model's implementation in Ptolemy II as an HDF graph. The same model is shown in Figure 7 with actors alphabetically labeled (Actor A, Actor B, and so on) and the production and consumption rates. 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.

Figure 8 shows the number of actors, out of the total 13, allocated to the scratchpad memory for varying scratchpad sizes. As expected, the number of actors allocated to the scratchpad increases with increasing scratchpad sizes.

Figure 9 shows the percentage of memory accesses that hit the scratchpad with increasing scratchpad sizes. The number of accesses to the scratchpad increases sharply with increase in scratchpad sizes. This shows that the optimization achieves its desired purpose in allocating only such actors to the scratchpad that maximize the number of accesses to the scratchpad while minimizing net memory access time. At a reasonable scratchpad size of 35-45% of total actor size, over 60-65% of all memory accesses hit the scratchpad.

Figure 10 shows the total actor code memory access time for a single iteration of a particular state for both cache and scratchpad. The memory access time for cache remains constant irrespective of the cache size. This is due to the

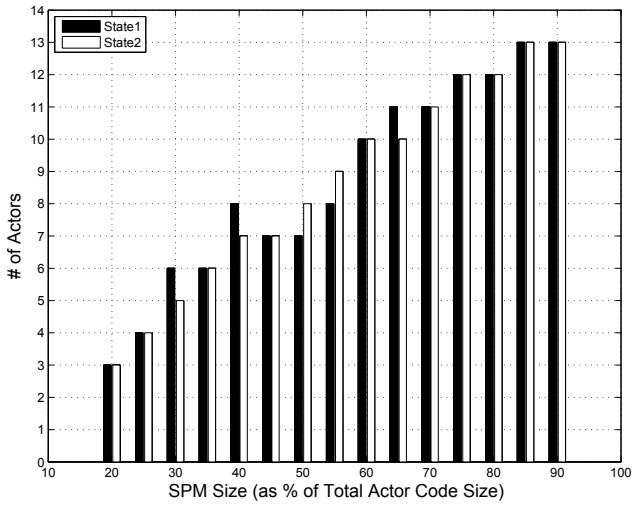


Figure 8: Number of actors allocated to scratchpad for each state

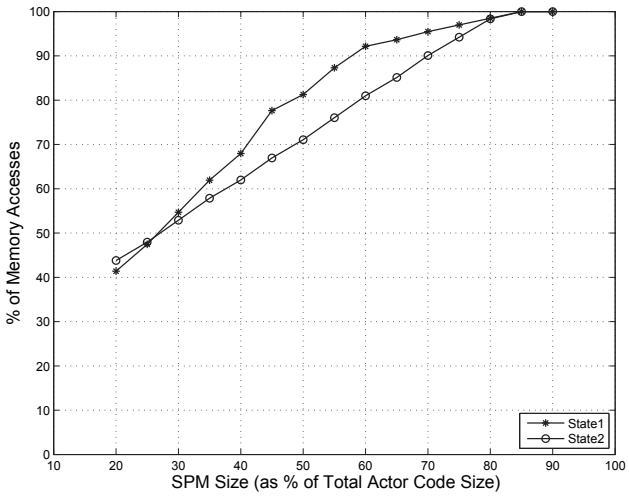


Figure 9: Percentage of Memory Accesses to Scratchpad for State 1 and State 2

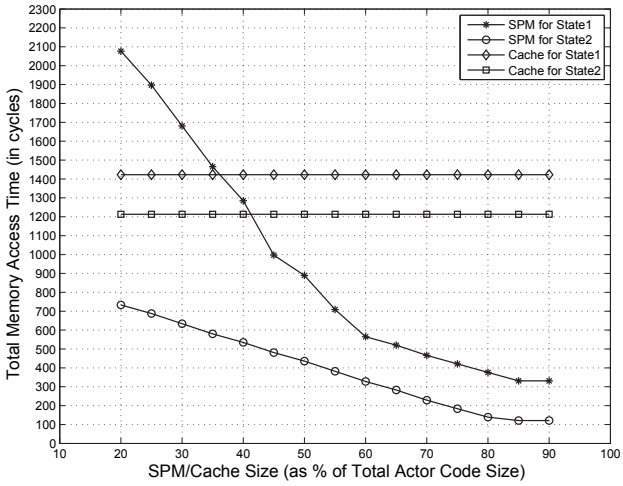


Figure 10: Total Memory Access Time for a Single Iteration of a Particular State for Scratchpad and Cache Configurations

fact that the only cache misses are compulsory misses, i.e. the first reference to an actor code causes a cache miss. Since we do not encounter any conflict or capacity misses the total access time for cache remains constant irrespective of the actual cache size. While every byte access of the actor code results in a compulsory miss for the cache, our allocation algorithm minimizes the access time costs by preloading the SPM from the memory allocation map. The precomputed memory map algorithm performs significantly better than the hardware policy implemented by the cache. The memory access time for scratchpad in State 2 is lower than the cache for all memory sizes. However, for State 1 the scratchpad shows improvements over cache for sizes of about 40% and higher. This is because all actors except I are fired multiple times in State 1; hence, the penalty for not being able to allocate such actors to the scratchpad is significantly higher.

We generate 25 execution traces of the adaptive coding model with the Ptolemy II environment [11]. Each of these traces is 15 iterations long. The total memory access times for actor code access for these traces are computed and are averaged over the executions to generate the average memory access time for a fifteen iteration long execution of this model. Figure 11 shows the comparative graph for the average memory access times using cache, the basic allocation algorithm and the multiple allocations per state modification to the original algorithm. Both the scratchpad allocation algorithms outperform the cache for all memory sizes above 25%.

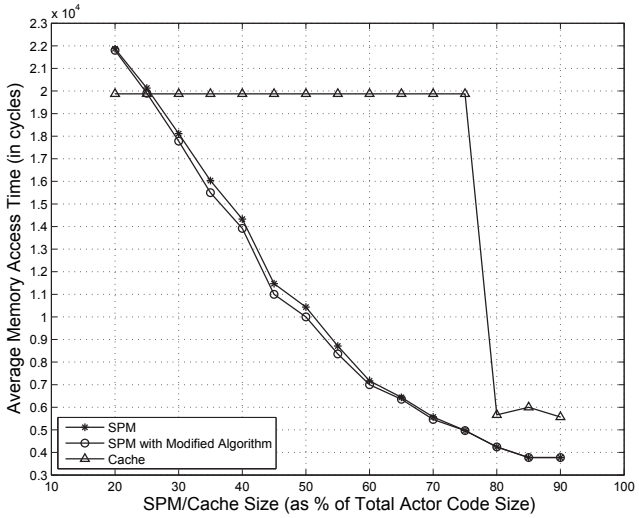


Figure 11: Average Memory Access Time for Actor Code for a Fifteen Iteration Execution of the Adaptive Coding Model

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

The performance results for the allocation of data buffers show a trend similar to that seen for actor code (Figures 8 and 9). For brevity, we omit the corresponding graphs for data allocation.

The allocation of data buffers however, involves no migration overhead during state transition. This is because there are no delay tokens to be preserved across state transitions in this case study. More generally, the total number of delay tokens present in an HDF model is often quite small and hence migration costs are considered negligible in comparison with the migration costs of actor allocation.

Figure 12 shows the comparative graph for average memory access times for data buffer access for a 15-iteration execution of the adaptive coding model for both cache and scratchpad allocation. It can be observed that the scratchpad allocation scheme performs consistently better than cache for all memory sizes, again providing evidence that the algorithm does minimize memory access time as intended. Since there is no migration cost involved, the total memory access time for data buffer access for N iterations of an 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 the cache since there is no preserved state to be maintained across state transitions, in the absence of delay tokens.

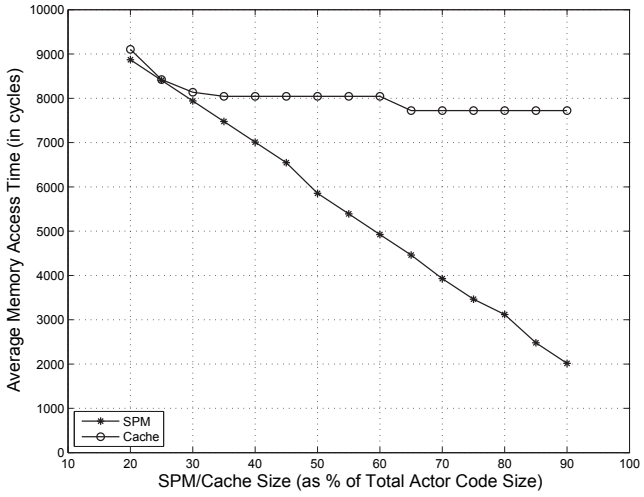


Figure 12: Average Memory Access Time for Data Buffer for a fifteen iteration execution of the Adaptive Coding Model

6.2 Randomly Generated HDF Graphs

To further analyze the performance of our algorithm, we apply our allocation scheme to a set of 50 randomly generated HDF graphs, and compare its performance to using a cache. In order to randomly generate the HDF graphs, we use several basic model templates including the adaptive coding model and the models shown in Figure 2. The production and consumption rates of the actors are chosen randomly from an interval of 1 to 10. Actor sizes are randomly chosen between 5 and 40. We consider single appearance schedules for our analysis, since this provides the best cache performance for a LRU replacement policy. Memory sizes for both scratchpad and cache are set at 35% of the total actor/data buffer sizes. The procedure used to generate the average memory access times for comparison is identical to the procedure used for Figures 11 and 12.

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

Figure 13 shows the performance improvement or deterioration for use of our algorithm for actor allocation, the modified algorithm for actor allocation and our algorithm for data allocation, respectively. The average performance im-

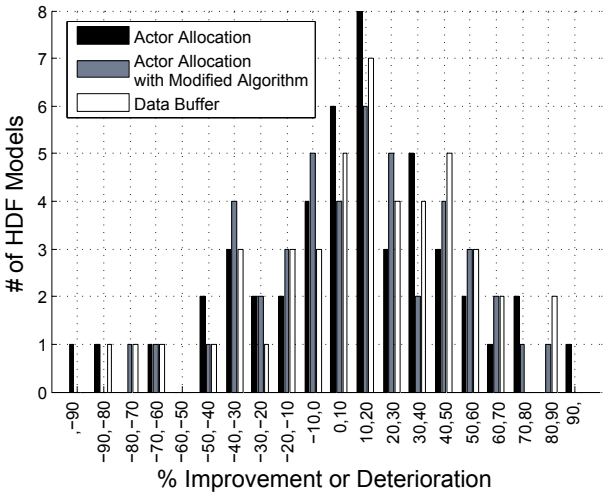


Figure 13: Performance of actor allocation, actor allocation with modified algorithm, and data buffer allocation to scratchpad versus cache for 50 randomly generated HDF graphs

provement for actor allocation is 13.43%. Using the modified algorithm result in a 15.64% average performance improvement. The data buffer allocation shows an average performance improvement of 17.24%. The fact that data buffer allocation performs better than actor allocation can be attributed to the potential for both capacity and compulsory cache misses.

6.3 ILP Runtime Analysis

The most computationally expensive stage of our allocation algorithm is the integer linear program. In order to perform the execution time analysis for the ILP solver, problems of different sizes are randomly generated. The objective function and constraints are set up to represent actor and data buffer allocation problems of varying sizes. The generated ILP problems are solved using the open-source LP Solve package [7]. The computer used for the generating the execution times has a 1.86GHz Intel Pentium M processor with 1.49GB of RAM. Figure 14 shows the execution time graph for increasing linear program size on a per state basis.

As shown in Figure 14, even for models with 850 total actors and data buffers, the execution time of the ILP solver is well below 0.5 seconds. A sharp exponential growth is observed only when we reach about 1000 actors and data buffers. We consider that many HDF models in practice would be limited to at most a couple of hundred actors/data buffers. For those HDF models, our allocation algorithm returns optimal results in a very short time.

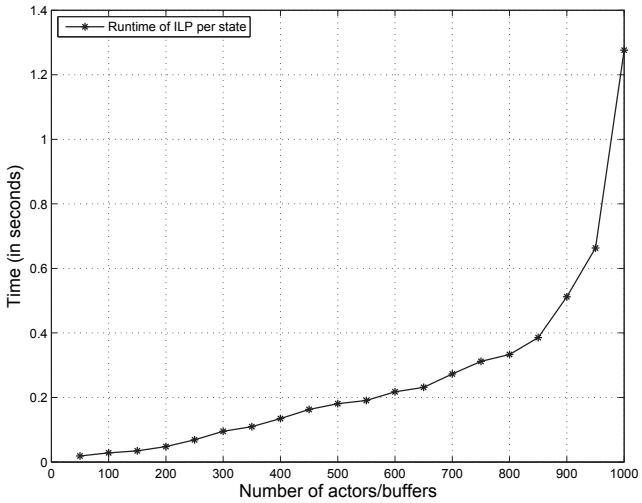


Figure 14: Execution Times for ILPs of Increasing Sizes

6.4 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 an 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 s_k is:

$$\begin{aligned} T(s_k) =& M_{Offchip}(a_i)\big(T_{Offchip\,Rd} \cdot F(a_i) \cdot S(a_i)\big) \\ &+ M_{SPW}(a_i)\big(T_{SPW\,Rd} \cdot F(a_i) \cdot S(a_i)\big) \end{aligned}$$

For some $l \in [1, S]$, let $T(S)'$ be the greatest per state memory access time amongst all the states of the HDF model such that:

$$\forall k \in [1, S], T(S)' = T(s_l) \text{ where } T(s_l) \geq T(s_k)$$

The migration cost of moving an actor block into the scratchpad during a state transition can be similarly computed by using *Observation 3*, 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 an HDF model is:

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

This is a strict upper bound that is not reachable. The reason is that if an HDF model stays 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. Let the access time for data memory for a particular state s_k be:

$$\begin{aligned} B(s_k) = \Big(& M_{Offchip}(d_i)\big(T_{Offchip\ Wr} \cdot prod(d_i) \cdot F_{source}(d_i) \\ & + T_{Offchip\ Rd} \cdot cons(d_i) \cdot F_{dest}(d_i)\big) \\ & + M_{SPW}(d_i)\big(T_{SPW\ Wr} \cdot prod(d_i) \cdot F_{source}(d_i) \\ & + T_{SPW\ Rd} \cdot cons(d_i) \cdot F_{dest}(d_i)\big) \Big) \end{aligned}$$

Now, let $B(S)'$ be the greatest per state access time for data memory and $B_{migration}(S)'$ be the greatest migration cost for a single state transition. The two bounds can be combined to form the upper bound for the overall memory access time of the model:

$$N \cdot \Big(T(S)' + B(S)'\Big) + (N-1) \cdot \Big(T_{migration}(S)' + B_{migration}(S)'\Big).$$

7 Conclusion and Future Work

We have shown that by using a dataflow model for constructing programs, we gain statically analyzable information about the application that can be exploited to get effective use of scratchpad memories. In embedded systems, this will lead to lower cost and better predictability and repeatability of the execution. To show feasibility of this approach, we developed an ILP based allocation algorithm that makes use of the coarse grained structure and semantics present in the HDF block diagram programs to generate a state-wise optimal memory allocation for scratchpad memories. We 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 cache with LRU 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 difficult to obtain for caches. We have also provided a modification to our algorithm that generates multiple allocations per

state by looking at a history of one state. The modification also serves as a beacon for revealing a whole field of possible multiple allocations per state schemes that provide better performance. This memory allocation scheme is an important step toward using information from higher-level models of computation that is otherwise difficult to extract from C/C++ programs.

Many elaborations are possible to further exploit the static information in the actor-oriented dataflow model. Specifically, we can 1) explore energy and area tradeoffs along with memory access times, 2) use profiling in computing the probabilities of the possible state transitions so that a multiple allocation algorithm can utilize the probability information, 3) extend the history of one state to a sequence of states using the probability transitions and then optimize over probable sequences of the HDF execution and 4) optimize data buffer allocation, while accounting for the dependence of data buffer sizes on the scheduling algorithm and firing schedule.

References

[1] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, September 2004.

[2] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.

[3] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of CODES*, pages 73–78, Estes Park, Colorado, May 2002.

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

[5] S. Battacharyya, P. Murthy, and E. Lee. *Software Synthesis from Dataflow Graphs*. Springer, 1996.

[6] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *Design & Test of Computers, IEEE*, 17(2):74–85, Apr-Jun 2000.

[7] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve: Open source (mixed-integer) linear programming system, May 2004. Version 5.1.0.0.

[8] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of dsp systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1948–1951, Istanbul, Turkey, June 2000.

[9] S. S. Bhattacharyya. *Compiling Dataflow Programs for Digital Signal Processing*. PhD thesis, University of California, Berkeley, July 1994. Technical Memorandum UCB/ERL 94/52.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Static scheduling of multi-rate and cyclo-static dsp applications. In *Workshop on VLSI Signal Processing*. IEEE Press, 1994.

[11] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Z. (eds.). Heterogeneous concurrent modeling and design in Java: Ptolemy II. Technical Report Technical Memorandum UCB/ERL M05/21-M05/23, University of California, July 15 2005.

[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] M. Corporation. M-core - mmc2001 reference manual, 1998.

[14] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation in embedded systems. *Journal of Embedded Computing*, 1(4), 2005.

[15] H. Falk and M. Verma. Combined data partitioning and loop nest splitting for energy consumption minimization. In *Proceedings of Software and Compilers for Embedded Systems, 8th International Workshop, SCOPES 2004*, Amsterdam, The Netherlands, September 2004.

[16] 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*, 18(6):742–760, 1999.

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

[18] J. L. Henessey and D. J. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[19] E. Lee, S. Neuendorffer, and M. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[20] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, Jan. 1987.

[21] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[22] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, Piscataway, NJ, USA, 2004. IEEE Press.

[23] A. Milidonis, N. Alachiotis, V. Porpodas, H. Michail, A. Kakarountas, and C. Goutis. A decoupled architecture of processors with scratch-pad memory hierarchy. pages 1–6, 16-20 April 2007.

[24] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 115–125, 2005.

[25] P. Panda, N. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. *European Design and Test Conference, 1997. ED&TC 97. Proceedings*, pages 7–11, 17-20 Mar 1997.

[26] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, 2006.

[27] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, pages 1–6, 16-20 April 2007.

[28] S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, March 2002.

[29] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 409–415, 2002.

[30] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Syst.*, 28(2-3):157–177, 2004.

[31] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, volume LNCS 2304, Grenoble, France, April 8-12, 2002 2002. Springer-Verlag.

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

[33] S. Udayakumaran and R. Barua. An integrated scratch-pad allocator for affine and non-affine code. *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 925–930, 2006.

[34] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, February 2004.

[35] L. Wehmeyer and P. Marwedel. Influence of onchip Scratchpad Memories on WCET Prediction. *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2004.

[36] J. Whitham and N. Audsley. MCGREP–A Predictable Architecture for Embedded Real-Time Systems. *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 13–24, 2006.

[37] J. Whitham and N. Audsley. Using Trace Scratchpads to Reduce Execution Times in Predictable Real-Time Architectures. *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008.

[38] L. Xue, M. Kandemir, G. Chen, and T. Yemliha. Spm conscious loop scheduling for embedded chip multiprocessors. pages 391–400, 2006.

[39] Y. Zhou. Communication systems modeling in Ptolemy II. Master's thesis, University of California, Berkeley, December 18 2003. Technical Memorandum No. UCB/ERL M03/53.