Scheduling of Synchronous Data Flow Models onto Scratchpad Memory-Based Embedded Processors

WEIJIA CHE and KARAM S. CHATHA, Arizona State University

In this article, we propose a heuristic algorithm for scheduling synchronous data flow (SDF) models on scratch pad memory (SPM) enhanced processors with the objective of minimizing its steady-state execution time. The task involves partitioning the limited on-chip SPM for actor code and data buffer, and executing actors in such a manner that the physical SPM is time shared with different actors and buffers (formally defined as code overlay and data overlay, respectively). In our setup, a traditional minimum buffer schedule could result in very high code overlay overhead and therefore may not be optimal. To reduce the number of direct memory access (DMA) transfers, actors need to be grouped into segments. Prefetching of code and data overlay that overlaps DMA transfers with actor executions also need to be exploited. The efficiency of the our heuristic was evaluated by compiling ten stream applications onto one synergistic processing engine (SPE) of an IBM Cell Broadband Engine. We compare the performance results of our heuristic approach with a minimum buffer scheduling approach and a 3-stage ILP approach, and show that our heuristic is able to generate high quality solutions with fast algorithm run time.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Scratchpad memory, stream computing, code overlay, scheduling, compiler, embedded systems

ACM Reference Format:
Che, W. and Chatha, K. 2013. Scheduling of synchronous data flow models onto scratchpad memory-based embedded processors. ACM Trans. Embedd. Comput. Syst. 13, 1s, Article 30 (November 2013), 25 pages.
DOI: http://dx.doi.org/10.1145/2536747.2536752

1. INTRODUCTION

Scratch pad memory (SPM) has been adopted to replace traditional caches in many embedded processors. Examples include IBM Cell Broadband Engine (Cell BE) [Pham et al. 2006], Nvidia GPU series [Owens 2007], and TI TMSC6472 [Truong 2009].[1] SPM is favored in these architectures for its reduction in hardware complexity, data access time, chip area, and power consumption. These properties are highly valued in today's multicore and embedded processor designs. Unlike cache architectures, processors with SPM require a programmer to explicitly manage the limited on-chip SPM for both code segments and data buffers. In other words, in an SPM-based architecture, the workload

[1]TMSC6472 has one L1 Data/Program memory and one local L2 memory attached to each of its six cores that can be configured as cache or scratchpad memory.

The research presented in this article was supported in parts by grants from National Science Foundation (CCF-0903513), and Semiconductor Research Corporation (SRC).
Authors' address: Arizona State University, Brickyard Engineering (BYENG) 553, 699 South Mill Avenue, Tempe, AZ 85281; email: {Weijia.Che, Karam.Chatha}@asu.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 1539-9087/2013/11-ART30 $15.00
DOI: http://dx.doi.org/10.1145/2536747.2536752

of dynamically management of the limited on-chip memory is shifted from the hardware side to a programmer.

In recent years, stream applications have been well recognized in today's multimedia and network processing domains. These applications exhibit common characteristics such as operation on large amounts of streaming data, independent actors that communicate through FIFOs, stable computation and data access pattern, and high throughput expectations. Stream applications can be modeled as synchronous data flow (SDF) graphs. Several actor-based programming languages were developed in the past few years to assist programming stream applications. Examples of stream languages include StreamIt [Thies et al. 2002], CAL [Willink et al. 2002], and Brook [Buck et al. 2004]. Compared to traditional sequential programming languages, stream languages have much simpler structures. Stream applications are good candidates to execute on SPM based processors because of their explicit code and data access patterns. This property grants a compiler the opportunity to statically schedule actor executions, and transfers of code and data between the on-chip SPM and the off-chip main memory at compile time.

The challenge of compiling an SDF based stream applications onto an SPM enhanced embedded single core comes from the fact that the limited on-chip SPM needs to be dynamically managed for both actor code and data buffer. The challenges involve scheduling of the given software application, partition of the hardware on-chip SPM, and mapping of the program code to different partitions. Different schedules of the software have different data buffer usage. A classic minimum buffer schedule tends to have many execution switches from one actor to another. Each such switch potentially introduces a code overlay overhead. On the other hand, if we only consider minimizing the number of execution switches from one actor to another, we end up with executing all instances of the same actor consecutively. As a result the data buffer usage could be very high and the available code memory is low. Data overlay could be introduced at this point, which off-loads the data from the on-chip SPM to the off-chip main memory. However, just as fetching code from the off-chip memory introduces code overlay overhead, pushing the data to the off-chip memory also introduces data overlay overhead. Finding the tradeoffs between minimizing the data buffer usage and the number of execution switches from one actor to another is one of the key issues that need to be addressed in our problem. Another key aspect that has to be resolved is the partition of the on-chip SPM. Since there is no cache in the embedded single core, every actor that gets executed needs to be physically present in the on-chip SPM. If the SPM partitions are too fine grained then many large actors will have very few physical memory regions to chose from, thus resulting in a potentially higher code overlay overhead. On the other hand, if the memory partitions are coarse grained, the total number of physical memory regions is small, which also could lead to high code overlay overhead. Further, the nonlinear behavior of DMA transfer hints that it is beneficial to group small actors into one code segment and transfer them altogether with one DMA instead of several scattered DMAs. However, this process may not improve the performance due to the fact that the actors that belong to the same segment are not necessarily always executed together.

1.1. Problem Formulation

The input to the problem is given by an SDF specification and an architecture description. The SDF specification is described by graph $G <V, E>$ where a node $v \in V$ represents an actor and an edge $e \in E$ represents a data transfer between two actors. A node v is again given by the following parameters $<C_v, \tau_v, N_v>$ and an edge e is given by $<P_{ev}, C_{ev}>$ as described in Table I. The architecture description is given by the on-chip SPM size and DMA behavior as introduced in Table I, $P <C_p, \tau_{base}, \tau_{slope}>$.

Table I. Architecture and SDF Description

Constant			Description
SDF (G)	V	C_v	Code size of actor v
		τ_v	Run time of actor v
		N_v	executions of actor v in a PASS
	E	P_{ev}	Tokens produced to edge e by actor v per iteration
		C_{ev}	Tokens consumed from edge e by actor v per iteration
Arch. (P)	SPM	C_p	Scratch pad memory size
	DMA	τ_{base}	Base latency for any DMA transfer
		τ_{slope}	Additional latency increasing rate with data size

*PASS is a periodic admissible sequential schedule.

We assume that the memory for storing library functions, global data, stack and heap has already been reserved. Consequently[2] the on-chip SPM C_p is only partitioned for actor code and data buffer. Given an SDF specification and architecture description, the objective of our techniques is to derive a PASS with SPM partitions, actor to region assignments <*V,R*>, and actor to segment assignments <*V,S*> such that the overlay overhead is minimized.[3]

2. RELATED WORK

Many previous work have been conducted over the years to statically assign program code and data to SPM based architectures. Steinke et al. [2002] presented an algorithm that selectively chooses program code and data to place in SPM. Angiolini et al. [2003, 2004] developed a post compiler technique that maps certain segments of external memory to physically partitioned banks of an on-chip SPM. Avissar et al. [2002] presented a compiler strategy that partitions global and stack data among different memory units. Nguyen et al. [2005] presented a memory allocation scheme for embedded systems where the SPM size is unknown at compile time. In contrast to these approaches, in this work, we propose heuristics that focus on dynamic management of an SPM where code segments are overlayed with each other during program run time.

There have been several previous works that address the problem of dynamic management of SPM with code overlay. Verma et al. [2004] and Verma and Marwedel [2006] discussed the dynamic management of SPM as an extension to the Global Register Allocation problem and proposed an allocation technique that copies program code and data into SPM at runtime. Egger et al. [2006] provided an integer linear programming (ILP) approach that loads required program code into the SPM on demand at runtime. Janapsatya et al. [2006] developed an optimization that utilizes concomitance metrics to determine appropriate code segments to be loaded into an SPM. Pabalkar et al. [2008] presented an ILP and a heuristic that overlay code based on static analysis of the global call control flow graph (GCCFG) of a program. While this work focusses on minimization of power or energy consumption, our work studies the dynamic management of SPM with the objective of overlay minimization.

Most recently, Baker et al. [2010] addressed instruction mapping on an SPM by partitioning it into regions and loading functions to regions. Functions assigned to the same region are overlayed with each other during program execution. Jung et al. [2010] also utilized the same function to region assignment scheme and presented two heuristics

[2]Since in a stream application we typically have no recursive calls and dynamic allocations of memories, we can assume that the stack and heap size is bounded by a constant.

[3]The code overlay overhead calculation is fairly complicated and will be discussed in our heuristic approach as a subroutine.

for generating function to region mappings. Our work distinguishes from the above two approaches in that we focus on stream applications rather than traditional C++ programs. In our scheme, a steady-state schedule of the stream program has to be generated together with code and data memory partition. Apart from actor to region assignments, we also introduce segmentation to amortize DMA cost. Further, we also extend our approach with code pre-fetching and data overlay optimizations to overlap DMA transfers with actor executions.

In the literature of scheduling SDF models on SPM enhanced embedded processors, Bandyopadhyay [2006] and Bandyopadhyay et al. [2008] present a SPM allocation scheme that makes optimal use of SPM by analyzing the structure and semantics of a heterochronous dataflow model. However, their work is focussed on static allocation of code and data, whereas in our approach we optimize for dynamic code and data overlay. Che and Chatha [2010] proposed a 3-stage ILP approach that utilizes SDF scheduling, region assignment, and segmentation to schedule an SDF on an embedded core with an on-chip SPM. Although the 3-stage ILP approach was able to generate high quality solutions, the algorithm runtime grows exponentially with the input program size. This work presents fast heuristics that are able to perform SDF scheduling, region assignment and segmentation. Further, in addition to the basic code prefetching approach, our proposed heuristic also incorporates deep code prefetching and data overlay optimizations.[4] The deep prefetching optimization tries to issue a prefetching of code at a much earlier time than a basic prefetching and data overlay optimization tries to reduce the data buffer usage of a schedule by transferring data to the off-chip memory after it is produced and retrieve it before it is consumed.

In the next section, background knowledge of StreamIt language (software specification) and IBM Cell BE (hardware specification) and code/data overlay schemes are provided. Section 4 motivates our heuristics by discussing various design tradeoffs. Section 5 discusses our SDF scheduling heuristic. Extensions with basic prefetching, deep prefetching and data overlay optimizations are provided in Section 6. Section 7 presents our experimental results, and Section 8 concludes the article.

3. BACKGROUND KNOWLEDGE

StreamIt [Thies et al. 2002] serves as the software input specification in our experiments and IBM Cell BE is utilized as the hardware platform. Note that even though IBM Cell BE has eight synergistic processing engines (SPEs), only one of them is adopted in order to mimic the behavior of a single core embedded processor with a limited on-chip SPM. In this section, we first provide background knowledge of the StreamIt language and the architecture overview of the IBM Cell BE. Then descriptions of code and data overlays and their implementation details are provided.

3.1. StreamIt

We adopted the StreamIt language from MIT [Thies et al. 2002] as the input specification to our techniques. StreamIt models stream applications as SDF models. Four basic structures, namely *filter*, *pipeline*, *split-join*, and *feedback-loop* are provided by StreamIt to construct a stream application program. The actors in the SDF model represent computationally intensive units in the stream application. The edges in the SDF model stand for data transfers or FIFOs among distinct actors. An actor can execute as long as it has sufficient data tokens on its input FIFOs. In each actor execution,

[4]The current manuscript is an extended version of our Estimedia 2011 conference publication. The primary technical contributions in the current version are the deep code pre-fetching and data overlay optimizations. We have expanded the experimental results section to evaluate the two new optimizations, and also considered the impact of scaling up code size and DMA costs, respectively.

Fig. 1. DMA performance on IBM Cell BE.

Fig. 2. Segment-region code overlay overview.

the number of tokens produced or consumed on each edge is constant in a StreamIt program.

3.2. IBM Cell BE

IBM Cell BE [Pham et al. 2006] was selected as the target architecture to evaluate the efficiency of our techniques. IBM Cell BE is a heterogeneous multicore processor collaboratively developed by IBM, Sony and Toshiba. There are nine processing elements in the Cell BE architecture with one PowerPC Engine (PPE) and eight SPEs [Flachs et al. 2005]. The PPE works as a control plane and launches tasks on SPEs. Eight SPEs run as high-performance data processing plane. Each SPE hosts a 256K SPM, formally referred to as SPE local store. The SPE local store is used to host both actor code and data buffer. Each SPE communicates with PPE and other SPEs through a four-ring structured Element Interconnect Bus (EIB) [Kistler et al. 2006].

In Figure 1, we characterize the DMA performance between an SPE and the off-chip main memory. The x-axis in the figure indicates the entry size of each DMA request in *log* scale. The y-axis depicts the latency for the corresponding DMA in microseconds. There are two curves being plotted, the *DMA latency* curve provides the average *read* latency over one thousand iterations. The *approximated latency* curve indicates the latency calculated by our approximation function. The approximation function is given by $T_c(x) = \tau_{base} + \tau_{slope} * x$, where x is the code or data size. In the equation, $\tau_{base} = 0.21us$ represents the DMA base cost and $\tau_{slope} = 7.5 \times 10^{-5} us$ indicates the linear latency increase with each additional byte of code or data. From Figure 1, we observe that with a simple linear approximation, we were able to achieve a very close estimation of the actual DMA latency. The values of τ_{base} and τ_{slope} are selected such that the estimated latency is higher than the profiled DMA latency on average.

3.3. Code and Data Overlay

3.3.1. Region-Segment Scheme. We utilize the region-segment scheme that is supported by *spu-gcc* version 4.1.1 for executing an SDF on an SPE. In region-segment scheme, as introduced in Figure 2, the following three stages are performed.

—*SDF Scheduling.* A Periodical Admissible Sequential Schedule (PASS) of the SDF is defined as a finite sequence of actor firings that brings buffers back to their initial state. In our instances, a PASS is also a valid steady-state schedule of the stream program. In SDF scheduling stage, a PASS for the given stream program is generated. The buffer usage is then calculated based on the PASS. The available memory for actor code is given by the difference of the SPM size and the buffer usage.[5]

[5]We assume that the memory of library functions, global data, stack and heap has already been subtracted from the SPM.

Fig. 3. Code prefetching and deep prefetching.

Fig. 4. DMA engine status with basic prefetching.

—*Actor to Region Assignments.* In this stage, we assign actors to regions such that each actor is mapped to one and only one region and the total region size is no more than the available code memory. The size of each region is given by the largest actor code assigned to it. During program execution, actors assigned to the same region are overlayed with each other.

—*Segmentation.* A segment is a group of actors that are moved to the SPM altogether. In the segmentation stage, we selectively group actors in the same region into segments. Each segment size is given by the sum of all actors mapped to it. To respect the memory constraint, each segment size must be no more than its region size.

In region-segment scheme, an actor is assigned to exactly one segment and each segment is assigned to one region. At any given time instant, there can be only one segment present in any region. The regions essentially represent the memory partition for code.

3.3.2. Basic Prefetching and Deep Pre-fetching. Since the DMA engine works independently from the execution unit of an embedded core, we overlap DMA transfers with actor executions. Figure 3 introduces the behavior of a basic prefetching scheme and a deep prefetching scheme. In the example, we have an actor execution sequence of A, B, C, A, D, E. Actor A is mapped to region 1, actor B and C are mapped to region 2, and actor D and E are mapped to region 3. Suppose actor C just finishes execution, the current memory state of the SPM is A in region 1, B in region 2, and C in region 3 as described in Figure 3. Without code prefetching, we will execute A, overlay D, execute D, overlay E and execute E.

Let us focus on the code overlay of actor E. Without code prefetching, actor executions of A and D and code overlay of E are sequential. The code overlay overhead equals the DMA cost of actor E. In the basic prefetching scheme, if the current actor execution introduces additional code overlay overhead, we try to overlap it with the previous actor's execution. In the basic prefetching scheme, if the previous actor resides in a different region from the current actor, we issue prefetching. Otherwise, a basic prefetching cannot be issued because of memory conflict. In Figure 3, we will initiate the DMA transfer of E before execution of D with a basic prefetching.

The deep prefetching scheme extends the basic prefetching scheme by searching backward along the PASS and issuing a prefetching as early as possible. That is, immediately after the last execution of an actor from the same region. In Figure 3, we start from D, continue with A, and we stop at C since both C and E reside in region 3. Therefore, with deep prefetching, we can start the prefetching of actor E right after actor C's execution.

Fig. 5. DMA engine status with deep prefetching.

Fig. 6. Data overlay overview.

The extension from the basic prefetching scheme to deep prefetching scheme seems reasonable and straightforward. However, there is another constraint that we need to consider. That is, we only have one DMA engine. In the basic prefetching scheme, this is not a problem since the DMA engine is guaranteed to be idle when we initiate the DMA prefetching of the next actor. This is because when we start the execution of the current actor, the previous prefetching has completed (one of the prerequisite for an actor to be executed) as introduced in Figure 4. However, this is not necessary true for deep prefetching. For example in Figure 5, we can initiate the code prefetching of E immediately after actor C' execution. However, at this time the DMA engine is still busy with prefetching of actor D's code. In this case, we utilize the largest DMA engine idle period available. In the example, actor D's execution period is utilized to overlap DMA transfer of actor E's code.

3.3.3. Data Overlay Scheme. In order to reduce the buffer usage of a given schedule, we can also introduce data overlay. The basic idea is that we do not have to keep a data token for its entire lifetime in the local SPM. A data token can be transferred to the off-chip main memory as soon as it is produced and we get it back before it is being consumed. Figure 6 introduces this data overlay scheme. Blindly incorporating data overlay to every data token will introduce additional data overlay overhead as we have to circle every data token through the off-chip main memory. Further, data overlay also uses the same DMA engine that is used by code prefetching. Unnecessarily introducing data overlay will also impact code prefetching optimization.

Given all the design challenges discussed here, we propose a fast heuristic approach that automatically schedules a stream application on an SPM equipped embedded single core. In the heuristic, the schedule evolves from a minimum buffer schedule to a minimum switching schedule. For each generated schedule, the heuristic partitions the resulting available code memory into regions and assign actors to regions. The execution pattern of the schedule is explored at this step such that actors assigned to the same region are less likely to interfere with each other. For actors assigned to the same region, we investigate options of grouping some of them together to reduce the number of DMA transfers. Then, we further introduce code pre-fetching and data overlay optimizations that investigate overlap of DMA transfers for code/data with actor executions. The contributions of this article include:

(1) a new efficient heuristic approach that extensively explores design alternatives with different schedules, code and data partitions, actor to region and segment assignments with the objective of minimizing the overlay overhead;

Fig. 7. Stream program with one producer and one consumer. The code, token sizes are in bytes.

(2) extensions to the heuristic with basic prefetching, deep prefetching, and data overlay which further improve the performance through overlapping DMA transfers with actor executions.

4. DESIGN TRADEOFFS

In this section, we discuss the design tradeoffs in each stage of scheduling a stream application on an SPM enhanced processor, including PASS generation, actor to region assignments, segmentation, and data overlay.

4.1. PASS Generation

Two properties of a PASS are particularly important in terms of scheduling SDF models on an limited SPM,

—*Buffer Usage*. Buffer usage of a PASS is the total memory required for storing the data buffer during its entire execution. The same buffer can be reused at different time intervals. In our scheme, a given SPM is partitioned into buffer and code memory. The smaller the buffer usage, the more memory can be devoted to program code.

—*Actor Switches*. Actor switches of a PASS is captured by the number of actor executions diverging from one actor to another in the PASS. A larger number of actor switches typically indicate that an actor is likely to be evicted out of the on-chip SPM after its execution and thus a higher overlay overhead.

Assume we have a program that consists of only one producer and one consumer, as described in Figure 7. Per execution, A pushes three tokens to edge A→B and B pops two tokens from the edge. In a PASS or a steady-state execution, A will have two executions and B will have three executions. A minimum buffer schedule (MBS) that achieves the smallest buffer usage is given by PASS = {A, B, A, B, B}. The buffer usage of this MBS is four tokens (or 400 bytes) and the number of actor switches is four, as introduced in Table II. Note that one of the actor switches is introduced between the last execution of B and the first execution of A. A minimum switch schedule (MSS) is given by PASS = {A, A, B, B, B}. The buffer usage of MSS is six tokens (or 600 bytes) and the number of actor switches is two.

In Table II rows 5–7, we examine the overlay cost of MBS and MSS under an SPM size of 800 bytes. In this configuration, the memory available for code in MBS is 400 bytes. Since the actor code size of A and B are also 400 bytes as given in Figure 7, the code memory is able to accommodate one and only one actor at any time interval. The code overlay overhead for MBS is equal to the cost of transferring actors A and B each twice from the off-chip memory to the on-chip SPM. Under the same configuration, the memory available for code in MSS is 200 bytes. In this case, the total code memory is less than the largest actor code size (400 bytes), therefore the program cannot execute and we set the overlay cost to be $+\infty$.

In another configuration, we set the SPM size to be 1000 bytes as introduced in Table II rows 8–10. The memory available for code in MBS is 600 bytes and in MSS, 400 bytes. In both cases, the total code memory can only accommodate one and only one actor. From Table II, MBS will have four actor switches (two A ⇒ B and two

Table II.
Design trade-offs with PASS generation. Buffer Usage and Available Code Memory in the table are represented in bytes.

Minimum Buffer Schedule		Minimum Switch Schedule	
A, B, A, B, B		A, A, B, B, B	
Buffer Usage	400	Buffer Usage	600
actor Switches	4	actor Switches	2
SPM Size = 800			
Avail. Code memory	400	Avail. Code memory	200
Overlay Cost	ABAB	Overlay Cost	$+\infty$
SPM Size = 1000			
Avail. Code Memory	600	Avail. Code Memory	400
Overlay Cost	ABAB	Overlay Cost	AB

Fig. 8. Stream program with four actors. The code sizes are in bytes.

Table III.
Design tradeoffs with actor to region assignments. Available Code Memory and Region Sizes in the Table are represented in bytes.

Avail. Code Memory = 700				
1st Region Assignments			2nd Region Assignments	
{A, B}	{C}	{D}	{A}	{B, C, D}
$R_{AB} = 400$	$R_C = 200$	$R_D = 100$	$R_A = 400$	$R_{BCD} = 300$
Overlayed Actors: A,B			Overlayed Actors: B,C,D	

B $\Rightarrow$ A) and MSS will have two actor switches (one A $\Rightarrow$ B and one B $\Rightarrow$ A). The code overlay overhead of MSS is half of the overlay overhead of MBS in this case.

From this discussion, we can see that neither MBS nor MSS is always optimal for our code overlay minimization problem. A schedule that achieves the minimum code overlay overhead should balance the buffer usage and the number of actor switches rather than concentrate on optimizing one of them.

4.2. Actor to Region Assignments

At this stage, we assume that we already have a PASS and the available code memory is calculated correspondingly. Under the scenario that not all code can fit into the available code memory, a programmer or compiler has to partition the code memory into regions and assign actors to regions. For a stream program described in Figure 8, the total code size is 1000 bytes. Table III provides two feasible actor to region assignments under a code memory of 700 bytes. One is given by {A, B} {C} {D} and the other given by {A} {B, C, D}. Actors within one pair of braces are assigned to the same region. In both solutions, the total region size is 700 bytes, indicating that the code memory constraint is satisfied. In the first solution, actors A and B are overlayed with each other and in the second solution, actors B, C, and D are overlayed with each other. Depending on the DMA behavior and the PASS from the previous stage, either of the two solutions in Table III could be superior than the other.

Fig. 9. Design tradeoffs with data overlay.

4.3. Segmentation

In the segmentation stage, we exploit the opportunities of grouping actors in the same region into segments to amortize the DMA base cost. The resulting segment size is equal to the sum of all actors assigned to it. The largest segment of each region determines the region size. For the same example given in Figure 8, assume the code memory available is 700 bytes and we adopt the second solution of actor to region assignments as presented in Table III. For region {B, C, D}, we can group actors C and D together into a segment without violating the region size. However, grouping actor C and D into one segment does not always promise us a performance improvement. Consider two feasible PASS of the stream program described in Figure 8. The first PASS is given by A, B, B, B, C, D, C, D, C, D. In this case, actors C and D are always executed together. Grouping C and D into one segment will definitely result in code overlay overhead reduction. A second PASS for the same stream program could be A, B, C, B, C, B, C, D, D, D. Now if we group actors C and D into one segment, it is likely that the code overlay overhead will increase, considering that for the first two executions of actor C, we bring in C and D together (C, D are grouped into the same segment). However, only actor C is executed before the execution of actor B evicts both C and D out of the on-chip SPM.

4.4. Data Overlay

Data overlay optimization reduces the data buffer usage by circling data tokens around the off-chip main memory and reduces the total time that they are present in the local SPM. Extra DMA transfers that are used to implement data overlay could result in extra latency overhead. For the example given in Figure 9 with PASS={A, B, C, D}. Without data overlay, when we execute actor B, there are 9 tokens alive, 1 on edge A→B, 4 on edge B→C, and 4 on edge A→D. This is also the time interval with the largest buffer usage, which is 900 bytes. With data overlay, we can transfer the 4 tokens on edge A→D to the off-chip main memory after A finishes execution and then retrieve the data tokens before the execution of D. In this case, the buffer usage is 500 bytes. The minimum memory requirement without data overlay thus is 900 (buffer usage) + 100 (code overlay region) = 1000 bytes. With data overlay, the minimum memory requirement is 500 (buffer usage) + 100 (code overlay region) = 600 bytes. Data overlay optimization reduces the buffer usage at the expense of potential increase of data overlay overhead. Another side effect of data overlay optimization is the occupancy of DMA engine, which could potentially block the code pre-fetching discussed in Section 3.3.2.

5. SDF SCHEDULING HEURISTIC

In this section, we present an SDF Scheduling heuristic that schedules an SDF on an SPM with code overlay. In this basic approach, a PASS for the given SDF is generated simultaneously with actor to region and actor to segment assignments. The objective is to minimize code overlay overhead.

ALGORITHM 1: $calCodeOverlay(G, PASS, <V,R>, <V,S>)$

1: $code_overlay \leftarrow 0$
2: **for** $r \in R$ **do**
3: /* s_{last} *is the last segment that is loaded to region r following PASS* */
4: Initialize $mem_state[r] \leftarrow s_{last}$
5: **end for**
6: **for** $i \in [0, |PASS| - 1]$ **do**
7: $s_{cur} \leftarrow getSegment(<V,S>, i)$
8: $r_{cur} \leftarrow getRegion(<V,R>, i)$
9: **if** $s_{cur} \neq mem_state[r_{cur}]$ **then**
10: $code_overlay \leftarrow code_overlay + T_c(C_{s_{cur}})$
11: $mem_state[r_{cur}] \leftarrow s_{cur}$
12: **end if**
13: **end for**
14: **return** $code_overlay$

5.1. Code Overlay Overhead Calculation

Prior to discussion of our SDF scheduling heuristic, we provide the calculation of code overlay overhead as a subroutine in Algorithm 1. In the algorithm, we first initialize *code_overlay* to be 0, Line 1. *mem_state* is an array that keeps track of the segments that are present in each region. We assume the SDF is being executed in an iterative manner. Therefore the segment that is in each region before the current execution is given by the last segment that is loaded to each region in the previous execution (Line 2–4). For each actor execution in the given PASS, the subroutine checks whether segment s_{cur} that contains the actor is already loaded to its corresponding region r_{cur}. If segment s_{cur} is absent from r_{cur}, we increase *code_overlay* by $T_c(C_{s_{cur}})$ (Line 10). $T_c(C_{s_{cur}})$ calculates the DMA cost for transferring segment s_{cur} from the off-chip memory to the local SPM. T_c is the DMA cost function that is discussed in Section 3.2. $C_{s_{cur}}$ is given by $C_{s_{cur}} = \sum_{v \in s_{cur}} C_v$. Then, $mem_state[r_{cur}]$ is updated with s_{cur}, indicating segment s_{cur} is loaded into region r_{cur} (Line 11). After iterating through the entire PASS, *code_overlay* is returned.

5.2. Overall Description

A high-level description of our heuristic approach is given in Algorithm 2. In the algorithm we first initialize the overlay overhead *min_overlay* to be infinitely large. The initial PASS is set to be a minimum buffer schedule of the given SDF [Jantsch 2003]. We deliberately evolve the PASS from a minimum buffer schedule to a minimum actor switching schedule in this algorithm. The buffer usage *buf_mem* and SPM memory available for code, *code_mem*, is calculated based on the given PASS. Starting from Line 5, we enter an iterative procedure where at each iteration, we perform actor to region assignments (*RegionAssignment*) and actor to segment assignments (*Segmentation*). The implementation details of the *RegionAssignment* and *Segmentation* are given by Algorithm 3 and Algorithm 4 with the discussions provided in Sections 5.3 and 5.4, respectively. The total region size after *RegionAssignment* is calculated by $\sum_{r \in R} C_r$, where C_r denotes the size of region r and is given by $max_{v \in r} C_v$. Line 7 in Algorithm 2 checks whether *RegionAssignment* is successful. If *RegionAssignment* succeeds, further actor to segment mapping is generated and *cur_overlay* is updated accordingly, Line 9. If *cur_overlay* is less than *min_overlay*, we update *min_overlay*. The function *clone* makes a copy of the current PASS, actor to region and segment assignments, Line 12.

ALGORITHM 2: $MinOverlayScheduling(G, P)$

1: Initialize $min_overlay \leftarrow +\infty$
2: Initialize $PASS \leftarrow MinBufferScheduling(G)$
3: Calculate current buffer usage, buf_mem, of $PASS$
4: $code_mem \leftarrow C_p - buf_mem$
5: **repeat**
6: <V,R> ← *RegionAssignment*($G, PASS, code_mem$) /**Actor to region assignment**/
7: **if** $\sum_{r \in R} C_r \leq code_mem$ **then**
8: <V,S> ← *Segmentation*($G, PASS$, <V,R>) /**Actor to segment assignment**/
9: $cur_overlay \leftarrow calCodeOverlay(G, PASS$, <V,R>, <V,S>) /**Overlay overhead**/
10: **if** $cur_overlay < min_overlay$ **then**
11: $min_overlay \leftarrow overlay$
12: $solution \leftarrow clone(G, PASS$, <V,R>, <V,S>)
13: **end if**
14: **end if**
15: **until** $collapseTwoExecs(PASS) = false$ /*Evolve from Min. Buf to Min. Switch*/
16: **return** $solution$

ALGORITHM 3: $RegionAssignment(G, PASS, code_mem)$

1: Initialize actor to region assignments <V,R>, as each actor occupies a separate region
2: Construct IF table entry for each region pair $<(r_i, r_j), Integer>$, where $r_i, r_j \in R, i < j$
3: $region_mem \leftarrow \sum_{r \in R} C_r$
4: **while** $region_mem > code_mem$ and $|R| > 1$ **do**
5: Collapse a region pair with minimum IF
6: Update <V,R> and IF table
7: $region_mem \leftarrow \sum_{r \in R} C_r$
8: **end while**
9: **return** <V,R>

After current evaluation, we generate the next PASS to be evaluated (Line 15). We implement this procedure by *collapseTwoExecs()*. Function *collapseTwoExecs()* creates a copy of the current *PASS* for every actor (say v_{cur}). In the temporary *PASS* associated with an actor v_{cur}, we check whether there is another execution v_{next} of the same actor v_{cur} which does not immediately follow v_{cur}. We remove v_{next} from the temporary *PASS* and insert it right after v_{cur}. We validate the new *PASS* by checking the data tokens on each edge at every time interval. If there are no negative tokens on any edge at any time interval, the new *PASS* is legal. If a *PASS* is illegal, it is discarded. Among all legal *PASS*es, we select the one that has the least buffer usage increment as the next *PASS* to be evaluated. The procedure terminates when no two non-consecutive executions of the same actor can be found. Upon termination, a solution that consists of a PASS, actor to region, and actor to segment assignments is returned.

5.3. Region Assignments

For a given SDF, Algorithm 3 assigns actors to regions such that each actor is mapped to one and only one region and all regions fit into the available code memory. In this stage, we assume that each actor occupies a separate segment. Since the actors being assigned to the same region are overlayed with each other over time, we would like them to interfere with each other as little as possible. In the algorithm, we utilize the number of times that the actors belonging to two regions interfere with each other during the program execution to define their interaction factor (*IF*). In Algorithm 3, we

ALGORITHM 4: *Segmentation*(G, $PASS$, <*V,R*>)

```
1: Initialize actor to segment assignments <V,S>, as each actor occupies a separate segment
2: min_overlay ← calOverlay(G, PASS, <V,R>, <V,S>)
3: for each region r ∈ R do
4:   for each segment pair (s_i, s_j), where s_i, s_j ∈ R, i < j do
5:     if collapsing s_i and s_j does not violate region size then
6:       Update <V,S>
7:       overlay ← calOverlay(G, PASS, <V,R>, <V,S>)
8:       if overlay < min_overlay then
9:         min_overlay ← overlay
10:      else
11:        Restore <V,S> to the previous state where s_i and s_j were not collapsed
12:      end if
13:    end if
14:  end for
15: end for
16: return <V,S>
```

first initialize actor to region assignments <*V,R*> such that each actor is assigned to a different region. We construct *IF* table for every pair of regions $<r_i, r_j>$ by iterating through the given PASS. If there is an execution switching from region r_i to region r_j or vice-versa, we increase *IF* of $<r_i, r_j>$ by one. The current total region size *region_mem* is calculated based on the actor to region mapping. We keep on collapsing two regions with the smallest *IF* while the total region size is larger than *code_mem* and the number of regions is more than one, Line 4. At each iteration, we collapse a region pair with minimum *IF* by moving all actors from r_j to r_i. If there are several region pairs that have the same minimum IF, we collapse the region pair that decreases *region_mem* the most. After each collapsing of a region pair, we update actor to region assignments <*V,R*>, IF table and recalculate *region_mem*. The algorithm terminates when the first actor to region mapping <*V,R*> that fits into *code_mem* is found or $|R| = 1$. The complexity of *IF* table construction and the iterative procedure are both $O(n^3)$, where n is the number of actor executions in the given *PASS*. Therefore, the complexity of *RegionAssignment* algorithm is $O(n^3)$.

5.4. Segmentation

The actor sizes assigned to each region could be very diverse and the DMA base cost may be overwhelming if we have too many DMA transfers. In the segmentation phase, we explore opportunities of combining actors (that are in the same region) into segments to amortize the DMA base cost. In Algorithm 4, we initialize the actor to segment mapping $<V, S>$, as each actor occupies a different segment. The minimum overlay for the given PASS, *min_overlay*, is calculated based on the current PASS, <*V,R*>, and <*V,S*> mappings. Then we start an iterative procedure where for each region, we examine combining every pair of segments s_i and s_j for opportunities of code overlay reduction. Note that even if two segments in the same region can be grouped together, it does not necessarily promise a performance gain (refer to the discussion provided in Section 4.3). In Algorithm 4, if collapsing the current segment pair does not violate the region size constraint, then we try to update the actor to segment mapping by moving all actors from s_j to s_i. If the overlay overhead after collapsing s_i and s_j is less than *min_overlay*, we update *min_overlay*. In the next iteration s_i is re-evaluated with every other segment. Otherwise, we restore the actor to segment mapping to the previous state where s_i and s_j are recognized as separate segments. The exhaustive search in

Algorithm 4 is less expensive than enumerating every pair of segments in S and the complexity of $calCodeOverlay$ is $O(n)$. Therefore, the complexity of Algorithm 4 is $O(n^3)$.

5.5. Algorithm Complexity

Given that the procedures of actor to region assignment and segmentation are both in $O(n^3)$ and they are nested in the loop of PASS generation, which is $O(n)$, the overall complexity of our heuristic is $O(n^4)$.

6. EXTENSIONS TO SDF SCHEDULING HEURISTIC

In this section, we extend our SDF scheduling heuristic with three optimizations, namely, basic code prefetching, deep code prefetching, and data overlay. We discuss each of the optimizations in the following sections.

6.1. Basic Code Prefetching Optimization

We first incorporate the basic code prefetching optimization into our existing approach. Basic code prefetching optimization tries to overlap DMA transfer of the current segment with the previous actor's execution. We denote the current actor being executed as v_{cur} and the previous actor executed as v_{pre}. The corresponding regions and segments for the current actor and the previous actor are r_{cur}, r_{pre}, s_{cur}, and s_{pre}, respectively. If s_{cur} is absent from the local SPM, we try to overlap DMA transfer of s_{cur} with v_{pre}'s execution. That is, we try to issue a prefetching of s_{cur} right before the execution of v_{pre}. The overlay overhead with this basic prefetching optimization is calculated based on the following scenarios.

—*Case* I. If s_{cur} resides in the same region with v_{pre}, then code prefetching for s_{cur} cannot be issued because of memory conflict. In this case, DMA transfer for s_{cur} is given by $T_c(C_{s_{cur}})$ (the calculation of $T_c(C_{s_{cur}})$ is provided in Section 5.1).

—*Case* II. s_{cur} does not reside in the same region with v_{pre}. In this case, a code prefetching for segment s_{cur} can be issued before v_{pre} starts execution. The DMA transfer for s_{cur} is overlapped with v_{pre}'s execution and the resulting code overlay overhead is given by $T_{overlap}(s_{cur}) = max(0, T_c(C_{s_{cur}}) - \tau_{v_{pre}})$, where $\tau_{v_{pre}}$ indicates the run time of actor v_{pre}. If DMA of s_{cur} is less than τ_{pre}, then 0 is returned. Otherwise, the DMA cost that exceeds v_{pre}'s execution is returned.

We incorporate this basic prefetching in our original heuristic approach and update the code overlay overhead calculation as shown in Algorithm 5. In Algorithm 5, the initializations of *code_overlay* and *mem_state* are identical to Algorithm 1. Then we calculate the code overlay overhead of each actor execution with basic prefetching, Line 6-18. The last actor execution is treated as the previous actor execution of the first actor in a PASS (Line 9 when $i = 0$). The total code overlay overhead is returned after iterating through the entire PASS, Line 19.

6.2. Deep Prefetching Optimization

The basic prefetching optimization only considers overlapping the DMA fetch cost of s_{cur} with previous actor v_{pre}'s execution. However, the prefetching can be issued at a much earlier time to grant longer execution time to overlap with a DMA transfer. Essentially, we can trace back the given PASS and identify the last actor execution v^r_{pre} that resides in the same region with v_{cur}. A prefetching for s_{cur} can be issued as soon as v^r_{pre} finishes its execution. With this deep prefetching optimization, we may over use the DMA engine, meaning there could be several concurrent DMA transfers on-the-fly for the same DMA engine. To resolve this problem, we introduce a DMA engine status for each actor execution.

ALGORITHM 5: *calCodeOverlayBasicPre*(G, $PASS$, <V,R>, <V,S>)

```
Initialize code_overlay ← 0
for r ∈ R do
    /* s_last is the last segment that is loaded to region r following PASS */
    Initialize mem_state[r] ← s_last
end for
for i ∈ [0, |PASS| − 1] do
    s_cur ← getSegment(<V,S>, i)
    r_cur ← getRegion(<V,R>, i)
    r_pre ← getRegion(<V,R>, (i − 1 + |PASS|)%|PASS|)
    if s_cur ≠ mem_state[r_cur] then
        if r_cur = r_pre then
            code_overlay ← code_overlay + T_c(s_cur)
        else
            code_overlay ← code_overlay + T_overlap(s_cur)
        end if
        mem_state[r_cur] ← s_cur
    end if
end for
return code_overlay
```

ALGORITHM 6: *calCodeOverlayDeepPre*(G, $PASS$, $STATUS$, <V,R>, <V,S>)

```
1: Initialize code_overlay ←0
2: for r ∈ R do
3:     /* s_last is the last segment that is loaded to region r following PASS */
4:     Initialize mem_state[r] ← s_last
5: end for
6: for j ∈ [0, |PASS| − 1] do
7:     s_cur ← getSegment(<V,S>, j)
8:     r_cur ← getRegion(<V,R>, j)
9:     if s_cur ≠ mem_state[r_cur] then
10:        i ← (j − 1 + |PASS|)%|PASS|
11:        while getRegion(<V,R>, i) ≠ r_cur do
12:            i ← (i − 1 + |PASS|)%|PASS|
13:        end while
14:        < max_start, max_period > ← findMaxPeriod(PASS, STATUS, i, j)
15:        cost ← T_c(C_s_cur)
16:        overhead ← setDMABusy(PASS, STATUS, cost, max_start, j)
17:        code_overlay ← code_overlay + overhead
18:        mem_state[r_cur] ← s_cur
19:    end if
20: end for
21: return code_overlay
```

To adopt the deep prefetching optimization into our heuristic, we update the calculation for code overlay overhead as described in Algorithm 6. The difference from the basic prefetching algorithm is that when v_{cur} is absent from the local SPM, we first identify the range that a prefetching call for s_{cur} can be inserted, Line 10–13. Then we issue prefetching s_{cur} at the start of the longest DMA engine idle period, Line 14–16. *findMaxStart* in the algorithm finds the maximum DMA engine idle period that a prefetch of s_{cur} can be issued. The inputs to *findMaxStart* are the current PASS, the DMA engine status, the DMA cost of s_{cur}, the deep prefetching start location, and the current execution location. *setDMABusy* marks the current DMA engine to be busy for

ALGORITHM 7: *RegionAssignmentAndDataOverlay*$(G, PASS, STATUS)$

1: Initialize actor to region assignments <*V,R*>, as each actor occupies a separate region.
2: Construct IF table entry for each region pair $<(r_i, r_j), Integer>$, where $r_i, r_j \in R, i < j$
3: $region_mem \leftarrow \sum_{r \in R} C_r$
4: Initialize life time of all data segments $LIFE$
5: $buf_mem \leftarrow calBuf(PASS, LIFE)$
6: $data_overhead \leftarrow 0$
7: **while** $region_mem + buf_mem > C_p$ and $!(|R| = 1$ and $buf_mem = BUF_MIN)$ **do**
8: **if** $buf_mem = BUF_MIN$ **then**
9: $do_weight \leftarrow +\infty, co_weight \leftarrow 0$
10: **else if** $|R| = 1$ **then**
11: $co_weight \leftarrow +\infty, do_weight \leftarrow 0$
12: **else**
13: $do_weight \leftarrow \Delta_{t_do}/\Delta_{m_do}, co_weight \leftarrow \Delta_{t_co}/\Delta_{m_co}$
14: **end if**
15: **if** $do_weight < co_weight$ **then**
16: $< buf_mem, overhead > \leftarrow DataOverlay(PASS,\ STATUS, LIFE)$
17: $data_overhead \leftarrow data_overhead + overhead$
18: **else**
19: Collapse region pair $<r_i, r_j>$ with minimum IF. Update <*V,R*>, and IF table.
20: $region_mem \leftarrow \sum_{r \in R} C_r$
21: **end if**
22: **end while**
23: **return** <<*V,R*>, $data_overhead$>

the entire period of transferring s_{cur}. Subsequent inquiry of a DMA engine with a busy status will be denied in order to respect to resource constraint.

6.3. Data Overlay Optimization

To reduce the data buffer usage of a given schedule, we further introduce data overlay as discussed in Section 3.3.3. Now, we have two ways of reducing memory, namely, by collapsing two regions or by data overlay. We update Algorithm 3-*RegionAssignment* in our SDF scheduling approach with Algorithm 7 to either perform data overlay or collapsing two regions when the data buffer usage and the total region size exceed the local SPM. The first three lines in Algorithm 7 initialize <*V,R*>, IF, and $region_mem$, which are identical to Algorithm 3. The lifetime of each data segment is initialized for implementing data overlay. A data segment is defined as the total number of tokens produced on each of an actor's outgoing edge. For example, given the SDF described in Figure 9, there are two data segments being produced at actor *A*' execution, one consists of 4 tokens (on edge $A \rightarrow D$) and one consists of 1 token (on edge $A \rightarrow B$). The lifetime of a data segment starts from the actor execution where it is produced and ends when all its tokens are consumed. $LIFE[i][j][k] = 1$ denotes that the data segment produced by the ith actor execution in the given *PASS* on the actor's jth outgoing edge is alive at time interval k. The current data buffer usage buf_mem is calculated by $calBuf(PASS, LIFE)$. Function $calBuf(PASS, LIFE)$ iterates through a given *PASS* and calculates the buffer usage of each time interval k by summing up all data tokens that are alive at k. The maximum data buffer usage among all time intervals is returned as the data buffer usage of the given *PASS*.

Line 6 initializes data overlay overhead to be 0. Starting from Line 7, we enter a while loop where we keep on collapsing two regions or applying data overlay until the total region size and data buffer fits into the local SPM, or both operations fail ($|R| = 1$ and $buf_mem = BUF_MIN$). BUF_MIN is the minimum buffer usage required

ALGORITHM 8: $DataOverlay(PASS, STATUS, LIFE)$

1: $old_buf \leftarrow calBuf(PASS, LIFE)$
2: $cur_buf \leftarrow old_buf, overhead \leftarrow 0$
3: **while** $old_buf \neq BUF_MIN$ and $cur_buf = old_buf$ **do**
4: Find time interval $k \in [0, |PASS| - 1]$ with the largest data buffer usage
5: $<i, j> \leftarrow findToken(PASS, STATUS, LIFE, k)$
6: $cost \leftarrow T_c(getDataSegment(PASS, i, j))$
7: $overhead \leftarrow overhead + setDMABusy(PASS, STATUS, cost, i, k)$
8: $overhead \leftarrow overhead + setDMABusy(PASS, STATUS, cost, k, j)$
9: Update $LIFE$ for $<i, j>$
10: $cur_buf \leftarrow calBuf(PASS, LIFE)$
11: **end while**
12: **return** $<cur_buf, overhead>$

even with data overlay and it is given by the maximum buffer usage of each actor, including input buffers and output buffers. In each iteration of the while loop, we calculate the data overlay weight factor *do_weight* and the code overlay weight factor *co_weight*. The weight factors indicate the code or data overlay overhead for each unit of memory requirement reduction. If *buf_mem* reaches *BUF_MIN*, we set *do_weight* to be $+\infty$ and *co_weight* to be 0, Line 9. Else if there is only one region left, we set *co_weight* to be $+\infty$ and *do_weight* to be 0, Line 11. Otherwise, we calculate the data overlay overhead Δ_{t_do} and memory saving Δ_{m_do} for applying data overlay, and code overlay overhead Δ_{t_co} and memory saving Δ_{m_co} for collapsing two regions. Δ_{t_do} is calculated by applying Algorithm 8 (discussed in the following paragraph). We can calculate Δ_{m_do} by simply taking the difference of data buffer usage before and after applying data overlay. Note that since the DMA engine *STATUS* and data segment life time *LIFE* are modified by Algorithm 8, only their copies are passed in at this step (the original copies of *STATUS* and *LIFE* are modified when we realize data overlay and code pre-fetching optimizations as discussed in Algorithm 7, Line 16 and Algorithm 9, Line 11 that is discussed later). For calculating Δ_{t_co} and Δ_{m_co}, we collapse the region pair $<r_i, r_j>$ with minimum *IF* and calculate the code overlay overhead before and after. Again, <*V,R*>, *IF*, and *STATUS* are modified by this procedure and we pass in their copies. Based on the calculated *do_weight* and *co_weight*, we either perform data overlay (Algorithm 8) and update *buf_mem*, Line 16–17, or collapse two regions and update *region_mem*, Line 19-20. Upon termination of the while loop, We return the actor to region mapping <*V,R*> and the total data overlay overhead *data_overhead*.

The algorithm for data overlay is provided in Algorithm 8. Given the current *PASS*, DMA engine status, and the lifetime of each data segment, Algorithm 8 iteratively introduces data overlay until a memory reduction is achieved or *BUF_MIN* is reached. In the algorithm, the buffer usage before data overlay is stored to *old_buf*. *cur_buf* is initialized to *old_buf* and *overhead* is initialized to 0. At each iteration of the while loop, we identify time interval k with the largest data buffer usage, Line 4. Then for all data segments that are alive at k, we find the data segment $<i, j>$ (the data segment that is produced on the jth outgoing edge of the ith actor execution in the given *PASS*) that if overlayed will result in the smallest overhead, Line 5. The overhead of overlaying a data segment $<i, j>$ at time interval k is given by

$$overhead = max(0, cost - backward_period) + max(0, cost - forward_period) \quad \text{where } cost = T_c(getDataSegment(PASS, i, j)). \tag{1}$$

In this equation, the *backward_period* and *forward_period* denote the consecutive DMA engine idle periods that can be used for pushing data segment backward to the

ALGORITHM 9: *MinOverlaySchedulingOptimized*(G, P)

```
1: Initialize min_overlay ← +∞
2: Initialize PASS ← MinBufferScheduling(G)
3: repeat
4:   Initialize STATUS to be idle for every time interval
5:   /* Perform actor to region assignment and data overlay */
6:   <<V,R>, do_overhead> ← RegionAssignmentAndDataOverlay(G, PASS, STATUS)
7:   if Σ_{r∈R} C_r ≤ code_mem then
8:     /* Perform actor to segment assignment */
9:     <V,S> ← Segmentation(G, PASS, STATUS, <V,R>)
10:    /* Calculate current code overlay overhead */
11:    cur_overlay ← calCodeOverlayDeepPre(G, PASS, STATUS, <V,R>, <V,S>)
12:    cur_overlay ← cur_overlay + do_overlay
13:    if cur_overlay < min_overlay then
14:      min_overlay ← cur_overlay
15:      solution ← clone(G, PASS, <V,R>, <V,S>)
16:    end if
17:  end if
18: until collapseTwoExecs(PASS) = false
19: return solution
```

off-chip memory and bringing it forward to the local SPM. *getDataSegment* returns the data segment size that is produced on the jth edge of the ith actor execution in the given PASS. $T_c(getDataSegment(PASS, i, j))$ calculates the DMA cost for transferring data segment <*i*,*j*> between the local SPM and the off-chip main memory. Then we overlay token <*i*,*j*>, Line 6–8, update its life time Line 9, and calculate the buffer usage after overlaying <*i*,*j*>, Line 10.

The overall minimum overlay scheduling algorithm after incorporating deep prefetching and data overlay is given in Algorithm 9. Compared to Algorithm 2, we added the initialization of DMA engine status in Line 4. The original *RegionAssignment* is replaced with *RegionAssignmentAndDataOverlay* and the *cur_overlay* is the sum of code overlay overhead with deep prefetching[6] and data overlay overhead, Line 12. After incorporating data overlay operation, the complexity of *RegionAssignmentAndDataOverlay* becomes $O(n^4)$. The complexity of *Segmentation* is $O(n^3)$. They are both nested inside the loop of SDF scheduling, which is $O(n)$. Therefore, the overall algorithm complexity becomes $O(n^5)$.

7. EXPERIMENTAL RESULTS

7.1. Experimental setup

We evaluated the efficiency of our heuristic by compiling ten benchmarks from the StreamIt compiler 2.1.1 [MIT b] onto one SPE of the IBM Cell BE. The source code of each benchmark is delivered with the StreamIt compiler and a brief discussion can be found in MIT [a]. Table IV gives the detailed view of each benchmark. The first column provides the benchmark names, the second and third columns provide the number of actors and edges in each benchmark. The fourth column provides the number of actor executions in a PASS. The last two columns provides the total code size of each benchmark and its minimum buffer usage. The last row of the table computes the average values for each column. We implemented our techniques as an optimization

[6]In Algorithm 9, every call to *calCodeOverlay* is replaced with *calCodeOverlayDeepPre*, including the *Segmentation* subroutine. It is why *STATUS* is also passed in as a parameter to *Segmentation* in Algorithm 9.

Table IV. Benchmark Specifications

Benchmarks	#Actors	#Edges	#Executions	Total Code Size (Bytes)	Minimum Buffer (Bytes)
Beamformer	53	88	308	12356	2272
Bitonicsort	26	48	104	576	256
Channelvocoder	53	68	142	22996	6800
DCT	36	290	156	2673	1024
DES	33	40	188	2256	1024
FFT	17	16	128	2318	2048
Filterbank	35	41	232	41879	416
Fmradio	29	39	114	34285	204
Serpentfull	26	25	164	10056	3584
Tde	25	24	238	3226	6144
Average	33	68	177	13262	2377

Fig. 10. Our heuristic approach compared with previous 3-stage ILP and minimum buffer scheduling.

pass in the StreamIt compiler that operates on the intermediate representation (IR) of a stream application. Each benchmark is cross-compiled for executing on one SPE.

7.2. Comparison with Previous 3-Stage ILP and Minimum Buffer Scheduling

Figure 10 compares the overlay overhead of our heuristic with the 3-stage ILP [Che and Chatha 2010] and a minimum buffer scheduling approach. The SPM size is set to be 8K in the experimental set up. The 8K SPM is selected such that for most of the benchmarks it is enough to hold the minimum buffer usage but still not so large that all code and data fit into it. The performance results of minimum buffer scheduling (MBS), 3-stage ILP with/without code pre-fetching (ILP basic and ILP BP), and heuristic with/without code pre-fetching (Heu. basic and Heu. BP), deep prefetching (Heu. DP), and data overlay (Heu. DP&DO) are provided in Figure 10. The x-axis gives us the benchmark names and the y-axis provides the overlay overhead for each benchmark normalized to the minimum buffer scheduling results. BP, DP, and DO are short forms for basic prefetching, deep prefetching, and data overlay, respectively.

For benchmarks *Bitonicsort*, DCT, DES and FFT, there is no overlay overhead in all techniques. The reason is that for these four benchmarks, the buffer usage and total code size all fit into the 8K SPM with a minimum buffer schedule. For the remaining six benchmarks, our heuristic approach without code pre-fetching achieves an overlay overhead reduction of 50% compared with the minimum buffer scheduling. The performance is within 5% compared with 3-stage ILP approach which takes exponential time to run. With basic prefetching our heuristic generates no overlay overhead for most of the benchmarks. This is due to the fact that with basic prefetching, most of the DMA

Fig. 11. Impact of each optimization in our heuristic approach.

transfers are hidden by actor executions. Compared with minimum buffer scheduling, the average overlay overhead reduction is around 97% with basic prefetching. For these benchmarks that still impose overlay overhead after basic prefetching, deep prefetching optimization results in an average overlay overhead reduction of 23% compared with results of basic prefetching. Further introducing of data overlay gives us a performance improvement of 19% compared with results without data overlay. The average algorithm runtime of our heuristic is 84 seconds and the average algorithm runtime of the previous 3-stage ILP is 25926 seconds. In other words, compared with the previous 3-stage ILP approach, our heuristic runs more than 300 times faster. With optimizations of deep prefetching and data overlay, we also achieved better performance than the ILP approach.

7.3. Impact of Each Optimization

We evaluate the impact of each optimization in our heuristic in this section. Since our heuristic algorithm is much faster than the previous 3-stage ILP approach, we were able to run our experiments through a series of different SPM sizes. The SPM sizes are determined in the following way. We first calculated $memMIN$ and $memMAX$ for each benchmark such that without data overlay, there is no feasible solution for any SPM smaller than $memMIN$ and there is no overlay overhead for any SPM larger than $memMAX$. We iterate through the SPM size from $memMIN$ to $memMAX$ with a step size of $(memMAX - memMIN)/STEPS$ ($STEPS$ is set to 10 in our experiments). The calculation of $memMIN$ and $memMAX$ is given by

$$memMIN = getMem(MBS) + max_{v \in V} C_v, \quad memMAX = getMem(MBS) + \sum_{v \in V} C_v. \tag{2}$$

In Eq. (2), $getMem(MBS)$ calculates the buffer usage of a minimum buffer schedule and $max_{v \in V} C_v$ calculates the largest code size among all actors. Figure 11 shows the average overlay overhead after each optimization. The overlay overhead is normalized to the result at the SDF scheduling stage. Observed from Figure 11, the region assignment delivers the most significant performance improvement at an average of 48%. This is due to the fact that without region assignment, all actors are assigned to the same region. The local SPM is not fully utilized and all actors are overlayed with each other. Prefetching gives us the next most significant performance improvement at around 28%. With prefetching, most of the DMA transfers are hidden by actor executions. In our heuristic, segmentation explores opportunities to combine actors in the same region to reduce the actual number of DMA transfers. It delivers an average performance gain of 17%. Finally, deep fetching and data overlay each give us an additional overlay overhead reduction of 1%.

Fig. 12. Memory usage comparison.

The performance improvements of deep prefetching and data overlay are not as significant as other optimizations because of the following reasons. Code overlay overhead has been almost optimized away with basic prefetching. Also when the SPM size is very small compared with the total code and data size, we only have a limited number of regions and the opportunities for implementing deep prefetching is highly restricted. For example if there are only two regions, prefetching and deep prefetching in fact behave identical to each other. Data overlay optimization has a potential to reduce the data buffer usage of a schedule. However, it occupies DMA engine for transferring data, thus could potentially impact the code prefetching optimization. In our heuristic, data overlay optimization is more significant in terms of improving the feasibility. In Figure 12, we provide the minimum memory requirement for a program to be executable on an SPM. The x-axis provides us with the benchmark names and the y-axis shows the memory required for a feasible solution to exist. We experimented under three configurations, heuristic without code overlay, heuristic with code overlay, and heuristic with code and data overlay. The average memory required for the twelve benchmarks without code and data overlay is 15640 bytes. With code overlay, the average memory required drops down to 3588 bytes. Adding data overlay further reduces the average memory requirement to 2774 bytes, a more than 20% improvement compared with solutions without data overlay.

7.4. Impact of SPM size

In this section, we show the code overlay overhead of each benchmark under various SPM sizes. Ten different SPM configurations were taken for each benchmark as discussed in Section 7.3, Eq. (2). Figures 13 and 14 present the performance results from the first five and second five benchmarks, respectively. The x-axis in each figure provides the SPM steps and the y-axis provides the normalized overlay overhead of our heuristic with deep pre-fetching and data overlay. The performance results at each SPM step is normalized to the overlay overhead at step 0. As we iterate from $memMIN$ to $memMAX$, the overlay overhead went down dramatically for all the benchmarks we experimented with. There are three benchmarks, *BeamFormer*, *ChannelVocoder*, and *SerpentFull* that impose no overlay overhead even before the SPM size reaches $memMAX$. This is due to data overlay optimization. Note that for benchmark *DES* step 1, 2, 4, 9 and benchmark *Tde* step 5, although the SPM size was increased, our heuristic generates almost the same overlay overhead as the previous step. This is due to the fact that we terminate the process of region assignment and data overlay in our heuristic as soon as a feasible solution is found. However, there could be scenarios where by further collapsing two regions and thus resulting in more actors in certain regions, segmentation could benefit so much that a better solution is generated. Our heuristic

Fig. 13. SPM size variation (1st set).

Fig. 14. SPM size variation (2nd set).

can be improved at this point at the cost of increasing the algorithm complexity from $O(n^5)$ to $O(n^8)$. For the interest of algorithm run time, we left segmentation out of region assignment and data overlay optimization in our SDF scheduling heuristic.

7.5. Code Overlay Evolution

In this section, we demonstrate the evolution of code overlay overhead with SDF scheduling. The SPM size is again set to be 8K and we plot the code overlay overhead as the PASS gradually evolves from a minimum buffer schedule to minimum actor switch schedule. Figure 15 shows the performance results normalized to the overlay overhead from the initial stage where a minimum buffer schedule was adopted. For benchmarks *BitonicSort*, *DCT*, *DES*, and *FFT*, there is no overlay overhead and our heuristic terminates immediately. Figure 15 plots the overlay overhead evolution for the remaining six benchmarks. We calculate the code overlay overhead *cur_overlay* for each schedule being generated. If it is smaller than the recorded code overlay overhead, then we update *min_overlay* with *cur_overlay*. *min_overlay* maintains the best solution obtained so far. As observed from Figure 15, the code overlay overhead gradually reduces as the schedule evolves. It reaches a steady-point after a certain number of steps where a best solution is recorded. Benchmark *SerpentFull* achieved a very low overlay overhead with its minimum buffer schedule. Therefore, its overlay overhead was not updated until an improved schedule is achieved much later.

Fig. 15. Overlay cost evolution with SDF scheduling.

Fig. 16. Impact of scaling DMA cost.

7.6. Impact of Scaling DMA Cost

In this section, we examine the impact of scaling DMA transfer cost to simulate the scenarios where there are multiple stream applications running on SPM based multi-core architecture. With a fixed on-chip bandwidth multi-core architecture, as we add more and more cores the DMA transfer cost becomes larger and larger. In this experiment, we scaled the DMA cost by 2, 4, 8, and 16. The SPM is set to be 8K. In Figure 16, the x-axis shows the six out of ten benchmarks that generate overlay overhead. The y-axis shows the overlay overhead normalized to overlay overhead without scaling. For benchmarks *BeamFormer*, *FMRadio*, and *SerpentFull*, there is no overlay overhead at scale factor of 1 with deep prefetching and data overlay implemented. Therefore their overlay overhead is normalized to the performance at scale factor 2. As observed from Figure 16, the overlay overhead increases much faster than the DMA overhead. This because both deep prefetching and data overlay optimizations rely on using actor executions to overlap with DMA transfers. When the DMA cost scales up, not only the costs for transferring code and data scales up, it also greatly impacts the deep prefetching and data overlay optimization.

7.7. Impact of Scaling Code Size and Run Time

In this section, we simulate scheduling larger applications by scaling the code size and runtime of each actor in the original benchmark. The SPM size is set to be 8K. In Figure 17, the x-axis provides the benchmarks and the y-axis provides the normalized overlay overhead with deep prefetching and data overlay. The overlay overhead is normalized to the overlay overhead without scaling. When there is no feasible solution,

Fig. 17. Impact of scaling code size and run time.

the overlay overhead is infinite and we only show it up until 32. We examine scale factors of 2, 4, 8, and 16. Observed from Figure 17, several benchmarks become infeasible after a few scaling steps, for example *ChannelVoCoder* and *SerpentFull* at scaling factor 2, *FilterBank* and *FMRadio* at scaling factor 4, and *Tde* at scaling factor 16. The behavior is resulting from the fact that for these benchmarks, the data buffer usage is close to the SPM size and their actor code sizes are comparably large. After scaling, the largest actor code size cannot fit into the available code memory. For *BeamFormer*, *BitonicSort*, *DCT*, *DES*, and *Tde* (the first several scaling steps), the overlay overhead increase is less than the scaling factor. This is because when the runtime of an actor is scaled, we have a better chance to utilize actor executions to overlap with code and data overlay. The fact that the DMA base cost does not change after the code size is scaled also contributes to this behavior.

8. CONCLUSION

We presented a SDF scheduling heuristic for scheduling SDF specifications on SPM based architectures with the objective of overlay overhead minimization. We also presented extensions to our base heuristic with basic prefetching, deep prefetching, and data overlay optimizations. The comparison with minimum buffer scheduling and the ILP approach demonstrates that our heuristic is able to efficiently explore various design tradeoffs and generate high-quality solutions. We evaluated the efficiency of our heuristic approach with different SPM configurations, schedules and optimizations. The final results show that our heuristic approach is efficient and generates high-quality solutions in all cases.

REFERENCES

Angiolini, F., Benini, L., and Caprara, A. 2003. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. ACM, New York, 318–326.

Angiolini, F., Menichelli, F., Ferrero, A., Benini, L., and Olivieri, M. 2004. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. ACM, New York, 259–267.

Avissar, O., Barua, R., and Stewart, D. 2002. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst. 1*, 6–26.

Baker, M. A., Panda, A., Ghadge, N., Kadne, A., and Chatha, K. S. 2010. A performance model and code overlay generator for scratchpad enhanced embedded processors. In *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES'10)*. ACM, New York, 287–296.

Bandyopadhyay, S. 2006. Automated memory allocation of actor code and data buffer in heterochronous dataflow models to scratchpad memory. Tech. rep. No. UCB/EECS-2006-105.

BANDYOPADHYAY, S., FENG, T. H., PATEL, H. D., AND LEE, E. A. 2008. A scratchpad memory allocation scheme for dataflow models. Tech. Rep., University of Berkeley, Berkeley, CA.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUS: Stream computing on graphics hardware. In *Proceedings of the ACM SIGGRAPH Papers (SIGGRAPH'04)*. ACM, New York, 777–786.

CHE, W. AND CHATHA, K. 2010. Scheduling of synchronous data flow models on scratchpad memory based embedded processors. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 205–212.

EGGER, B., KIM, C., JANG, C., NAM, Y., LEE, J., AND MIN, S. L. 2006. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. ACM, New York, 223–233.

FLACHS, B., ASANO, S., DHONG, S., HOTSTEE, P., GERVAIS, G., AND KIM, R. E. A. 2005. A streaming processing unit for a cell processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'05)*. Digest of Technical Papers, Vol. 1, 134–135.

JANAPSATYA, A., IGNJATOVIĆ, A., AND PARAMESWARAN, S. 2006. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'06)*. IEEE Press, 612–617.

JANTSCH, A. 2003. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation; Electronic Version*. Morgan Kaufmann Series in Systems on Silicon, Elsevier.

JUNG, S. C., SHRIVASTAVA, A., AND BAI, K. 2010. Dynamic code mapping for limited local memory systems. In *Proceedings of the 21st IEEE International Conference on Application-Specific Systems Architectures and Processors (ASAP)*. 13–20.

KISTLER, M., PERRONE, M., AND PETRINI, F. 2006. Cell multiprocessor communication network: Built for speed. *Micro, IEEE 26*, 3, 10–23.

MIT. a. Streamit benchmarks. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.

MIT. b. Streamit compiler source code. http://groups.csail.mit.edu/cag/streamit/restricted/files.shtml.

NGUYEN, N., DOMINGUEZ, A., AND BARUA, R. 2005. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*. ACM, New York, 115–125.

OWENS, J. 2007. GPU architecture overview. In *ACM SIGGRAPH Courses (SIGGRAPH'07)*. ACM, New York.

PABALKAR, A., SHRIVASTAVA, A., KANNAN, A., AND LEE, J. 2008. SDRM: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Proceedings of the 15th International Conference on High Performance Computing (HiPC'08)*. Springer-Verlag, Berlin, Heidelberg, 569–582.

PHAM, D., AIPPERSPACH, T., BOERSTLER, D., BOLLIGER, M., CHAUDHRY, R., COX, D., HARVEY, P., HARVEY, P., HOFSTEE, H., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., PHAM, M., PILLE, J., POSLUSZNY, S., RILEY, M., STASIAK, D., SUZUOKI, M., TAKAHASHI, O., WARNOCK, J., WEITZEL, S., WENDEL, D., AND YAZAWA, K. 2006. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE J. Sol.-State Circ. 41*, 1, 179–196.

STEINKE, S., WEHMEYER, L., LEE, B.-S., AND MARWEDEL, P. 2002. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2002*. 409–415.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. Streamit: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction*. Grenoble, France.

TRUONG, L. 2009. Low power consumption and a competitive price tag make the six-core tms320c6472 ideal for high-performance applications. White Paper, Texas Instruments.

VERMA, M. AND MARWEDEL, P. 2006. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. VLSI Syst. 14,* 8, 802–815.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04)*. ACM, New York, 104–109.

WILLINK, E. D., EKER, J., AND JANNECK, J. W. 2002. Programming specification in CAL. In *Proceedings of OOPSLA Workshop Generative Technology Context Model-Driven Architecture*.

Received September 2011; revised March 2012, September 2012, December 2012; accepted March 2013