Dynamic Scheduling of Stream Programs on Embedded Multi-core Processors

by

Haeseung Lee

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

Approved August 2013 by the
Graduate Supervisory Committee:

Karamvir Chatha, Chair
Sarma Vrudhula
Chaitali Chakrabarti
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

December 2013

ABSTRACT

Stream computing has emerged as an important model of computation for embedded system applications particularly in the multimedia and network processing domains. In recent past several programming languages and embedded multi-core processors have been proposed for streaming applications. This thesis examines the execution and dynamic scheduling of stream programs on embedded multi-core processors. The thesis addresses the problem in the context of a multi-tasking environment with a time varying allocation of processing elements for a particular streaming application. As a solution the thesis proposes a two step approach where the stream program is compiled to gather key application information, and to generate re-targetable code. A light weight dynamic scheduler incorporates the second stage of the approach. The dynamic scheduler utilizes the static information and available resources to assign or partition the application across the multi-core architecture. The objective of the dynamic scheduler is to maximize the throughput of the application, and it is sensitive to the resource (processing elements, scratch-pad memory, DMA bandwidth) constraints imposed by the target architecture. We evaluate the proposed approach by compiling and scheduling benchmark stream programs on a representative embedded multi-core processor. We present experimental results that evaluate the quality of the solutions generated by the proposed approach by comparisons with existing techniques.

ACKNOWLEDGEMENTS

First and foremost, I would like to give my special thanks to my graduate advisor Dr. Karam Chatha for the patient guidance, encouragement, and advice he has provided throughout my time as his student. His deep knowledge of wide ranging subjects has the capability to provide solutions to almost every problem that I encountered in my research. Without his guidance and persistent help, I will not be able to achieve that I have achieved today. I am really honored to work with Dr. Chatha during two years of my Master's study.

I also would like to thank my committee members for agreeing to be on my thesis committee despite their extremely busy schedule. I would like to thank to Dr. Vrudhula for his dedication to the Embedded Systems Consortium that benefits all the members including me. I would like to thank to Dr. Chakrabati for her professional knowledge and guidance. I also would like to thank to Dr. Wu for continuous support.

All the lab members who have spent their time with me, discuss research problems with me, giving insight to me, and written papers with me, I would like to thank them too. I would like to thank Weijia Che for his professional knowledge of stream programs that enlightened me so many times. I also would like to thank to Amrit Panda for his understanding of architectures and discussions that we had. I would like to thank to Jyothi Swaroop for the project that we did together and Anil Chunduru for the support and discussions that we had.

I would like to thank to Arizona State University for giving this great opportunity to conduct research and to meet so many good people. I would also like to thank my family, father, mother and my brother. My family's support keeps me staying on the right track even I am far from home.

Last but not the least, I would like to express my gratitude to Jiyoun Kim for her continued support and encouragement. She was always there cheering me up and stood by me through the good times and bad. I cannot put it into words to show my appreciation and love.

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

2 PREVIOUS WORK 4

3 PRELIMINARIES 6

3.1 Stream program 6

3.2 Embedded multi-core architecture 6

3.3 Problem description 6

4 DYNAMIC SCHEDULING OF STREAM APPLICATIONS 8

4.1 Compile time analysis and re-targetable code generation 8

4.1.1 Structure of the application 8

4.1.2 Periodic admissible sequential schedule 9

4.1.3 Retargetable code generation 9

4.1.4 SDF annotation and actor classification 10

4.2 Dynamic scheduler 10

4.2.1 Calculation of optimal workload per core 10

4.2.2 Calculation of workload and memory requirements of a batch 11

4.2.3 Actor to batch assignment or fusion 12

4.2.4 Batch replication and core assignment 13

4.2.5 DMA overhead amortization 13

4.2.6 Algorithm and complexity analysis 14

4.3 Implementation details 16

4.3.1 SPM layout and allocation 17

4.3.2 Software pipelining 17

4.3.3 Run-time control thread 19

5 EXPERIMENTAL RESULTS 21

5.1 Comparisons with existing approaches 21

CHAPTER Page
5.1.1 Comparison with Flexstream [8] 21
5.1.2 Comparison with Baker et al. [1] 23
5.1.3 Performance comparison with compile time approach 24
5.1.4 Impact of DMA amortization optimization and comparisons with optimal . 25
5.1.5 Performance scaling with available cores and memory 25
6 CONCLUSION WITH FUTURE WORK 28
REFERENCES 29

LIST OF TABLES

Table Page

5.1 StreamIt Benchmarks 22

LIST OF FIGURES

Figure		Page
1.1	Problem addressed by the thesis	2
4.1	Dynamic Scheduler Framework	9
4.2	DMA overhead amortization	14
4.3	Memory layout for a local SPM.	16
4.4	Software pipelined execution.	18
5.1	Comparison with Flextream	22
5.2	Comparison with Baker et al.	23
5.3	Comparison with compile time approach	24
5.4	Performance without DMA amortization	26
5.5	Performance with DMA amortization	26
5.6	DCT performance scaling	27
5.7	FFT performance scaling	27

Chapter 1

INTRODUCTION

Stream computing has emerged as an important programming model for high performance compute intensive embedded system workloads. Examples of such workloads include network processing, multimedia and signal processing and computer vision. Stream programs are typically specified as a set of actors or filters communicating with each other through FIFOs. In each execution an actor consumes a fixed amount of tokens or data items on its input FIFO(s) and produces a fixed amount of tokens on its output FIFO(s). Thus, a stream program specification exposes the task and data level parallelism of the application. A number of stream programming languages have also emerged in recent past for specifying such applications. CUDA [15], Peakstream [18], Rapidmind [14], TStreams [11], OpenCL [19], StreamIt [20], and CAL [6] are examples of commercial and academic stream programming languages. Typically, stream applications exhibit stable actor execution sequence and memory access patterns. Consequently, stream applications can be aggressively analyzed statically to generate highly optimized implementations.

Processor architectures specifically aimed at stream programming workloads have also emerged concomitantly with stream computing and programming languages. Such processor architectures are also called as stream processors and they incorporate specialized architectural features. As stream computing workloads are characterized by high computation requirements, stream processors are implemented by multi-core architectures. Specifically, stream processors are implemented as heterogeneous multi-core architecture where control plane processor(s) implements the more mundane workloads (such as OS, JVM and so on) and a multi-core data plane sub-system implements the streaming workloads. The data plane sub-system typically consists of an array of homogeneous processor cores that may incorporate specialized instructions based on the target domain. More importantly as stream programs permit extensive static analysis, the data plane processors incorporate scratch pad memories (or SPM) instead of caches. SPM requires the programmer or compiler to manage the code execution and scheduling along with data communication. Although, there is a greater burden on the static (manual or automated) design phase with SPM, the generated designs exhibit higher performance and lower power consumption in comparisons to cache based architectures. Finally, the data plane processors do not host an OS. Rather, the OS or scheduler

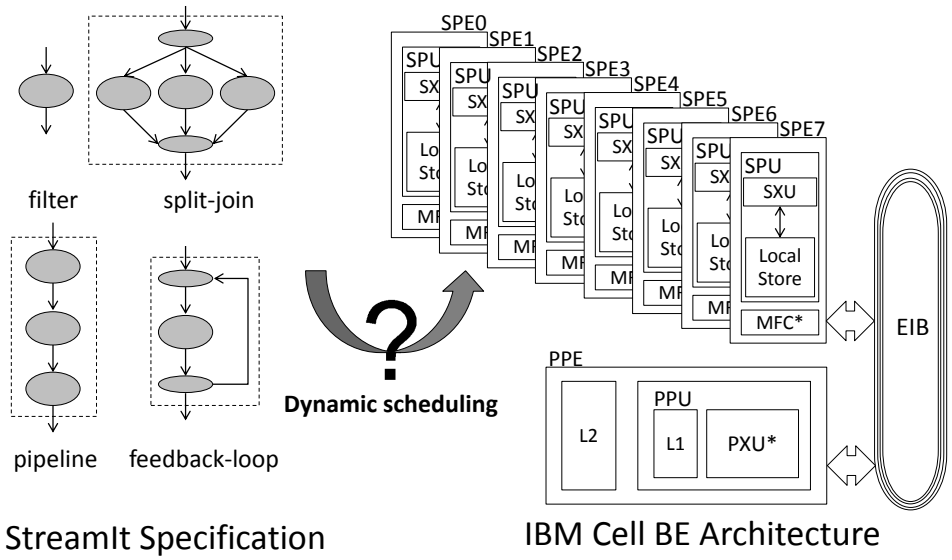


Figure 1.1: Problem addressed by the thesis

executing on the control plane processor is responsible for overall orchestration of application execution on the data plane. Examples of commercial stream processors include IBM Cell Broadband Engine (BE) [9], Nvidia GeForce series [10], Ageia's PhysX [22], TI TMS320C6472 [21] and many DSPs.

A key challenge in implementing stream programs on associated stream processor architectures is the partitioning or parallelization of the application across the multi-core data plane. As the data plane processors incorporate an SPM the programmer or compiler is burdened with the task of also partitioning the limited SPM for code and data, and initiating transfers of code from external DRAM memory, and data to/from external DRAM and SPMs of other cores. The programmer or compiler must balance computation with communication overheads to maximize the throughput of the application. To the best of our knowledge there are not commercial tool chains that can effectively perform all the design trade-offs for implementing an application on stream processor architectures. However, several compiler techniques and optimizations have been proposed in recent past in academia.

A key drawback of the static compilation approaches is that it produces a single executable code implementation. For example if the target processor architecture has 8 cores, the compiler will optimize the application to utilize all the 8 cores (or fewer if specified). However, the number

of cores available to execute an application vary in a typical multitasking embedded system. For example it is quite possible that 4 cores are initially available to run an application, after some time 8 cores are available and next 2 cores are available. In other words the number of cores available to execute a stream application may vary over time. Existing compiler approaches do not permit dynamic variation in the available number of cores for a streaming application. *This thesis addresses the problem of dynamic scheduling of streaming applications on embedded multi-core architectures (see Figure 1.1).* Specifically, we present a two stage approach consisting of retargetable code generation at compile time, and its dynamic scheduling at run time. The proposed dynamic scheduling approach is able to assign actors to processing cores at runtime while effectively balancing the limited SPM memory and communication overheads. We present experimental results that compare the proposed approach against other existing dynamic scheduling techniques, and static compilation approaches.

The remainder of the thesis is organized as follows. Chapter 2 compares the proposed approach against existing research, Chapter 3 discusses required background on stream program specification, embedded processor architectures, and formally defines the problem, Chapter 4 describes the proposed approach, Chapter 5 discusses the experimental results, and finally Chapter 6 concludes the thesis.

Chapter 2

PREVIOUS WORK

Pino et al. [17] addressed the problem of scheduling SDF specifications on multi-core processors. Chen et al. [5] and Ostler et al. [16] proposed approaches for mapping networking applications on multi-core network processors. Liao et al. [13] proposed techniques for parallelizing Brook language specification on general purpose processors. Gordon et al. [7] developed a heuristic approach to generate multi-threaded code for RAW processor architecture. Kudlur et al. [12] proposed an integer linear programming (ILP) based approach for mapping StreamIt applications on IBM Cell BE. Che et al. [4] proposed an ILP and heuristic technique for mapping StreamIt specifications on Cell BE that accounted for the on-chip memory constraints. They improved upon their previous work by addressing cyclical dependencies in the StreamIt specification [3]. All of these approaches generate one parallelized implementation for the input specification based on the available cores in the target architecture. If the available number of cores that are available on the target architecture are reduced the implementation cannot be utilized. Thus, the compiled executables are not able to address dynamic variations in available resources.

Wiggers et al. [23] made a case for runtime scheduling and noticed that for dataflow based applications runtime scheduling can be managed at high level of abstraction. Zhang et al. [24] proposed a dynamic scheduling scheme for IBM Cell BE that utilizes a operating system like ready queue based approach which ignores the overall structure of the application. Blagojevic et al. [2] presented a multigrain parallel scheduling (MGPS) approach for IBM Cell BE that uses message passing interface (MPI) protocol and aims to combine task and loop parallelization schemes. The approach relies upon a fixed task to SPE assignment and therefore cannot effectively adopt to variations in number of available cores.

Flexstream proposed by Hormati et al. [8] is one of the two existing approaches whose problem focus closely matches ours. Flexstream utilizes a compile time ILP optimization to generate a maximal parallel implementation of the StreamIt specification on the IBM Cell like architecture. The parallel implementation is then adopted to address variations in available number of cores at run time. Baker et al. [1] also proposed an approach to dynamic scheduling of StreamIt

specification on IBM Cell BE. They utilize compile time heuristic approach to generate a periodic admissible sequential schedule (or PASS) of the StreamIt specification that is dynamically parallelized at run time. Our static compile time approach is similar to Baker et al. However, we also describe schemes for generation of re-targetable code and its implementation at run-time. Our dynamic scheduler is distinguished from both Flexstream and Baker et al. by its structured unrolling approach to achieve highly optimized implementations through fusion and fission. Further, in comparison to Baker et al. the computational complexity of our algorithm is linear for a fixed number of cores. The experimental results demonstrate the superior performance of the proposed approach against both Flexstream and Baker et al., and establish the contribution of the thesis.

Chapter 3

PRELIMINARIES

3.1 Stream program

We consider stream programs that are specified using StreamIt [20]. In StreamIt a filter, also called an actor in the thesis, is the smallest unit of computation (see Figure 1.1). An actor can be composed by container classes into split-join construct to specify task level parallelism, pipeline construct to specify data level parallelism, and feedback construct. A filter communicates with other filters through FIFOs. A filter in each execution or firing consumes a fixed number of tokens or data items from its input FIFOs and produces a fixed number of tokens on its output FIFOs. An actor can fire as long as there are enough tokens on its input FIFOs and the output FIFOs have sufficient space to hold the data tokens. As such the StreamIt execution semantics closely follow synchronous dataflow (SDF) model of computation, and the intermediate representation (IR) used by our approach is based on SDF.

3.2 Embedded multi-core architecture

We consider the IBM Cell BE [9] as a representative stream processor architecture (see Figure 1.1). The Cell BE is a 32-bit architecture composed of a Power processor element (PPE) that acts as the control plane processor, and eight synergistic processing elements (SPEs) that form the data plane subsystem. Each SPE has 256 KB SPM that must be managed by the programmer for hosting both code and data. Each SPE is also equipped with a DMA engine for communicating both code and data to/from DRAM and other SPMs. The 9 processing cores are connected with each other and the DRAM controller through the element interconnect bus (EIB). The DMA overhead for sending data over the EIB has been characterized as $(0.21 + 0.075 \times d)\mu s$ where d is the size of data in KB that is being communicated [3].

3.3 Problem description

Given:

- An SDF based description of the StreamIt specification as $G(V,E)$ where V is the set of actors and E is the set of FIFOs. For each $u \in V$, $t(v)$ describes the run time of the actor on a single SPE and $s(v)$ describes the code size in KB. Similarly for each $e(u,v) \in E$, $p(e)$ denotes the

number of tokens produced by u, $c(e)$ gives the number of tokens consumed by v, $s(e)$ gives the size of each token in KB of each token produced by u, and $d(e)$ gives the initial number of tokens on an edge (relevant in the case of cycles).

- An embedded multi-core processor where P denotes the set of cores currently available for the application in the data plane, M denotes the size of SPM associated with each processor, and DMA overhead for communicating d KB is given by $f(d) = (0.21 + 0.075 \times d)\mu s$.

Generate a run-time schedule that maps the actors to cores in the data plane subject to the memory constraints and communication overheads with an objective of maximizing the throughput of the application.

Chapter 4

DYNAMIC SCHEDULING OF STREAM APPLICATIONS

Our approach to dynamic scheduling of stream applications on embedded multi-core architectures consists of two stages (see Figure 4.1). In the first stage we utilize a compiler to analyze the stream application and generate re-targetable code for the actors along with some other information. During run time a dynamic scheduling framework executes the application on the available cores. The dynamic scheduling framework has the capability to execute the same application on variable number of cores. In the following few sections we discuss the compilation stage, dynamic scheduling algorithm and implementation details. In our approach we have considered StreamIt as the input specification language and IBM Cell BE as the target multi-core architecture.

4.1 Compile time analysis and re-targetable code generation

The key objective of the compilation stage is to statically analyze the application, generate key information for the dynamic scheduling algorithm, generate a periodic sequential schedule for the application, and create re-targetable executables for the actors within the application.

4.1.1 Structure of the application

The compiler analyzes the StreamIt specification to extract the SDF based intermediate representation of the application. The SDF representation captures the graphical structure of the application along with the number and sizes of data tokens produced and consumed on each edge. We make two transformations on the SDF based representation driven by our focus on dynamic scheduling. We first transform the SDF into a single appearance schedule. Thus, each actor in the transformed schedule has only one firing for one complete execution of the application. Further, we collapse any cycles in to a single actor. Both these optimizations reduce the size of the SDF and complexity of the dynamic scheduling problem. We would like to emphasize that as we are considering SDFs that represent entire applications, cycles are not very common. We call the transformed structure as SDF' and the corresponding graph is denoted by $G'(V',E')$. In the single appearance SDF' each edge $e \in E$ has the same number of tokens that are produced and consumed, and we denote it as $pc(e)$.

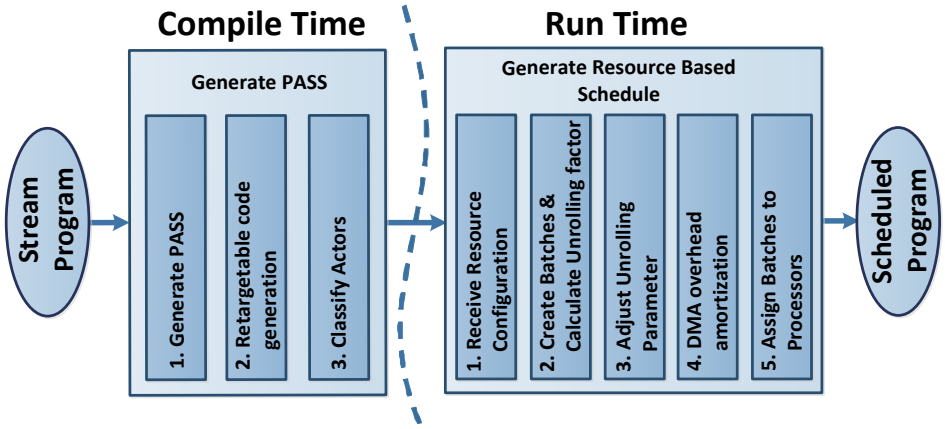


Figure 4.1: Dynamic Scheduler Framework

4.1.2 Periodic admissible sequential schedule

The compiler conducts the rank test on the application and generates the periodic admissible sequential schedule or PASS for the application if a schedule exists. The PASS is generated by a depth first traversal of the SDF' subject to precedence constraints between parent and child actors. The algorithm requires that a special source actor exist in the application that generates the input data stream. For example in the StreamIt benchmarks the file reader actor acts as the source actor for the application. As the SDF' is a single appearance graph, the generated PASS fires all instances of the same actor in the original SDF as a consecutive sequence. Further, all actors belonging to a cycle are also fired consecutively.

4.1.3 Retargetable code generation

The compiler generates retargetable binaries for all the actors. We are interested in run time scaling up or down of the number of cores available to the application. As we do not know the number of cores available to the application, we initially assume that all the actors are assigned to the same core and executed according to the PASS. In the case that the application does not contain cycles or stateful (described below) actors, replication of the same PASS across the available number of cores is indeed the throughput optimal solution (subject to available memory).

4.1.4 SDF annotation and actor classification

After the binaries have been compiled the static analysis phase generates additional information for the dynamic scheduler. It records the code memory requirements for each actor, workload for one firing of the actor in SDF', total workload for the PASS which is represented by W_{pass}, and classification of the actors in SDF' as stateless or stateful. An actor is classified as stateful and is represented by u_{stf} if the output in each firing is a function of the current and past inputs (stored as state). Alternatively, an actor is stateless and is denoted by u_{stl} if the outputs are purely a function of the current inputs. An actor in SDF' that represents a cycle in SDF is labeled as stateful.

4.2 Dynamic scheduler

The objective of the dynamic scheduler is to generate a highly optimized software pipelined schedule for execution of the streaming application. The dynamic scheduling algorithm utilizes software pipelining and unrolling to maximize the throughput of the application. Software pipelining is achieved by introducing double buffering for inter-core communication. Unrolling is achieved by utilizing a fusion-fission approach. Fusion assigns multiple actors to a single batch. Fission assigns each batch to one or more cores. When a batch is assigned to more than one core it effectively unrolls the SDF'. As the schedule is generated dynamically we are interested in a near linear time algorithm. In the following paragraphs we describe the calculation of optimal and batch workloads, actor to batch assignment, fission or unrolling of a batch, and block processing to amortize DMA overhead. Finally, we discuss the pseudo code for the algorithm and its computational complexity.

4.2.1 Calculation of optimal workload per core

The optimal workload for each core W_{opt} for a given application is a function of the number of cores, and maximum workload of a stateful actor, and is given by:

$$W_{opt} = \max\{\frac{W_{pass}}{n}, \max_{\forall u_{stf} \in V'}(W(u_{stf}))\}$$

As multiple instances of a stateful actor in an unrolled SDF' must be executed sequentially, the effective throughput of the stateful cannot be increased by its replication across multiple cores. Alternatively, the effective throughput of a stateless actor can be increased in an unrolled SDF by replicating it on multiple cores. Thus, the optimal workload of the application is also lower bounded by the maximum workload of any stateful actor.

4.2.2 Calculation of workload and memory requirements of a batch

As an actor u is assigned to a batch b, the workload of the batch $W(b)$ is summation of its current workload plus the actor workload. Thus,

$$W(b) = W(b) + W(u)$$

The memory requirement of a batch $M(b)$ is summation of the memory required by code for actors assigned to the batch, intra-batch (same as intra-core) communication buffers, and inter-batch (identical to inter-core) communication buffers. Thus,

$$M(b) = M_{code}(b) + M_{intra}(b) + M_{inter}(b)$$

As an actor is assigned to a batch, the code memory requirement of the batch increases by the size of binary executable file for the actor. The increase in buffer memory requirement is dependent upon if the parent or child of the actor have been assigned to any batch or not (currently unassigned). Further, if the parent or child has already been assigned to the same batch as the actor or a different one, the memory requirements are different. Thus, we have the following three cases:

- **Case I :** The parent u of the actor v has not been assigned to any batch. In this case we initially assume that u will be assigned to a different batch, and the inter-batch communication buffer requirement is increased to accommodate for double buffering.

  $$M_{inter}(b) = M_{inter}(b) + 2 \times pc(e) \times s(e), e(u,v) \in E'$$

  Similarly, for a child w of v that has not been assigned to any batch we initially assume double buffering. Thus, the equation is identical except the edge $e(v,w)$ is considered.

- **Case II :** The parent u of the actor v has been assigned to the same batch as v. In this case we remove the double buffers that we had initially assigned for inter-batch communication, and increase the buffer for intra-batch communication. Thus,

  $$M_{inter}(b) = M_{inter}(b) - 2 \times pc(e) \times s(e), e(u,v) \in E'$$

  $$M_{intra}(b) = M_{intra}(b) + pc(e) \times s(e), e(u,v) \in E'$$

The calculation for a child actor w which has also been assigned to the same batch as v is also similar.

- **Case III :** The parent u of the actor v has been assigned to a different batch b'. In this case we only need to assign space for double buffering in batch b to which v is being assigned. Thus,

$$M_{inter}(b) = M_{inter}(b) + 2 \times pc(e) \times s(e), e(u,v) \in E'$$

The buffer space calculation for a child w of actor v is similar.

4.2.3 Actor to batch assignment or fusion

The dynamic scheduling algorithm first assigns stateful actors to batches followed by stateless actors. The strategy for actor to batch assignment is different for each of these two types of actors.

1. *Stateful actors :* We assign stateful actors to a batch in the order that they appear within the PASS. We assign multiple stateful actors to a batch as long as the workload of the batch is less than W_{opt} and memory requirements of the batch are lower than the SPM (M) size. When addition of an actor to a batch results in the violation of either of these two constraints, we assign the actor to a new batch. We refer to a generic stateful actor batch as b_i^{stf}.

2. *Stateless actors :* The stateless actors are also assigned to batches in the order that they appear within the PASS. However, in the assignment of the stateless actors we only consider the memory constraints. That is we only ensure that the memory requirements of the batch are lower than the SPM size. We ignore the workload of the batch as we can maximize the workload of a stateless actor batch by fission (or unrolling). We refer to a generic stateless actor batch as b_i^{stl}.

It is possible that at this stage after assigning actors to batches the total number of batches are greater than the number of cores. Thus, we cannot do one to one mapping between the batches and cores. The primary reason for mapping stateful and stateless actors to different batches is to be able to improve the throughput of stateless batches by batch replication. If on the other hand the number of batches are already greater than the number of cores, we cannot do replication. In such a case we attempt to generate a new solution where we mix both stateful and stateless actors in

batches. The actors are assigned in the order that they appear within the PASS. We again ensure that W_{opt} and SPM memory constraints are not violated. If the solution still has more batches than the number of cores, then we make a final attempt by re-distributing the actors in the smaller workload batches across batches with larger workloads subject to memory constraint. If a valid solution is still not found we declare failure.

4.2.4 Batch replication and core assignment

After the actors have been assigned to batches we improve the throughput of the application by unrolling or fission provided the number of batches are less than the number of cores. During fission we only consider batches that contain stateless actors. Each batch of stateful actors are assigned a unique core. Let P' denote the number of cores that is remaining after the stateful batches have been assigned. We first calculate the optimal fission for each batch of stateless actors b_i:

$$U^*(b_i^{stl}) = \frac{W(b_i^{stl})}{\max_{\forall j}(W(b_j^{stf}))}$$

Notice that $U^*(b_i^{stl})$ is not necessarily an integer. We then estimate the total core requirement after fission as $K = \sum_{\forall i} U^*(b_i^{stl})$. The fission of a stateless batch is then set as:

$$U(b_i^{stl}) = \begin{cases} K > P' & \left\lfloor U^*(b_i^{stl}) \times \frac{P'}{K} \right\rfloor \\ K \leq P' & \lfloor U^*(b_i^{stl}) \rfloor \end{cases}$$

If at the end of batch replication there are some free cores (number of batches less than the number of cores) and the optimal workload of the application is constrained by stateless batch, then we attempt to improve the throughput by generating an alternative solution. In the new solution we mix both stateful and stateless actors together, and assign actors to batches in the order that they appear within the PASS. We compare the new alternative with the previous solution, and select the one with better performance.

4.2.5 DMA overhead amortization

In our actor to batch (and eventually core) mapping we do incorporate double buffers for amortizing DMA overheads. However, DMA overheads are typically governed by a minimum base cost ($0.21\mu s$ on Cell BE). The base cost is constant below a certain threshold, D_{th} of data transfer (D_{th} is 2KB in Cell BE) and then increases linearly with data size X ($7.5 \times 10^{-2} \times X\mu s$ in Cell BE).

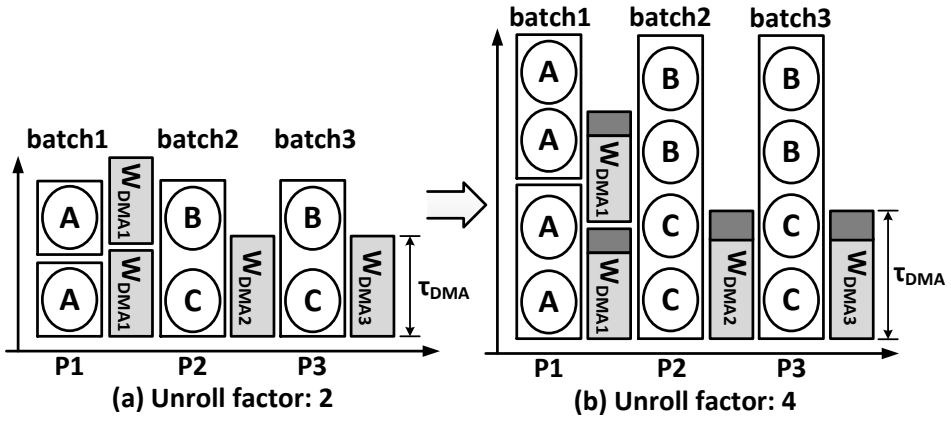


Figure 4.2: DMA overhead amortization

Thus, if a DMA operation involves fewer than D_{th} KB of data, its cost can be amortized by block processing. Block processing involves delaying the transfer of data until more data is produced by computation and then initiating bulk transfers. Effectively, block processing involves further unrolling of the application and requires more buffer memory for inter-core communication. Figure 4.2 shows an example of block processing. The left hand side of Figure 4.2 depicts the original design produced by the fission stage that has an unrolling factor of 2. However, the DMA transfer size for batch 1 is lower than the threshold, and its overhead is greater than associated computation. We unroll the implementation thereby increasing the unrolling factor to 4 as shown in the right hand side of Figure 4.2. As the rate of increase of DMA overhead beyond the threshold is quite low, we are able to completely mask the communication overhead by computation. Our algorithm applies block processing only if the DMA overhead for some batch is greater than its associated computation, and if the block processing indeed would result in an improvement.

4.2.6 Algorithm and complexity analysis

The pseudo-code for the dynamic scheduling algorithm is given in Algorithm 1. Line 3 calculates the W_{opt}, Line 4 assigns the stateful actors to batches, and Line 6 assigns stateless filters to batches. Line 8 performs batch replication or fission if the number of batches in current solution is less than number of cores ($F = false$). Line 9 checks for the case if the number of batches is less than the number of cores, and if the optimal workload is constrained by stateless batches. Function $W_{max}()$ returns the maximum workload over a set of batches. Line 10 sets R as true if the condition is

```
   Input  : List of stateful actors L_stf in PASS order
          : List of stateless actors L_stl in PASS order
          : List of actors denoting the pass PASS
          : P, M, G'(V',E')
   Output: List of batches B (|B| = |P|), or failure
 1 B ← ∅ ;
 2 F ← false, R ← false ;
 3 W_opt ← CalcWopt(L_stf, L_stl, P);
 4 AssignBatches(L_stf, B, W_opt, M);
 5 if |B| > |P| then F ← true;
 6 if not F then AssignBatches(L_stl, B, ∞, M) ;
 7 if |B| > |P| then F ← true;
 8 if not F then ReplicateBatches(B);
 9 if |B| < |P| and W_max(B_stf) < W_max(B_stl) then
10     R ← true;
11 if F or R then
12     B' ← ∅ ;
13     AssignBatches(PASS, B', W_opt, M);
14     if |B'| > |P| then ReAssignBatches(B');
15     if F and |B'| ≤ |P| then
16         F ← false;
17         B ← B';
18     if R and |B'| ≤ |P| and W_max(B') < W_max(B) then
19         B ← B';
20 if not F then DmaAmortization(B, f);
21 if F = true then return failure else return B;
```

Algorithm 1: DynamicScheduler()

```
   Input                : List of actors A,W_opt, M
   Input and Output: List of batches B
 1 b ← newbatch() ;
 2 while A ≠ ∅ do
 3     a ← A.getactor();
 4     if (W(b) + W(a) ¿ W_opt) OR
 5     (M(b) + M(a,b) ¿ M) then
 6         B.addbatch(b) ;
 7         b ← newbatch() ;
 8     b.addactor(a);
 9 if b ≠ ∅ then B.addbatch(b);
```

Algorithm 2: AssignBatches()

satisfied. The *if* block of Line 11 generates a solution by assigning actors in the PASS order to batches. The algorithm enters the block if no valid solution has been generated upto this stage ($F = true$) or R is true. Line 14 attempts to re-distribute the smaller workload batches across the larger workload batches (subject to memory constraint) if the generated solution is not valid. The function *ReAssignBatches()* performs this operation and modifies B'. If the new solution is valid,

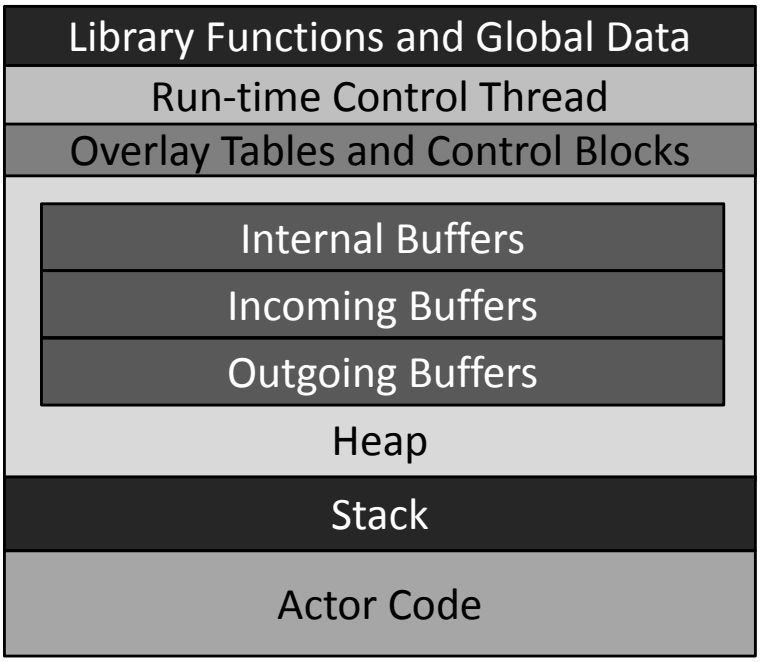


Figure 4.3: Memory layout for a local SPM.

Line 15 updates F and B accordingly. Line 18 checks if the new solution is better than the previous solution and updates B. Line 20 performs DMA overhead amortization, and finally Line 21 returns the final solution as B (if it is found).

The functions *AssignBatches()* and *ReplicateBatches()* have a complexity of $O(|V|)$. $W_{max}()$ has a complexity of $O(|P|)$. As the function *ReAssignBatches()* sorts the existing batches it has a complexity of $|P|log|P|$. Consequently the overall complexity of the algorithm is given by $O(|V|+|P|+|P|log|P|)$. In other words for a given architecture (P is constant) the complexity of the algorithm is linear in the number of actors in the application ($O(|V|)$.

4.3 Implementation details

In this section we discuss the implementation details for the dynamic scheduler in the context of an embedded multi-core architecture. In the following we have assumed the target architecture as IBM Cell BE. However, the discussion holds true for other multi-core architectures that utilize an SPM. In the following sections we discuss the memory map, software pipeline implementation, and run-time control thread.

4.3.1 SPM layout and allocation

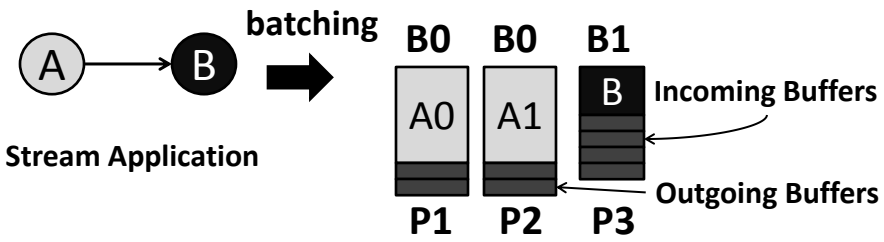
Figure 4.3 describes the memory layout of an SPM attached to a data plane core. The memory is partitioned into contiguous regions for library functions and global data, code region for our runtime control thread, overlay tables and control blocks, heap, and stack. Given a fixed size SPM, we first preserve the memory for library function and global data. Then, the memory for storing the code of our run time control thread is allocated. Immediately after the static code region, we allocate a fixed size memory for storing overlay tables and control blocks[1]. We maintain two overlay tables, one for storing information of the code source, destination addresses, and size of each actor (_ovly_table). The other table maintains information about whether an actor is present in the on-chip SPM (_ovly_buf_table). The control blocks also have two implementations, namely batch control block (bcb) and actor control block (acb). A batch control block contains information of the entire batch, and an actor control block contains information that is specific to each actor. The remaining memory is designated for heap and stack. The actor code and their data buffers of each SPM are dynamically allocated in heap because they are unknown at compile time. The buffers include space for intra-core communication, inter-core incoming data, and inter-core outgoing data. Section 4.2.2 included the discussion for calculation for data buffer sizes to be allocated.

4.3.2 Software pipelining

As discussed earlier the dynamic scheduler generates a software pipelined schedule for application execution. Figure 4.4 provides a simple example with 2 actors and 3 processing engines. We will first discuss this example as a case study and then explain the implementation details.

In Figure 4.4 (A) we have a stream application with one producer A and one consumer B. Assume that the workload of A is twice of B, and B is a stateful actor. After mapping them to 3 processors we will have actor A mapped to batch B0 then replicated once, and actor B mapped to batch B1. After batch to processing engine mapping, we have the first copy of actor A (A0) mapped to processing engine P1, the second copy of actor A (A1) mapped to processing engine P2, and actor B mapped to processing engine P3. Each copy of A and B has two buffers for every communication edge to implement double buffering. At iteration 0, both A0 and A1 start execution. *Exec A0,0*

[1] We assume an upper bound on the memory requirement for the tables.



(A)

	Processor 1		Processor 2		Processor 3
iter0	Exec A0,0		Exec A1,0		
iter1					
iter2	Exec A0,1	DMA {A0,0}	Exec A1,1	DMA {A1,0}	
iter3					
iter4	Exec A0,2	DMA {A0,1}	Exec A1,2	DMA {A1,1}	Exec B,0 {A0,0}
iter5					Exec B,1 {A1,0}
iter6		DMA {A0,2}		DMA {A1,2}	Exec B,2 {A0,1}
Iter7					Exec B,3{A1,1}
iter8					Exec B,4{A0,2}
iter9					Exec B,5 {A1,2}

(B)

Figure 4.4: Software pipelined execution.

in Figure 4.4 (B) denotes the first execution of A0 and $DMA\{A0,0\}$ denotes that the DMA engine transfers the output data of *Exec A0,0* to its destination. Starting from iteration 2, the output data of A0 and A1 are available. We start $DMA\{A0,0\}$ at iteration 2 and in iteration 4 we start the first execution of B that consumes this data, denoted by *Exec B,0*$\{A0,0\}$. We also start $DMA\{A1,0\}$ in iteration 2 that transfers data to the other incoming buffer of B. Consequently, we have parallelized executions of $(A0,1)$, $(A1,1)$, $(DMA\{A0,0\})$, and $(DMA\{A1,0\})$ in this iteration. Then in iterations 4 and iteration 5, we have A0, A1 each being executed once, their data being transferred to B, and Actor B executed twice. This is also the steady state execution of the entire application. We set synchronization barriers across the steady state schedule.

```
1  while true do
2      msg = wait_mailbox();
3      if msg.stop_execution==true then
4          break;
5      if msg.new_schedule_start==true then
6          Free previous memory and allocate new memory;
7          DMA control blocks and overlay tables;
8      if msg.flush_pipeline_start==true then
9          DMA control blocks;
10     Initiate DMAs for input data for index+1;
11     for i = 0; i < bcb.actors; i++ do
12         if bcb.idx ≥ acb[i].start && bcb.idx ≤ acb[i].end then
13             Execute actor[i];
14     Check and ensure DMAs complete;
15     write_mailbox(msg);
16 write_mailbox(msg);
```

Algorithm 3: Run-time Control Thread.

In our implementation, the control block of each batch holds the information of the current execution index and the number of actors mapped to the batch. The control block of each actor contains information of the iteration at which the current actor starts execution and DMA operations, the address location of actor code, pointers for its input and output data buffers in local SPM, and the source and destination of its incoming and outgoing data. All control blocks are initialized in the control plane processor by the dynamic scheduler, and then transferred to each on-chip SPM through DMAs. A main routine that operates on these data and executes a batch is discussed in the following section.

4.3.3 Run-time control thread

Algorithm 3 describes the main routine of our run time control thread that executes on the data plane core (SPE in IBM Cell BE) and implements the previously discussed software pipelined execution. The main body of the control thread is a while loop. Line 2 in the algorithm implements the synchronization barrier, and waits for a message from the control plane core (PPE in IBM Cell BE). The message has four components. The *msg.stop_execution* bit determines whether we terminate the slave thread. If *msg.stop_execution* is true, then the control thread exits immediately and writes a message back to the control plane core, Line 16. Otherwise, it checks *msg.new_schedule_start* bit to see whether the current iteration is a start of a new schedule. If *msg.new_schedule_start* is true, then

the control thread frees any heap memory that has been allocated, including previous allocations for actor code and data buffers. Then it reallocates memory for actor code and data buffers of the new schedule. The size of each memory allocation can be extracted from the same mailbox message. It then initializes the control blocks and overlay tables with data that is generate by the dynamic scheduler through DMAs. The *msg.flush_pipeline_start* bit indicates whether we terminate the execution of the current application and flush the entire pipeline. If *msg.flush_pipeline_start* is true, then the control thread updates the actor control blocks. The corresponding end iterations of actor executions and DMA transfers will be updated which flush the software pipeline. Lines 10-14 implement parallel execution of a batch for the current iteration and DMA transfers for the last iteration. The thread first initiates DMA transfers for data that are produced from the previous stage. Then for each actor, if the current batch execution index is no less than its execution start index and no more than its execution end index, it executes the current actor. After execution of every actor in the batch, it ensures that the DMA transfers for data that were produced in the previous are complete. Then it writes a mailbox message back to the control plane processor for starting the next iteration.

Chapter 5

EXPERIMENTAL RESULTS

We evaluated our technique by compiling and dynamic scheduling of StreamIt benchmark application suite (shown in Table 5.1). The target architecture was considered to be similar to IBM Cell BE. We compared the performance of our approach with two existing dynamic scheduling techniques proposed by Hormati et al. (Flexstream) [8] and Baker et al.[1]. We also compared the performance of our dynamic scheduler with a static scheduling technique proposed by Che et al. [3]. We characterized the workloads and buffer requirements of the actors by compiling and executing them on IBM Cell BE. In order to evaluate the proposed and existing approaches for architectures with larger number of cores we constructed SystemC models. The run time of actors and the communication overheads in the model were based on the characterization values obtained from the IBM Cell BE. The run-time of the dynamic scheduling algorithms were obtained by executing them on the PowerPC core of the IBM Cell BE. The run-time of the static analysis stage of the various approaches was obtained on Intel Xeon Quad-core processor at 2.8 GHz.

5.1 Comparisons with existing approaches

5.1.1 Comparison with Flexstream [8]

Flexstream utilizes an integer linear programming (ILP) formulation to obtain a maximal parallel schedule during design time on the available number of cores. It then reduces the degree of parallelism in the implementation dynamically as the number of cores are reduced. In our experiment we set the maximum number of cores as 32. Thus, Flexstream generated an implementation for each application at compile time that required 32 cores. We then evaluated the performance of the Flexstream approach as the number of cores were varied from 2 to 32. Figure 5.1 plots the performance improvement in throughput achieved by utilizing our approach in comparison to Flexstream for each of the StreamIt benchmarks and for 2-32 cores. A few trends are clearly observable. The designs generated by our approach are overwhelmingly superior to Flexstream in vast majority of the cases. Further, the performance due to our approach improves as the number of cores increase. The rate of increase in performance reduces at 32 cores. This is because the Flexstream implementation at 32 cores is the statically generated design. However, the designs by our approach even at 32 cores is better than Flexstream. The primary reason for the better performance of our approach is

benchmark	actors	edges	% stateful
beamformer	56	58	82%
bitonic_sort	40	46	35%
channelvocoder	55	70	65%
dct	8	7	0%
des	53	60	38%
fft	17	16	0%
filterbank	85	99	58%
fm	43	53	65%
mpeg2_subset	23	26	30%
serpent_full	120	128	43%
tde-pp	29	28	3%
vocoder	114	147	72%

Table 5.1: StreamIt Benchmarks

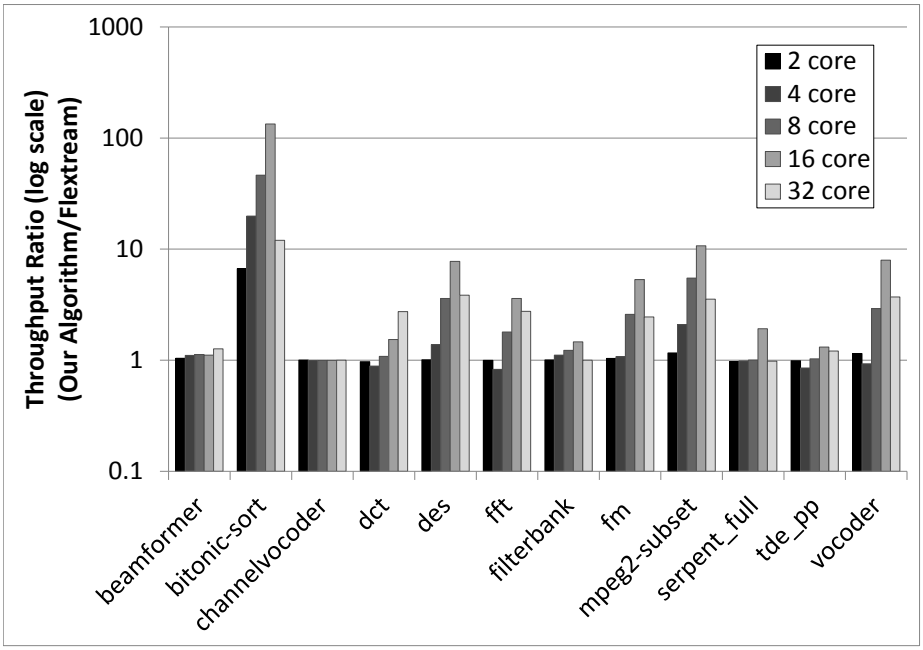


Figure 5.1: Comparison with Flextream

the aggressive unrolling that we apply both during fission (or batch replication) and DMA overhead amortization. Our approach generates designs that are on an average 5.55 times faster (standard deviation of 17.91) than Flexstream. Further, our compile time stage on an average executes in 0.03s while Flexstream ILP approach requires 49000s. Finally, our dynamic scheduling stage requires 0.3ms on average while Flexstream executes in 0.27ms.

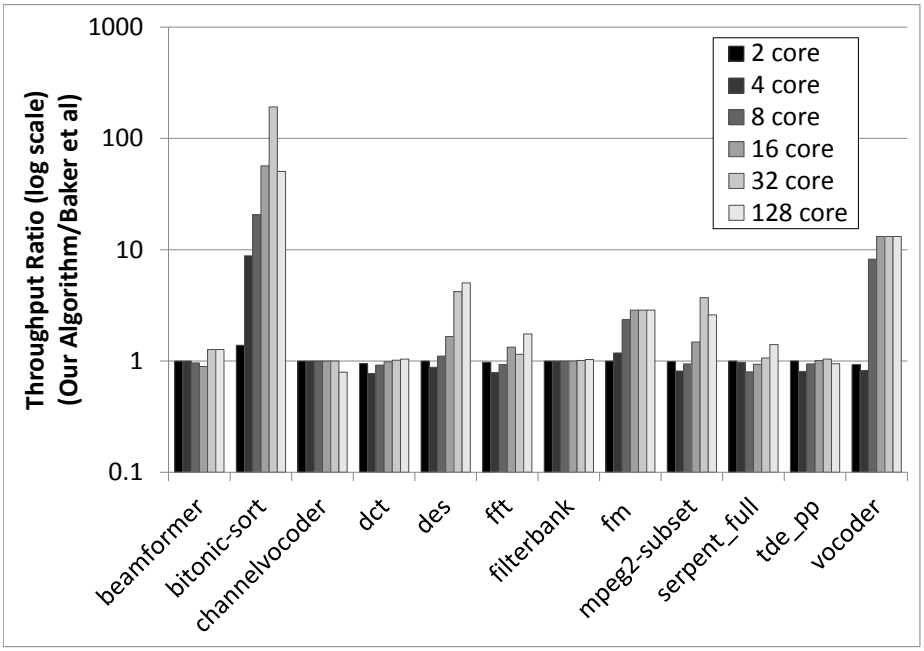


Figure 5.2: Comparison with Baker et al.

5.1.2 Comparison with Baker et al. [1]

The static analysis stage of Baker et al. is similar to our technique, and unlike Flexstream they utilize unrolling. We compared the designs generated by our approach against Baker et al. for 2 to 128 cores. Figure 5.2 plots the performance improvements achieved by our technique in comparison to Baker et al. for each of the benchmark applications. As can be observed from the figure our technique generates superior designs in comparison to Baker et al. In particular for *bitonic-sort, des, fft, fm, mpeg2-subset* and *vocoder* our approach gives large performance improvements. The primary reason for the performance improvement is that we restrict the DMA overheads by assignment of actors to batches and then replicating the batches to perform systematic unrolling which creates symmetric allocations. Baker et al. perform unrolling before assigning actors to batches (which are then one-to-one mapped to cores). Consequently, the distribution of actors across cores is quite arbitrary and results in larger DMA overheads. The average performance improvement due to our approach over Baker et al. is 6.38X with standard deviation of 23.84. Further, our dynamic scheduling algorithm executes in 0.32*ms* on average while Baker et al. requires 0.63*ms* or almost

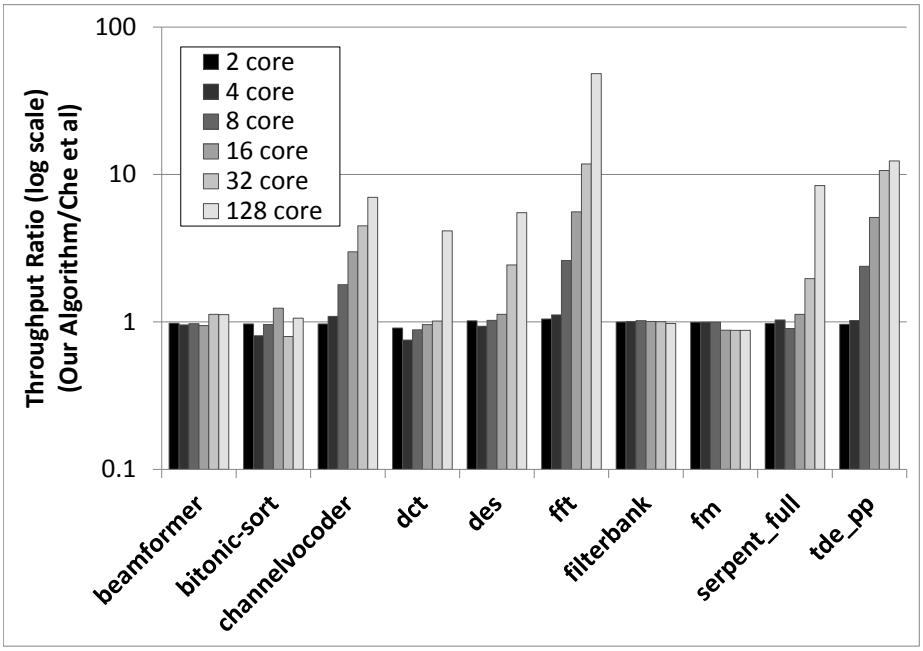


Figure 5.3: Comparison with compile time approach

twice as much as our approach. The computational complexity of our dynamic scheduling approach is linear time for a fixed number of cores and therefore we are able to generate solutions in half the time.

5.1.3 Performance comparison with compile time approach

We also compared the performance obtained by our technique against a compile time approach by Che et al. [3] that generates one solution for an application based on the number of cores specified to the compiler. Unlike, our technique the compile time approach by Che et al. utilizes retiming to address cycles in the application graph. However, in contrast to our approach they do not consider unrolling of the application. We compared the solutions generated by our approach against the static compilation technique for 2 to 128 cores. Figure 5.3 gives the performance improvement obtained by our dynamic scheduling technique against the static compilation approach. As can be seen from the figure our dynamic scheduling technique out performs the static compilation approach in many cases. The performance improvement is primarily due to the aggressive unrolling utilized by our approach. Further, even though we do not optimize for cyclic dependencies for the StreamIt benchmarks we are still able to out perform the static compilation approach that does optimize cyclic

dependencies. The average performance improvement due to our dynamic scheduling approach is 3X (standard deviation 6.48). Further, the average run time of the static compilation approach is 6*s* which is much larger than our dynamic scheduling technique that executes in *ms*.

5.1.4 Impact of DMA amortization optimization and comparisons with optimal

We also analyzed the impact of DMA amortization through block processing on the overall performance of the algorithm. Figure 5.4 depicts the normalized performance obtained by our approach without DMA amortization for 2 to 128 cores. Each performance bar in the figure is normalized to the performance of the same application on a single core. Further, for each bar we also show the maximal theoretical speed-up possible (optimal in the figure). As can be seen from the figure in the absence of DMA amortization the achieved performance is not near the theoretical optimal. This is particularly true for *bitonic-sort, des* and *fm*. The discrete DMA communication overheads limit the achievable performance. Figure 5.5 depicts the performance achieved with DMA amortization and compares it with theoretical optimal. As is clearly observable from the figure DMA amortization results in considerable speed-up in *bitonic-sort, des* and *fm*, and incremental speed-up in several others. Amortization of communication overhead is a key challenge in multi-core implementations and our technique is effectively able to do so. Further, the designs generated by our approach are quite close to the theoretical optimal solutions.

5.1.5 Performance scaling with available cores and memory

We also evaluated the performance scaling of the solutions generated by our approach for two applications without any stateful actors as the available memory and cores are scaled up. Further, we compared our designs with those generated by Baker et al. [1]. Figures 5.6 and 5.7 plot the performance obtained by our approach and by Baker et al. Each point in the two plots is normalized to the performance achieved by a single core architecture with identical memory. As can be observed from the two plots our approach generates designs that scale up linearly with the number of cores. The primary reason is that in the case of the stateless application our approach exploits the available data-level parallelism to the fullest and replicates the application across each of the cores. However, Baker et al. splits the application across the cores, and the resulting buffer requirements in the presence of memory constraints end up limiting the achievable performance.

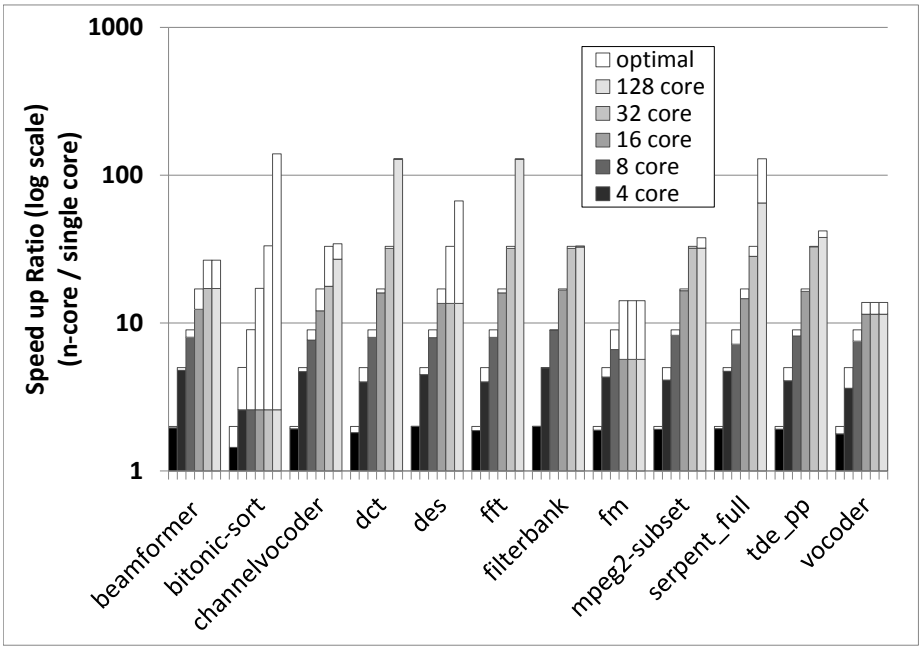


Figure 5.4: Performance without DMA amortization

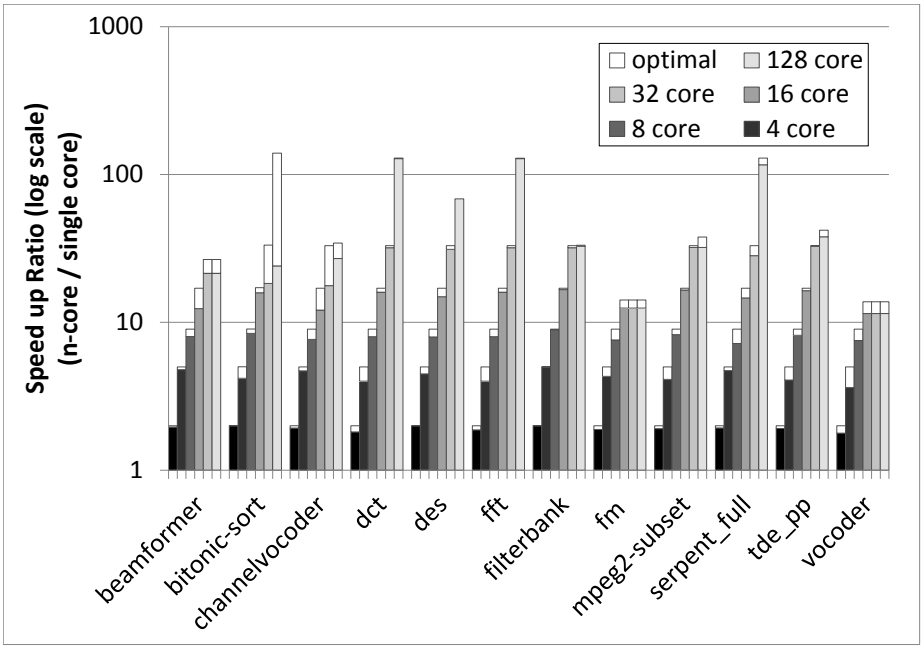


Figure 5.5: Performance with DMA amortization

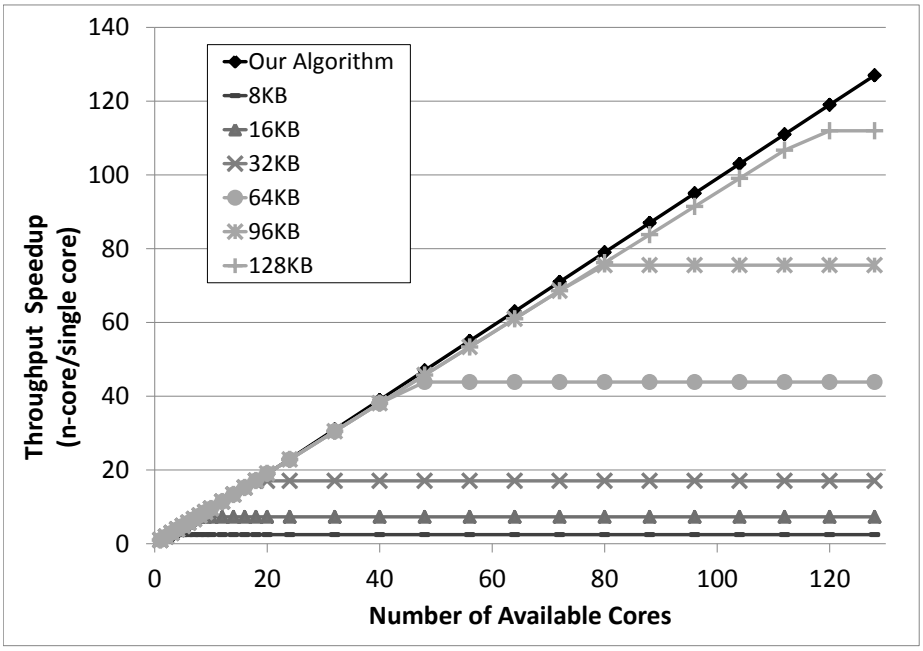


Figure 5.6: DCT performance scaling

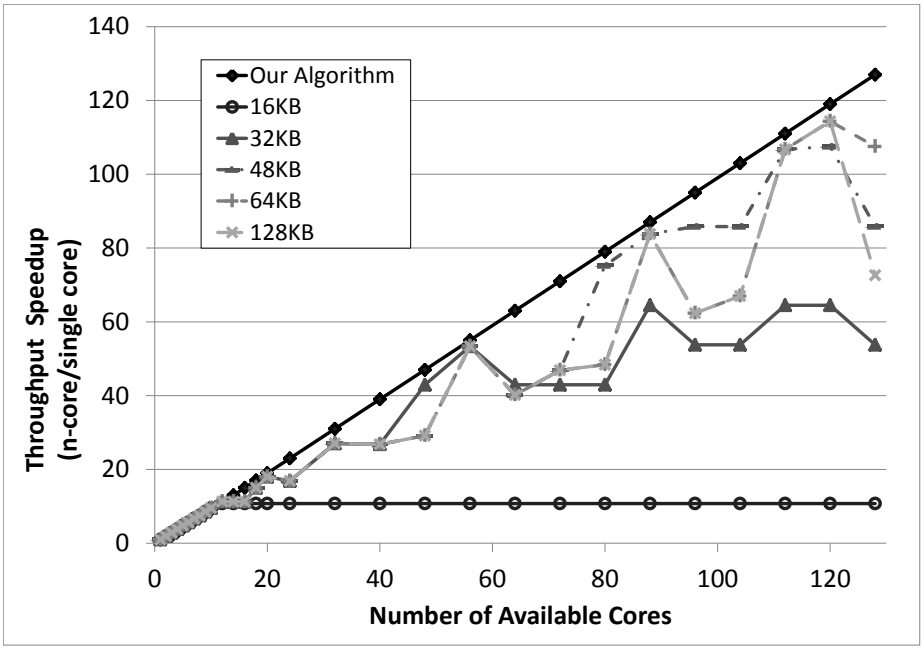


Figure 5.7: FFT performance scaling

Chapter 6

CONCLUSION WITH FUTURE WORK

The thesis addressed the problem of dynamic scheduling of streaming applications on embedded multi-core architectures. We presented a two stage static-dynamic analysis approach. The static or compile time phase collects useful information about the application and generates re-targetable executables. The dynamic scheduler reads the overall application information generated by the static compilation stage and generates highly optimized parallel implementations on the specified cores. The algorithm complexity of the dynamic scheduler is linear in the number of the cores. We evaluated our approach by comparing its solution quality and run time with two dynamic scheduling and one compile time approach. Our algorithm generates overwhelmingly superior designs in extremely low run times. The block processing based DMA amortization stage was shown to be particularly effective for reducing communication overheads. Further, the solutions generated by our approach was shown to scale up with available cores and memory.

In the proposed approach although we address SPM management through overlay tables we have not addressed problems of code and data overlays. To the best of our knowledge there are no known purely dynamic scheduling approaches that can also address code and data overlay optimizations at run time. Such optimizations have typically been addressed at compile time in the past. Future work can address incorporating low complexity code and data overlay optimizations that can be integrated with the dynamic scheduler.

REFERENCES

[1] Michael A. Baker and Karam S. Chatha. A lightweight run-time scheduler for multitasking multicore stream applications. In *Proceedings of International Conference on Computer Design (ICCD)*, 2010.

[2] Filip Blagojevic, Dimitris S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 90–100, New York, NY, USA, 2007. ACM.

[3] Weijia Che and K. Chatha. Compilation of stream programs onto scratchpad memory based embedded multicore processors through retiming. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 122 –127, june 2011.

[4] Weijia Che, Amrit Panda, and Karam S. Chatha. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1118–1123, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[5] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-la: achieving high performance from compiled network applications while enabling ease of programming. *SIGPLAN Not.*, 40:224–236, June 2005.

[6] J. Eker and J. W. Janneck. Cal language report. Technical report, University of California at Berkeley, Tech. Rep. UCB/ERL M03/48.

[7] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. *SIGARCH Comput. Archit. News*, 30:291–303, October 2002.

[8] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 214–223, Washington, DC, USA, 2009. IEEE Computer Society.

[9] J. A. Kahle et al. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49:589 –604, 2005.

[10] Emmett Kilgariff and Randima Fernando. The geforce 6 series gpu architecture. In *SIGGRAPH*. ACM, 2005.

[11] Kathleen Knobe and Carl D. Offner. Tstreams: A model of parallel computation (preliminary report). Technical report, HP Labs, Technical Report HPL-2004-78.

[12] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43:114–124, June 2008.

[13] Shih-wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[14] M. Monteyne and R.M. Inc. Rapidmind multi-core development platform. Technical report, RapidMind, Tech. Rep.

[15] NVIDIA. *Compute Unified Device Architecture Programming Guide*. NVIDIA: Santa Clara, CA, 2007.

[16] Chris Ostler, Karam S. Chatha, Vijay Ramamurthi, and Krishnan Srinivasan. Ilp and heuristic techniques for system-level design on network processor architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12, September 2007.

[17] J.L. Pino and E.A. Lee. Hierarchical static scheduling of dataflow graphs onto multiple processors. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 4, pages 2643 –2646 vol.4, may 1995.

[18] Jon Stokes. Peakstream unveils multicore and cpu/gpu programming solution. Last accessed April 2012.

[19] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66 –73, may-june 2010.

[20] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 49–84. Springer Berlin / Heidelberg, 2002.

[21] Loc Truong. White paper: Low power consumption and a competitive price tag make the six-core tms320c6472 ideal for high performance applications. *Processing Business*, oct. 2009.

[22] Scott Wasson. Ageia's physx physics processing unit. *The tech report, PC hardware explored*, Last accessed April 2012.

[23] Maarten H. Wiggers, Marco J.G. Bekooij, and Gerard J.M. Smit. Monotonicity and run-time scheduling. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 177–186, New York, NY, USA, 2009. ACM.

[24] David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, May 2008.