Advances in Streaming Software Synthesis for Manycore Processor Platforms

By

Mohammad H. Foroozannejad

B.S. (Shahid Beheshti University) 2001

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL AND COMPUTER ENGINEERING

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Soheil Ghiasi, Chair

Bevan Baas

Venkatesh Akella

Committee in Charge

2015

Abstract

Continuation of Moore's law and advances in fabrication technologies have enabled development of digital computing platforms that contain many processor cores. Such chip multiprocessor (CMP) architectures exhibit significant promise in energy and throughput sensitive applications, while offering the flexibility of software programmability. However, development of scalable parallel software for utilization of such CMP platforms remains a major challenge. As the trend in CMP design points to placing of more processor cores on the chip, the programming challenge is only going to be exacerbated in the future.

This PhD research takes several strides toward addressing the problem of parallel software development for a specific class of embedded applications. In particular, we focus on automated synthesis of parallel software for streaming applications that are to be executed on distributed-memory CMP platforms. Streaming applications demand processing of a seemingly endless stream of input data, as they are presented to the system. Typically, processing at any point demands access to a small window of input data, hence the output can be computed and streamed out as the input flows into the system. Such applications are abundant in the embedded systems space. Examples include various signal processing, data encoding/decoding schemes, multi-media, security and network inspection applications.

We advocate productive development of streaming applications by synthesizing software from high-level specifications, such as data flow graphs. In this context, we first study the problem of inter-actor buffer allocation during software synthesis from synchronous dataflow models. Buffer allocation strategy greatly impacts the memory footprint of the synthesized streaming software, which is critical for memory-constrained embedded CMPs. Next, we discuss the problem of mapping virtual processors onto physical processors of a GALS-based manycore platform, and present a constructive optimization approach that offers a controllable tradeoff between mapping quality and compilation runtime. Finally, we identify redundant memory access as a critical performance bottleneck in automatically generated streaming software. We develop a memory access analysis and optimization technique for streaming applications, which exploits properties of synchronous dataflow models to improve performance of the synthesized code.

Acknowledgments

I would like to express my special appreciation and thanks to my esteemed teachers and mentors especially my advisor Professor Soheil Ghiasi, who has been a tremendous mentor for me. I would like to thank him for all his encouragement and for helping me to grow as a research scientist. His advice on both research as well as on my career have been priceless. I would like to thank my committee members, Professor Bevan Baas and Professor Venkatesh Akella for their brilliant comments and suggestions. I would also like to extend a special thanks to the UC Davis faculty for teaching informative courses and the ECE department staff for their support.

A special thanks goes to my family; words cannot express how grateful I am to my mother and father for all of their support and sacrifice. Their prayers for me sustained me thus far. At the end I would like to express my appreciation to my dearest wife Sahbanoo who selflessly cherished our beloved children during my research. She always supported me in the moments when there was no one else by my side.

I am deeply indebted to Matin Hashemi, Trevor L. Hodges, Alireza Mahini, Mohammad Motamedi, and Brent Bohnenstiehl for their assistance.

Contents

CHAPTER 1

Introduction

For several years, advances in silicon technology has increased computing performance via enhancing clock frequency and employing more silicon budget. Nonetheless, increasing performance through maximizing clock speed does not seem to as efficient anymore as it appears to have been pushed to its limit [**Bor99**]. Moreover, the popularity of the hand-held devices in recent years with limited battery life has made energy efficiency an important factor in computing systems. On the other hand, it is well-known that we can achieve performance using parallelism in an energy efficient way [**CSB92**]. To remedy the power consumption limitations and still be able to achieve performance simultaneously, Chip Multiprocessor (CMP) platforms have been used for research purposes as well as the commercial marketplace [**ABC$^+$06**].

Current technology trends suggest that integrating more processor cores per chip will continue, and domain-specific **manycore** chips with 1000+ cores seem imminent [**ABD$^+$09, V$^+$07, TCM$^+$09**]. Figure 1.1 is an extended version of the data reported in [**Has11**] and shows the current trend in many-core technology. Distributed-memory CMPs are arguably most appropriate for massive parallel computing systems due to scalability of their memory architecture and their message passing-based programming model. Such platforms, e.g. the Asynchronous Array of Simple Processors (AsAP) [**TCM$^+$09**], are well-positioned to specifically benefit embedded systems with data-intensive applications.

A significant group of embedded applications are characterized by their requirement to process a steady stream of input data as they are presented to the system. Such applications, which are generally referred to as streaming applications, demand access to merely a small window of input data at each point in time. Thus, output can be produced and streamed out, possibly with a time lag, as the input flows into the system. Streaming applications emerge in different disciplines, such as encoding, decoding, transformation and inspection protocols in multi-media, signal processing, security, and networking domains.

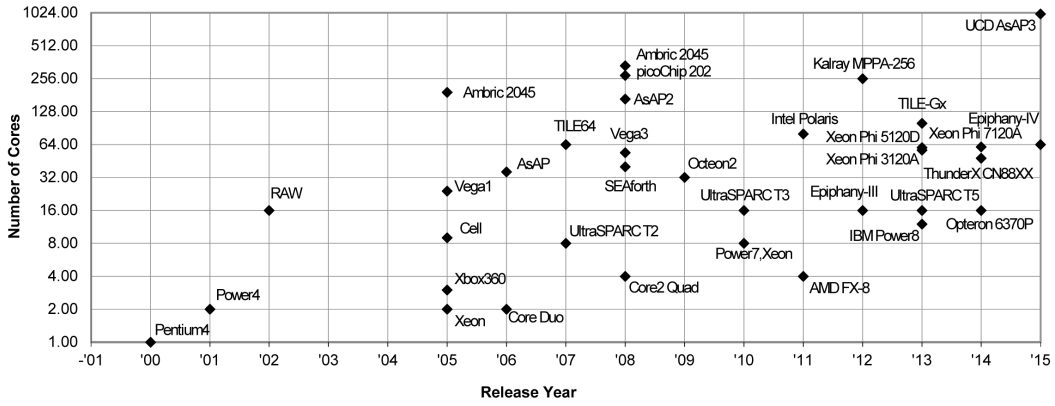


FIGURE 1.1. Current trend in many-core processors. The data shown in this Figure is an extended version of the data reported in [**Has11**]

Most streaming applications either have fixed-rate behavior, or contain fixed-rate kernels at their heart [**GB04**]. Synchronous Data Flow (SDF) graphs [**LM87b**] and their variations, such as Cyclo-Static data flow graphs [**BELP95**], are widely used to model fixed-rate applications. In these models, the processing is captured by a number of tasks that communicate solely via point-to-point channels. The channels deliver the data to the receiver in the same order that they are generated by the producer. Because of abundant parallelism and predictable data patterns, they are suitable for being efficiently executed on CMP platforms.

The intensity of the computation and communication among parallel processing units makes it very hard for the programmer to flawlessly implement the target function while paying attention to the various details of the system. On the other hand, productive parallel systems have shown sensitive to the implementation details [ABD+09]. In fact, a poorly-written parallel code may cause low performance compared to its equivalent sequential code while consuming more resources. Automated software synthesis significantly reduces the development and debugging time, providing portability of the generated function. The idea is to enable seamless and efficient transformation from a higher-order specification of the application to parallel software code for a given target parallel system, such as ones in manycore processor platforms. One of the main objectives of software synthesis is to maximize throughput, which is an important quality metric in the streaming application

domain. However, there can be other optimization objectives and/or constraints such as judicious use of memory and communication resources on chip which are imposed by resource availability of the target hardware platform.

An automated software synthesis process consists of several key algorithmic steps including task assignment, task scheduling, buffer allocation, processor mapping, and code generation. In Chapter 2.1, we will describe the flow of the software synthesis process with a brief explanation of each algorithmic step. We also present an overview of the abstraction models we use for manycore platforms and also streaming applications throughout this dissertation.

The major contribution of this dissertation is presented in Chapters 3 through 5. Chapter 3 contributes to the backend optimization step of the automatic software synthesis flow. In this Chapter, we study the inherent tradeoff between memory requirement and compilation runtime, under a given task firing schedule in the context of streaming software synthesis from data flow graphs. We utilize post-scheduling analysis granularity to control the amount of details in characterization of buffers spatio-temporal footprints. Subsequently, we transform the buffer allocation problem to two dimensional packing of polygons, where complexity of the packing problem (e.g., polygon shapes) is determined by the analysis granularity. We develop an evolutionary packing optimization algorithm, which readily yields buffer allocations. Experimental results highlight the tradeoff between complexity of the analysis and the total buffer size of generated implementations. In addition, they show dramatic improvements in total buffer size, if one is willing to pay the additional cost in optimization runtime.

In Chapter 4, we study the problem of mapping concurrent tasks of an application to cores of a chip multiprocessor that utilize circuit-switched interconnect and Global Asynchronous Local Synchronous (GALS) [**Cha84**] clocking domains. We propose a mapping algorithm called BAMSE that exploits specific characteristics of such systems. The mapping quality affects application throughput, energy consumption, and even feasibility of implementing the application. BAMSE can strike a balance between mapping quality and optimization run time, as it explores the space of mapping solutions. Thus, BAMSE is useful in both compile time and run time application mapping. The proposed technique naturally handles a number of practical requirements, such as architectural features of the

target platform, core failures, and hardware accelerators, and in addition, is scalable to a large number of tasks and cores.

Chapter 5 takes the discussion on automated software synthesis beyond code generation. The software synthesis process typically implements inter-actor communication in form of buffer arrays that are written to/read from by producers/consumers (discussed in Chapter 3). Due to practical considerations (e.g., actor IPs) and the nature of SDF models, the generated code could contain a number of redundant buffer array accesses that become evident only in the synthesized software. In this Chapter, we identify the optimization opportunity and develop an algorithm called RACE, which optimizes the synthesized code via elimination of such redundant memory operations.

In Chapter 6 a flavor of the prior studies in the field are presented and discussed. Finally, in chapter 7 an overall summary of this dissertation is provided along with potential future avenues in the field.

CHAPTER 2

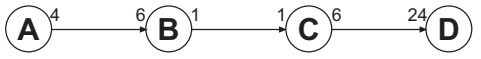
Background

Right abstractions are the key to generating quality software in the embedded domain. In this Chapter, we present an overview of the two abstract models that we use throughout this dissertation. In Section 2.1, an abstract model for manycore platforms is presented, and in Section 2.2, SDF graphs are presented as the programming model for streaming applications. Section 2.3 presents the optimization steps involved in automated software synthesis from data flow models to manycore platforms.

2.1. Manycore Platforms

Manycore processors can be represented as a graph in which, vertices model processing units (cores) and directed edges represent inter-core communication links. In this level of obstruction, the inter-core communication can be realized as FIFO channels where one core writes in and the other core reads from. The capacity of these channels represented as edge weights on the graph. Note that this is the abstract view of the hardware, and not necessarily the available physical hardware. For example, a shared-memory multi-core architecture can realize the abstract view by implementing inter-processor link in shared memory. In such architectures, unidirectional FIFO channels are implemented as arrays in the shared memory space. The access to the array has to be synchronized via a conventional locking mechanism.

Circuit-switched GALS (Globally Asynchronous Locally Synchronous) architectures are a variation of Manycore platforms. AsAP2 [**TCM+09**] is an example of such architectures. In AsAP-like circuit-switched architectures, links are statically allocated between two communicating cores at the programming phase after reset when the application is loaded to the processor. Therefor, these links cannot be shared by other inter-core connections. This is in contrast to packet-switched networks in which, the physical resources can be shared. The interconnect distance between communicating processors in AsAP-like architectures have a reverse impact on the clock frequency of the source processor [**TTB10**], i.e., in



V= { A, B, C, D} E= { A_B, B_C, C_D }
S1: 6A 4B 4C 1D Flat SAS
S2: 2(3A 2(1B 1C)) 1D SAS
S3: 2A 1B 1C 4A 3(1B 1C) 1D non-SAS
sink (A_B) = B src (A_B) = A
cns (A_B) = 6 prod (A_B) = 4
q = [6, 4, 4, 1] q[A] = 6 q[B] = 4

FIGURE 2.1. An example SDF graph, several valid schedules and some definitions are illustrated. SDF edges are annotated with corresponding production and consumption rates. V is the set of vertices and E is the set of edges of the SDF graph. $S1$, $S2$, and $S3$ are three different valid schedules for the given SDF graph. Src and $sink$ of an edge are the sending and receiving actors of the edge, and $prod$ and cns of an edge are the number of tokens produced and consumed on the edge, respectively. q is the repetition vector of the SDF containing all repetition factors of the actors in the graph. Each element of q shows how many times a specific actor should be fired in total in a valid schedule.

longer communications, the clock frequency of the source core is decreased. The drop in the source core frequency is due to the fact that the clock signal of the source core is sent along with the data to maintain communication synchrony [**TTB10**]. In Chapter 4, the Circuit-switched GALS architectures are discussed in more details.

2.2. Synchronous Data Flow Models

Synchronous Data Flow (SDF) graphs are widely used to model streaming applications. Let V_G and E_G denote the set of vertices and directed edges of the SDF graph G, respectively. Vertices of the SDF graph, also known as actors, model application tasks, and directed edges represent inter-task communication channels. Edge e starts from the actor $src(e)$ (source), and ends at the actor $snk(e)$ (sink). Figure 2.1 depicts an example.

Upon execution, each task consumes a fixed number of data items, also known as tokens, from each of its input channels. The consumed tokens are processed to generate output data, which is subsequently written to output channels of the task after completion of the execution. The generated output also has fixed rate. Equivalently, each edge e is annotated with two $prod(e)$ and $cns(e)$ numbers, which refer to the number of tokens produced by $src(e)$ and consumed by $sink(e)$ upon execution, respectively.

Application tasks can be executed only after there are enough tokens to consume on their incoming edges. The produced tokens after execution of a task might enable execution

of other tasks. Execution of a task is also referred to as firing of the corresponding actor in the model. Note that execution of a task implies that enough tokens already existed at its inputs. The streaming assumption implies that there is a sufficiently-large number of tokens at the primary input, input from outside the model, to be processed.

Task can be executed in different orders, also known as task schedules. Due to production and consumption rates, task execution changes the storage requirement of the inter-actor channels. If repetitive execution of a fixed task schedule maintains the channels storage requirement bounded, the schedule can be utilized to synthesize an implementation at compile time. Such a schedule identifies one period of execution of the application, which is iteratively invoked to process the input stream.

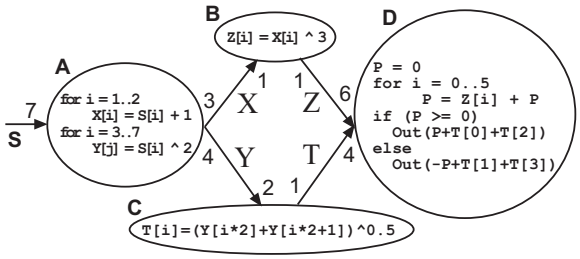
It follows that a periodic task execution schedule has to meet two conditions: 1) actors can be fired only after there are enough tokens to consume on their incoming edges, 2) all of the generated tokens have to be consumed by the end of the period, to enable infinite repetition of the schedule using finite channel storage. It is well-known that realistic application SDF can be scheduled statically [**LM87a**].

Let vector q denote the number of repetition of actors in the periodic schedule. Without loss of generality, we assume q refers to the simplest such vectors, i.e., not all of its elements can be divided by an integer larger than 1. To guarantee that all produced tokens are consumed by the end of the period, any static schedule has to guarantee the following for all edges of the SDF:

$$q[src(e)] \times prod(e) = q[sink(e)] \times cns(e)$$

The vector q is unique for real-life streaming applications [**LM87a**]. Thus, the number of firings of actors in any static schedule is constant, although their ordering might differ in the period. In particular, "Single Appearance" (SA) schedule refers to the ordering, in which each actor appears exactly once. Figure 2.1 depicts an example SDF graph, along with several example schedules and notations.

To synthesize software from a given SDF model, one needs to determine a periodic ordering for execution of the tasks, which can be infinitely repeated. In the baseline synthesis scheme, task v appears in a loop whose iteration count is $q_[v]$. Subsequently, the loops are



Schedule: 2A 6B 4C D

```
while(1)
   for i = 0..1
     for j = 0..2
       X[i*3+j] = S[j] + 1
     for j = 3..6
       Y[i*4+j] = S[j] ^ 2
   for i = 0..5
     Z[i] = X[i] ^ 3
   for i = 0..3
     T[i]=(Y[i*2]+Y[i*2+1])^0.5

     P = 0
     for i = 0..5
       P = Z[i] + P
     if (P >= 0)
       Out(P+T[0]+T[2])
     else
       Out(-P+T[1]+T[3])
end While
```

FIGURE 2.2. An example SDF, and the corresponding baseline implementation. Channels are implemented as distinct buffers. V is the set of vertices and E is the set of edges of the task graph. $S1$, $S2$, and $S3$ are three different valid schedules for the given SDF graph. *Src* and *sink* of an edge are the sending and receiving nodes of the edge, and *prod* and *cns* of an edge are the number of tokens produced and consumed on the edge, respectively. q is the repetition vector of the SDF containing all repetition factors of the nodes of the graph. Each element of q shows how many times a specific actor should be fired in total in a valid schedule.

"stitched" together in the given order, with appropriate fixtures to implement inter-task communication. Figure 2.2 illustrates the synthesized code for the depicted SDF.

SA task scheduling enables the synthesizer to save in application code size by instantiating tasks' internal computations exactly once, possibly within nested loops. The code size overhead of looping constructs is negligible with respect to typical size of task internal computations. Therefore, SA schedules are widely used in embedded systems, since they lead to small size synthesized software. In this work, we assume the given schedule to be SA, unless otherwise noted.

2.3. Software Synthesis

The software synthesis process involves several algorithmic steps, such as task assignment [**HG10b, SK03**], task scheduling [**BLM96, MBL97**], buffer allocation [**MB04,**

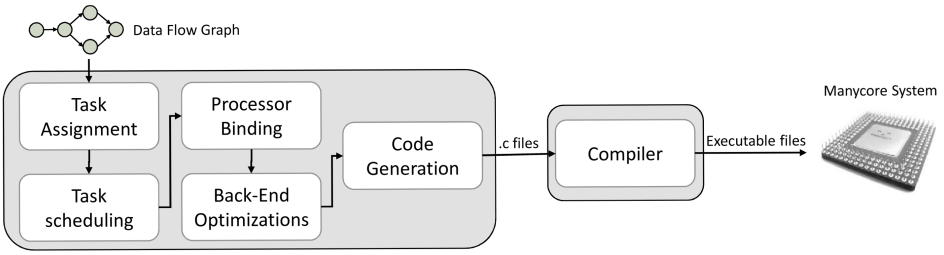


FIGURE 2.3. The flow of automatic software synthesis for SDF modeled streaming applications based on given application graph and target many-core model.

FHHG12], processor mapping [**FHM$^+$14, Tos11**], and finally code generation [**HRR91**]. The combined effect of these algorithmic steps undertakes to produce high quality software from the SDF modeled application for the target hardware platform. Figure 2.3 demonstrates these steps in the order that they are performed in the synthesis process.

In the task assignment step, every task is assigned to a processor for execution. The typical objective of task assignment is to have a balanced workload among all processors and reduce inter processor communication at the same time. At this point, the processing units are considered to be virtual processors since the physical location of the processors on the chip is ignored [**SK03**].

After the task assignment step, each group of tasks that are assigned to a virtual processor will be executed on a uniprocessor. The order in which the tasks are executed sequentially on each core is calculated in the task scheduling step. Latency, code size, and data memory footprint are common objectives in this step. To this end, different types of algorithms have been introduced each of which has a different effect in terms of the given quality measures. Single Appearance (SA) scheduling is one of the scheduling methods which is commonly used in embedded platforms. In SA scheduling, each task appears only once in the sequential code, and hence yields the minimum code size.

Processor binding (processor mapping) allocates each virtual processor to a physical core on the chip. In this step, maximum and total communication distances between connecting cores are considered in addition to other network criteria such as dead-lock and congestion.

In the backend optimization step, post-scheduling and in-core resource allocation optimizations (e.g. buffer allocation) are performed [**FHHG12**].

Finally, in the code generation step of the synthesis process, the sequential code for each core is generated (usually in a high level programming language) [**HRR91**]. At this point, the modeled application has been completely converted to the sequential codes assigned to each core processor of the final hardware platform. The generated code can be passed to a standard compiler to generate executable binaries.

CHAPTER 3

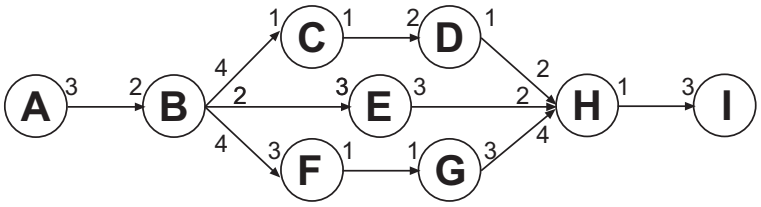
Post-Scheduling Buffer Allocation

In this Chapter, we study the inherent tradeoff between memory requirement and compilation runtime, under a given task firing schedule in the context of streaming software synthesis from data flow graphs. Due to the streaming nature of the applications, it is natural to realize the first-in-first-out (FIFO) channels as buffer arrays that are allocated as contiguous regions in the memory. In many streaming applications, buffers account for a substantial portion of the memory footprint of the synthesized programs [**MB01, MB04**]. The memory footprint of the synthesized application is especially critical when resource-constrained embedded platforms are targeted [**HG10b**].

In this context, we study the problem of post-scheduling buffer management[1] during synthesis of embedded software from SDF models[2]. We show that the resolution in analysis of buffers' spatio-temporal behavior can serve as a control knob by which synthesizer runtime (complexity) can be traded off with total buffer size (quality). On one end of the spectrum, the least amount of analysis resolution approximates buffers' spatio-temporal behavior with conventional live ranges, while on the highest analysis resolution end, perturbations to buffers' characteristics after firing of every actor are taken into consideration. There are also other alternatives available between the two extreme cases.

We transform the buffer allocation problem into packing of complex polygons in the two dimensional time-space plane. The complexity of the polygons depends on the resolution level (granularity) of the analysis. We develop an evolutionary algorithm for the packing problem, which readily allocates buffers in the memory during compilation. The technique is implemented within the MIT StreamIt compiler [**GTK+02**], which compiles a specific variation of SDF. Experimental results on a number of streaming applications illustrate the tradeoff between optimization complexity and buffer size. Additionally, the empirical evaluation illustrates the superiority of our approach over existing competitors.

[1] We use the term "buffer management" to refer to cost-benefit tradeoff in buffer allocation.
[2] Please refer to Section 3.1 for an illustrative example and problem statement.



V= { A, B, C, D ,E, F, G, H, I} |V| = 9
E= { A_B, B_C, B_E, B_F, C_D, F_G, D_H, E_H, G_H, H_I } |E| = 10
S1: 2A 3B 2(3(2C 1D) 1E 2(1F 1G)) 3H 1I SA Schedule
S2: 2A 3B 12C 6D 2E 4F 4G 3H 1I Flat SA Schedule
S3: 2A 1B 2(1B 2C 1D 1E 2(2C 1D 1F 1G)) 3H 1I non-SA Schedule
sink (A_B) = B src (A_B) = A
cns (A_B) = 2 prod (A_B) = 3
q = [2, 3, 12, 6, 2, 4, 4, 3, 1] q[A] = 2 q[B] = 3

FIGURE 3.1. An example SDF graph, several valid schedules and some definitions are illustrated. Edges of the SDF graph are annotated with corresponding production and consumption rates. V is the set of vertices and E is the set of edges of the SDF graph. $S1$, $S2$, and $S3$ are three different valid schedules for the given SDF graph. Src and $sink$ of an edge are the sending and receiving actors of the edge, and $prod$ and cns of an edge are the number of tokens produced and consumed on the edge, respectively. q is the repetition vector of the SDF containing all repetition factors of the actors in the graph. Each element of q shows how many times a specific actor should be fired in total in a valid schedule.

3.1. Buffer Memory Management

Streaming applications tend to require fairly large channel buffers, primarily due to the data intensive nature of their processing and different production and consumption rates. As a result, the total size of the buffer arrays usually accounts for a substantial portion of the application binary memory footprint. Enhanced management of the buffer memory can potentially lead to considerable reduction in memory requirement, which would be of great value especially for the resource constrained embedded platforms.

In Section 2.2 of Chapter 2 we introduced the SDF graphs as the application model for streaming applications. Figure 3.1 shows an example of SDF graphs along with some of the main notations of the abstract model. We use this example to explain buffer sharing concepts in the reminder of this Chapter.

Recall that edge e in a SDF graph represents a FIFO communication channel between $src(e)$ and $sink(e)$. The channel stores the produced data after firings of $src(e)$, and its data is consumed during firings of $sink(e)$. Let $MT(e, S)$ denote the maximum number of tokens stored in channel e during firing of tasks according to schedule S. Clearly, $MT(e, S)$

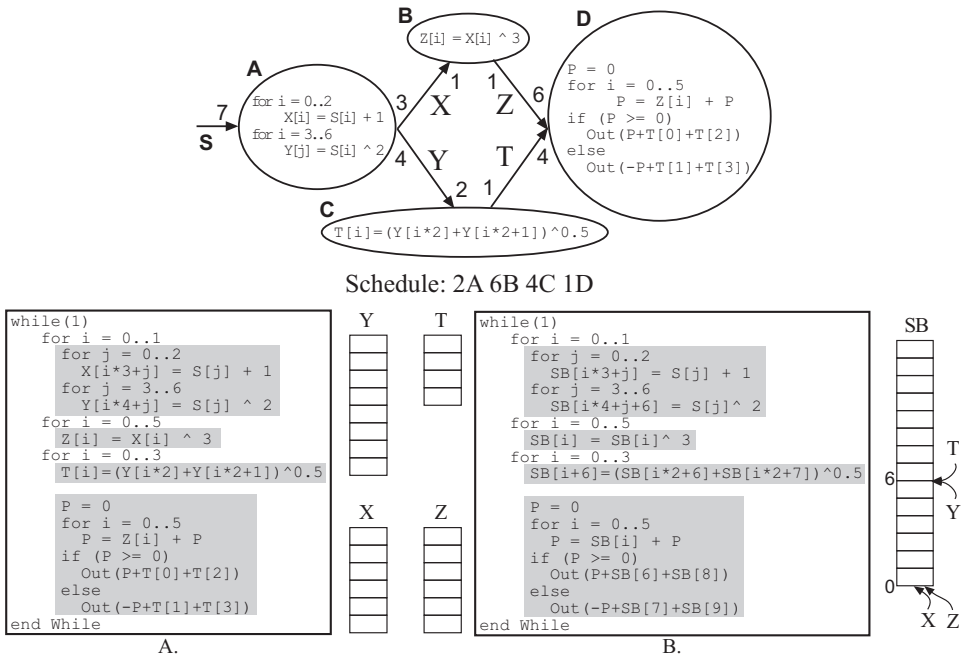


FIGURE 3.2. An example SDF, **A**. The corresponding baseline implementation. Channels are implemented as distinct buffers. **B**. Shared buffer implementation of the SDF.

indicates the minimum memory space required on this channel to implement the communication functionality.

The channels are typically implemented as buffer arrays to realize *in-order* communication with little cost. In the synthesized software, $src(e)$ writes into the buffer that implements channel e by maintaining a write index, referred to as the *Head*. The Head is reset at the beginning of the period, and is incremented after writing every token. The initial resetting enables reusing the same buffer memory in subsequent iterations. Similarly, $snk(e)$ maintains its own *Tail* index for reading from the buffer of channel e (buffer e for short), which is also reset at the beginning of the period, and is incremented after reading a token. Figure 3.2 illustrates the buffers in the synthesized code for the same example SDF given if Figure 2.2.

In presenting our work, we temporarily restrict our discussion to the aforementioned case of zero initial tokens on SDF edges, and single-appearance task execution schedule. We use the term "reference model" to refer to this case. Later in Section 3.7, we extend our discussions to demonstrate handling of both initial tokens, and non-single appearance schedules. We would like to emphasize that the temporary restriction of the discussion in

the next few Sections is merely for simplicity and clarity in presenting our work, and not a limitation of the proposed approach.

3.1.1. Baseline Buffer Allocation. Let $MT(e, S)$ denote the maximum number of tokens stored in channel e during the firing of tasks according to schedule S. Clearly, $MT(e, S)$ indicates the minimum memory space required on this channel to implement the communication functionality. Smaller buffer size would lead to an incorrect or infeasible execution under S, because at least at one point during execution $MT(e, S)$ tokens need to be stored in the buffer e. In our discussions, therefore, we assume that the size of buffer e is exactly $MT(e, S)$.

The baseline synthesis scheme would be to allocate the buffers as independent regions in the data memory. In the "baseline buffer allocation" scheme, the buffers do not share any physical memory location at any point during execution. It follows that the overall buffer size would be the sum total of individual buffers, i.e., $\sum_{e \in E} MT(e, S)$. Figure 3.2.A depicts a simple example. If the schedule is clear from the context, we use MT_e to denote $MT(e, S)$.

3.1.2. The Impact of Scheduling. Changes to task scheduling can impact individual buffer sizes, which in turn, would influence total buffer memory requirement. In case of Figure 3.1, for example, $MT(C\_D, S_1) = 2$ and $MT(C\_D, S_2) = 12$. Note that under S_1, after production of 2 tokens by the actor C the consumer (D) gets fired, which consumes all of the existing tokens in the channel. Thus, the maximum number of tokens in the channel does not exceed 2. Unlike MT, the number of exchanged tokens over an edge does not depend on the schedule, and is only a function of the SDF structure and rates.

Efforts have been made in the past to minimize total buffer size via task schedule optimization. Bhattacharyya et al. present two effective algorithms for constructing a single appearance (SA) schedule with emphasis on reducing the memory requirement [**BLM96**]. Furthermore, phased scheduling has been proposed as a method for scheduling a SDF graph to minimize the memory size considering both code and data memory [**KTA03**].

In addition to scheduling, the data memory requirement is impacted by the scheme used to allocate individual buffers in the memory. In this work, we direct our attention to this problem, i.e., minimizing overall buffer size through improved buffer analysis and allocation

techniques. That is, we seek to improve buffer management without perturbing the given schedule.

3.1.3. Buffer Sharing. In the baseline allocation scheme, separate portions of the memory are allocated to implement the channels of the SDF graph. During most of the execution time, however, the channel buffers are either partially or completely unused. For example in Figure 3.1, buffer A_B is completely empty during the firings of H and I in the schedule $S1$. Therefore, the memory allocated to this buffer can be reused to implement buffer H_I. That is, the two buffers can safely *share* at least one physical memory location during execution, without compromising the functionality of the streaming application.

Figure 3.2.B illustrates the synthesized code, under the buffer sharing assumption, for the example depicted in Figure 3.2. Buffers X, Y, Z and T are allocated at different offsets of the same array, called SB. Note that although buffers X and Z, and Y and T start at the same location in the shared buffer SB, the correctness of the computation is preserved. Extending the idea, any two channel buffers can be allocated to allow sharing of physical memory locations (space) as long as the two buffers do not conflict in time, i.e., if the two buffers do not need to maintain a token at the same memory location at the same time.

The software synthesis framework, including its code generation protocol, impacts the possibility of sharing between two buffers. In this work, we assume that code generation has to comply with the following rules:

(1) None of the valid tokens of any buffer must be over-written or read by another buffer at anytime during the execution of the program.

(2) Buffers must be statically allocated as contiguous regions in the application memory space.

(3) The data cannot be moved around within the buffer, i.e., data production and consumption operations are the only primitives that can access buffers. Token production and consumption increment head and tail indexes, respectively.

The rules collectively guarantee that the generated code implements the functionality according to the SDF semantics. They eliminate the need for implementation of a complex inter-actor communication mechanism, which would incur large performance and code size

penalty. Outstanding examples of academic and commercial SDF synthesis frameworks follow the same basic principles [**EJL$^+$03, sim**].

3.2. Granularity in Buffer Analysis

The SDF model of computation abstracts away the impact of intra-actor computations on synthesized software. That is, firing of an actor is viewed as the primitive execution operation in this model. Any valid schedule for a SDF gives the order and the number of firings of the actors. Using this sequence as a guideline, one can calculate the storage requirement (capacity) of buffers. Capacity of a buffer is time dependent due to firings of its producer, which adds tokens to the buffer and also firings of its consumer, which empties the buffer.

Such temporal changes, however, can be captured at different resolution levels [**MB01**]. The most accurate view of a buffer's temporal changes in storage requirement needs to follow the execution at the granularity of firing individual actors. In this scheme, execution of a task forms the unit of time for temporal analysis. We use the terms *fine grain* or *highest resolution* buffer analysis to refer to this level of abstraction.

Let "time step" refer to a unit of execution advancement at a given level of resolution. For example, a unit time step would refer to firing of an actor at fine granularity. For fine grain analysis we have:

$$T_{fg} = \sum_{v \in q_G} q_G[v]$$

T_{fg} : Time step in fine granularity

q_G : Repetition vector of graph G

where T denotes be the total number of time steps in one iteration of the schedule.

Nested loops make it possible to construct SA schedules in different forms. An actor might be executed in non-consecutive order, during one iteration of the SA schedule, depending on the presence and configuration of the nested loops. Note that the term "single appearance" only restricts the appearance of an actor in the closed form of the schedule.

To obtain the firing order of actors, and to accurately calculate the capacity of buffers at any time step, one has to unroll the nested loops and walk through the firing sequence

of individual actors. However, one could compromise the accuracy in analysis of a buffer's temporal pattern, by avoiding to unroll all nested loops. Specifically, temporal changes to required capacity can be crudely and conservatively estimated. This leads to the *coarse grain* or lowest resolution temporal view of the buffer storage requirement, in which the buffer has the capacity $MT(e, S)$ during its live range, and zero otherwise.

In coarse grain buffer analysis, not only loops are not unrolled, but all firings of the same actor are lumped together. That is, loop coefficients are distributed over the loop to arrive at a flat single appearance schedule. Relative to fine grain analysis, coarse grain analysis is conservative, since it allocates the maximum instantaneous memory requirement of a buffer throughout its life time.

The notion of time step at the coarse granularity refers to all firings of one actor. In other words, the coarse grain view is that all of $q[v]$ firings of actor v occur in a single time step, whereas in fine grain it would happen in $q[v]$ different time steps. Therefore:

$$T_{cg} = |V|$$

T_{cg} : Time step in coarse granularity

There can be a number of middle grounds between the two ends of the granularity spectrum. One can unroll the loops according to their nesting depth to arrive at resolution levels that are in between the two extreme fine and coarse grain cases. Unrolling each level of nested loops would introduce more time steps for temporal buffer analysis, which in turn, increases the analysis and subsequent optimization complexity. However, the unrolling would relax the conservative view of buffer requirements, by refining the temporal characteristics of buffers, which potentially creates more opportunities for memory savings.

Varying the granularity of buffer analysis leads to different number of time steps for characterization of buffers' spatiotemporal patterns in the periodic schedule. Essentially, the time steps imply a schedule guideline (SG), which captures the actor firings that are embedded in each time step. SG is not the actual sequence of the firings according to the schedule, except in fine granularity analysis. It merely captures actor firings that are lumped together in time steps, according to the analysis resolution.

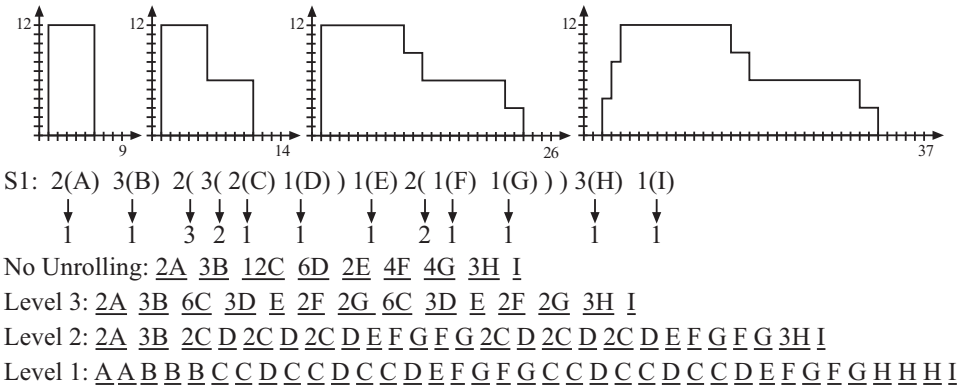


FIGURE 3.3. The impact of analysis granularity on characterization of buffer size B_F (Figure 3.1 under S_1). X and Y axis represent time steps and buffer size (number of its data tokens), respectively. Loops in the schedule are annotated with their nesting level, which are unrolled based on the analysis resolution to obtain the corresponding schedule guideline. In each level, a new Scheduling Guideline (SG) is given by unrolling the nesting loops marked with the same number or bigger in schedule S_1. The SG given in level 3 is the result of unrolling nesting loops (shown as parenthesis) marked with number 3. In level 2, the loops marked with numbers 2 and 3 are unrolled, and in level 1, loops marked with numbers 1, 2, and 3 are unrolled.

Figure 3.3 illustrates the impact of granularity in temporal analysis of buffer capacity for the buffer B_F in Figure 3.1 under schedule S_1. It also shows the schedule guidelines used to characterize the temporal behavior. The size of required space for buffer B_F grows with the firings of the producer actor B, and shrinks with the firings of the consumer actor F. For each level of granularity, the corresponding schedule guideline shows the actors that are fired in each time step, and their inter-time step order.

3.2.1. Visualizing Buffer Analysis and Allocation. Buffer analysis characterizes buffers' spatiotemporal footprint in a two-dimensional plane, in which the X-axis shows time steps, and the Y-axis represents the required storage size. As a result, buffer allocation can be viewed as placement of buffer's spatiotemporal footprint in the plane, with the minor adjustment that the Y-axis has to represent an offset in the shared memory space. Figure 3.4 visualizes buffer allocations, for the SDF example of Figure 3.1 under schedule S_1.

The gray area of each buffer illustrates the range between head and tail indices that contains valid data. The temporal update in the gray area is due to the production and consumption operations, which increment the head and tail indices, respectively. The buffers

are indexed relative to an *offset*, which indicates the start of the buffer within the shared space.

For a given analysis granularity, the capacity requirement of a buffer at any point in time is fixed. Thus, the X coordinate of buffers in the two dimensional time-memory plane cannot be modified. The Y coordinate, however, represents the physical location of the allocated memory to implement the buffers. Thus, the memory allocation problem can be viewed as geometric layout of buffer polygons, in which a solution is valid if the *laid out* buffers do not conflict in the time-memory plane. The only operation for perturbing the layout is vertical movement of the buffers. The geometric placement of a buffer in the plane readily gives its offset in the memory space. The optimization objective is to minimize the vertical dimension of the layout, which represents the total memory allocated to buffers.

3.2.2. Impact of Granularity on Buffer Allocation. The granularity in buffer analysis compromises accuracy in capturing temporal behavior of buffers with analysis and subsequent allocation complexity. Moving from coarse-grain to fine-grain analysis on the complexity spectrum, improves temporal details that enable more opportunities for buffer sharing at the cost of longer optimization runtime. The layouts in the Figure 3.4 illustrate the idea.

Figure 3.4.A shows the baseline buffer allocation scheme in which, buffers are assumed to have maximum capacity throughout the execution. Thus, they cannot share any locations during the runtime, and have to be allocated in separate locations. The Figure shows that the total size of buffers is 79.

Figure 3.4.B depicts the optimal allocation, when buffers are analyzed at the coarse granularity. This corresponds to the "no unrolling" case in Figure 3.3. In this scheme, buffers are assumed to have maximum capacity throughout their live range in the schedule. For example $MT(A\_B, S_1) = 6$, and under the coarse-grain analysis model six memory cells have to be allocated during its entire life time to implement this buffer. Thus, two buffers would conflict if they are alive in at least one point in time in which case, they cannot share any physical memory location and have to be allocated in distinct memory spaces. Live range is naturally defined over coarse grain "time steps". Under coarse grain analysis, the optimal total size of channel buffers is 42 (Figure 3.4.B).

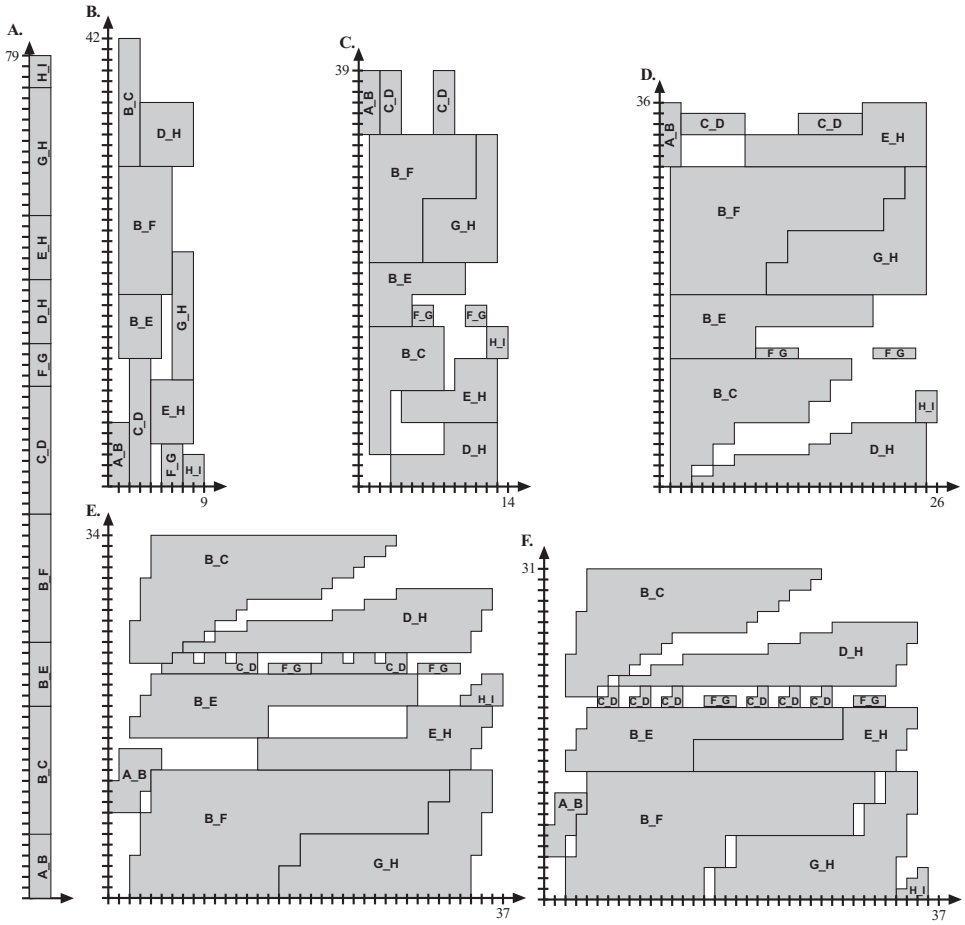


FIGURE 3.4. Buffer allocations for the example of Figure 3.1 under S_1: **A.** Baseline **B.** Coarse grain **C.**&**D.** Two possible in-between analysis points **E.**Fine grain **F.** Fine grain combined with buffer merging. X and Y axis show time steps and the offset within the shared memory space, respectively.

Figures 3.4.C and D present optimal buffer allocation for two levels of granularity in the middle of the spectrum. They correspond to levels 3 and 2 in the example of Figure 3.3. The total size of channel buffers is 39 and 36, correspondingly.

Finally, Figure 3.4.E shows the optimal allocation of buffers under fine grain analysis scheme, in which, buffers' temporal behavior is updated at the granularity of actor firings (Level 1 in Figure 3.3). Intuitively, fine-grain view of the buffers' spatiotemporal patterns enables more condensed packing of the buffers in the memory, which translates into smaller code size. In this example, the total size of channel buffers is 34.

The examples clearly demonstrate the tradeoff between complexity and quality of the allocation process, which is introduced by analysis granularity. Intuitively, as the number

of time steps increases the solution space becomes more complex, however it creates opportunity for denser packing of the polygons in the plane. We will study the tradeoff more closely in the subsequent Sections.

3.2.3. Buffer Characterization During Actor Execution. Our discussion so far has focused on analysis of buffers temporal capacity pattern at the boundary of actor firings. The SDF semantics is explicit in requiring input data being available upon firing; and input/output tokens being consumed/generated upon termination of the firing. Depending on the code generation assumptions that govern the timing of token consumption and production "during" execution of an actor, one might be able to refine the buffers further.

For simplicity, let us assume that actor v has an incoming edge e_i, and an outgoing edge e_o. For actor v to be executable, there must be at least $cns(e_i)$ tokens on e_i. Firing of actor v consumes $cns(e_i)$ tokens from e_i, and produces $prod(e)$ tokens on e_o. In the absence of any information on actor computation or code generation optimizations, one must assume that the input tokens have to remain valid during the execution of the actor. Similarly, it must be conservatively assumed that output tokens can be generated at anytime during the execution, and not necessarily upon its termination. Hence, $cns(e_i)$ and $prod(e_o)$ capacity should be allocated on e_i and e_o, respectively, during the time step of firing. Upon termination of the actor execution, the corresponding memory on e_i will become available, while e_o will have to retain the data until its consumer fires.

On another hand, it is possible that input tokens are consumed before generation of any output token. For instance, the code generation scheme might synthesize code to transfer the input tokens to local storage upon firing. As another example, the actor's specific computation might consume all input tokens before generating the output tokens. In such cases, the memory space on e_i could be used to store the produced tokens on e_o. This optimization is known as *buffer merging* [**MB04**].[3].

We use the term buffer merging to imply the assumption that actors locally store all their input tokens at the beginning of their execution, which in effect reduces the capacity of the input buffer upon firing of the consumer actor. This assumption is equivalent to assuming $CBP = 0$ in [**MB04, BM04**]. That is, the requirement to retain input tokens

[3]A complete merging would be also applicable if an actor iteratively consumes x input tokens to generate y output tokens, where $x \geq y$. All actors in the example of Figure 3.2 have this property.

during the execution of an actor would be relaxed under this assumption. The temporal variation in output buffers upon firing of producer actors remains unaffected.

Figure 3.4.F illustrates the impact of considering buffer merging during the allocation process. Assuming that buffer merging is available, buffers B_E and E_H can share their entire memory space, while without the merging assumption (Figure 3.4.E) this was not possible. Similar impact can be observed for buffers A_B and B_F, and also G_H and H_I, although in these two cases, the denser packing of polygons does not affect the total size of channel buffers. The optimal total size of buffers is reduced to 31.

Note that merging becomes possible when buffers are analyzed at the fine granularity in which, only one actor is fired in each time step. In the coarse grain or any in-between analysis, time steps do not have a one-to-one relationship with actor firings. Hence, combining merging with buffer sharing at those resolution levels would lead to incorrect implementations.

3.3. Buffer Allocation Problem

The temporal behavior of FIFO buffers can be characterized with a pair of Head and tail vectors. H and L refer to the head index $H_e[t]$ and tail index $L_e[t]$ at time step t of the schedule guideline, respectively (Section 3.2). Thus, the length of H and L are equal to the number of time steps, which depends on the analysis resolution. In the case of the reference model, head and tail indices periodically start from zero and are incremented upon write or read operations, until they reach their maximum size of $MT(e, S)$. Formally:

$$\forall e \in E \ : \ B_e = (H_e, L_e)$$

B_e : Buffer of edge e, or buffer e in short

$H_e[t]$: Head index at time $0 \leq t \leq T$ for B_e

$L_e[t]$: Tail index at time $0 \leq t \leq T$ for B_e

$T_{cg} \leq T \leq T_{fg}$ (cg: coarse grain - fg: fine grain)

The objective of buffer allocation is to assign an offset within the shared buffer space to each buffer. The offset has to be added to head and tail indices, the relative displacement within the buffer, to access the memory for read or write operations. Let vector O denote

the offset values for all the buffers of the SDF:

$$O = \{(o_{e_1}, o_{e_2}, o_{e_3}, \ldots, o_{e_M}) \mid e_1 : e_M \in E \ , \ M = |E|\}$$

$o_e$is the offset for buffer e

Given vectors H and L, the shared buffer size (SBS) can be formally stated as:

$$SBS = \max_{\forall e \in E}\{o_e + H_e^{max} \mid H_e^{max} = \max_{0 \leq t \leq T}(H_e[t])\}$$

The following Lemma specifies an important relationship between head and tail indices:

LEMMA 3.1. *Under the reference model (zero initial tokens on SDF edges, and single-appearance task execution schedule), the head index is always greater than or equal to the tail index in the same time step:* $\forall t \leq T \ : \ H_e[t] \geq L_e[t]$

Given a consistent SDF graph and a valid SA schedule, one can determine the vectors H and L. Subsequently, the objective of the buffer allocation problem is to determine the offset vector O, such that SBS is minimized, and the following constraint is satisfied:

$$\forall \ a, b \in E$$

$$\forall \ 0 \leq t \leq T, if H_a[t], L_a[t], H_b[t], and L_b[t] \neq 0 :$$

$$H_a[t] + o_a \leq L_b[t] + o_b \ OR \ H_b[t] + o_b \leq L_a[t] + o_a$$

The constraint ensures that no buffer can write to, or read from valid data of another buffer that is alive at the same time. Therefore, for buffer e at any time during its live range, other buffers have to be allocated before or after e in the memory. The offsets are determined statically and will not change in runtime. The static nature of the SDF guarantees that compile time calculation of offsets will lead to safe execution. Note that the formulation is applicable to all levels of granularity. The analysis granularity determines T, and head and tail indices in each time step.

3.4. ILP Formulation

Integer Linear Programming (ILP) provides a mechanism to obtain the optimal solution of a problem as long as its constraints and objective can be described as linear expression

of integer variables. Since there are many commercial ILP solvers available, one only has to cast the problem in ILP formulation to solve a specific instance. In case of the buffer sharing problem, linear constraints have to ensure that all buffers are allocated without any conflict.

The subtle difficulty in such formulation is to avoid buffer conflicts using linear constraints, because two conflicting buffers can be allocated in either order in the shared buffer. In other words, formulation of the "OR" logic is non-trivial, since a buffer can be allocated either before or after another conflicting buffer as long as there is no violations of the stated guidelines.

Because linear constraints cannot be easily used to articulate the "OR" logic, we had to reformulate the problem. For each buffer and each location in the shared memory space, specifically, we define a binary variables, whose '1' value would indicate allocation of the buffer in the corresponding memory location. Subsequently, buffer conflict constraints can be formulated as a large number of linear constraints that have to be generated for all time steps.

Let BBS refer to the baseline buffer size (Figure 3.4.A). BBS is known from static analysis, and is an upper bound on the shared buffer size (SBS):

$$BBS = \sum_{e \in E} MT(e, S)$$

We define the following variables to represent the head index:

$$\forall e \in E \ \forall 0 \leq i \leq BBS \ \forall 0 \leq t \leq T \ :$$
$$h_{e,i,t} = \begin{cases} 1 & \text{if } H_e[t] + o_e = i \\ 0 & \text{otherwise} \end{cases}$$

The idea is to have a binary variable for every location of the shared buffer to determine if the head index of buffer e is pointing to the location i of the shared buffer at the time t.

We also define variable o to represent the offset values of the buffers. Note that since the offset values for buffers do not change throughout the entire program, there is no index

of time in their definition:

$$\forall e \in E \; \forall 0 \leq i \leq BBS$$
$$o_{e,i} = \begin{cases} 1 & \text{if } o_e = i \\ 0 & \text{otherwise} \end{cases}$$

Vectors H and L are given, thus the number of tokens that exist in buffer e at any given point in time is known. We use variable $S_e[t]$ to refer to size of buffer e at the time t. $S_e[t]$ can be simply calculated by measuring the gap between head and tail at all points in time.

$$S_e[t] = H_e[t] - L_e[t]$$

Recall that the maximum size of buffer e in schedule S is $MT(e, S)$ (MT_e for short since we are not changing the schedule in this problem). Having defined the above variables and constants, the constraints of the problem can be stated as the followings:

(1) Each buffer must be allocated exactly once:

$$\forall e \in E \; : \; \sum_{i=0}^{BBS-MT_e} o_{e,i} = 1$$

Notice that $BBS - MT_e$ is the last possible location in the shared buffer for allocation of buffer e.

(2) At any time t, the head index points to exactly one location:

$$\forall e \in E \; \forall 0 \leq t < T \; : \; \sum_{i=0}^{BBS} h_{e,i,t} = 1$$

(3) Buffers that are alive at the same time must not conflict. That is for two buffers a and b that are both alive at time t, if head of buffer b points to location i, head of buffer a cannot point to locations between i and the tail of buffer b:

$$\forall \; co-existing \; a, b \in E \; \forall 0 \leq i < BBS \; \forall 0 \leq t < T \; :$$
$$h_{a,(i-S_a[t]),t} + \sum_{j=i-S_a[t]}^{i-1} h_{b,j,t} \leq 1$$

This constraint does not enforce buffer a to be assigned to index i. It ensure that if it was assigned, no conflicting buffer could have its head assigned to its live range at the same time.

(4) The relationship between a buffer's offset, head and its size are known at each point in time:

$$\forall e \in E \; \forall 0 \leq i < BBS \; \forall 0 \leq t < T \; : \; o_{e,i} = h_{e,(i+S_e[t]),t}$$

Note that we are only interested in finding out the offsets. The head variable is merely an auxiliary variable, which facilitates articulation of problem constraints. Once offsets are determined, the head variables are readily available.

(5) The following equation gives a lower bound on SBS. Minimizing this equation with consideration to other constraints will minimize SBS, which is the objective of the allocation problem:

$$\forall e \in E \; : \; MT_e + \sum_{i=0}^{BBS} i \times o_{e,i} \leq SBS$$

The complexity of buffer sharing ILP instance and the solver runtime grow exponentially with problem complexity. Therefore, ILP does not provide a scalable approach to solving the problem. Nevertheless, we utilize it to obtain the optimal solution to problem instances, although at the cost of unreasonably long solver runtime, primarily for evaluation of our proposed technique through measurement of the optimality gap (Section 3.9).

3.5. Strip Packing Problem and Buffer-Sharing

Buffer allocation can be viewed as a special packing of 2-dimensional polygons on the plane. Similar problems have been studied in several other industries, where there is a need for packing a set of 2-dimensional objects on a larger rectangular unit of material to minimize the waste. This larger unit can be a standardized sheet of material, from which the set of objects have to be cut. The objective is to pack all the items into the minimum number of units. This problem is a variation of the well-known bin-packing (BP) problem, and arises in some industrial applications such as wood or glass industries.

In other contexts the standardized unit is a roll of material such as a roll of paper or cloth, and the objective is to use the minimum roll length. This problem is called strip-packing (SP) problem, which is akin to the formulated buffer sharing problem. Both of these problems are known to be NP-complete, and there has been various attempts to solve them in the algorithms community [**LMM03**].

In the context of buffer sharing one can realize a large array of memory (which we call shared buffer) analogous to the roll of material in the SP problems, and the different buffers on different edges of the graph could be the set of objects. Figure 3.4 shows the geometrical aspect of buffers where we have time on one axis and the indices of shared-buffer on the other axis. In this model the objects are being constructed from the number of tokens that exist in the buffer during the run time of the program. Subsequently, we adopt a SP packing algorithm proposed in [**Jak96**] with some adjustments specific to the buffer sharing problem.

3.6. Evolutionary Buffer Optimization

Using the same concept as in 3.5, in this Section we propose an evolutionary genetic optimization technique to solve the buffer allocation problem. Our approach is to associate a cost to every sequence of buffers, and then, perturb the sequence via evolutionary optimization to improve quality.

3.6.1. Buffer Size Calculation via Move Down. Recall that buffer allocation can be transformed to a special version of 2-d packing of buffer polygons. In the packing instance, objects can only be moved vertically but not horizontally. Vertical relationship of objects in the layout can be viewed to define an order among buffers.

In particular, we take a sequence of buffers to define the order in which, the buffers are introduced to the 2-d packing problem. The associated cost, i.e. buffer size, of the order can be calculated by *moving down* the objects (buffers) toward the Y axis (beginning of the shared-buffer array) as much as possible. The packing becomes as dense as possible, for the given ordering, with the move down operation.

Move Down Algorithm (MDA) is our method for evaluation of the quality (buffer size) of a given sequence of buffers. Therefore, it will be iteratively executed during the run time of the evolutionary optimization algorithm. Figure 3.5 shows that the order of moving

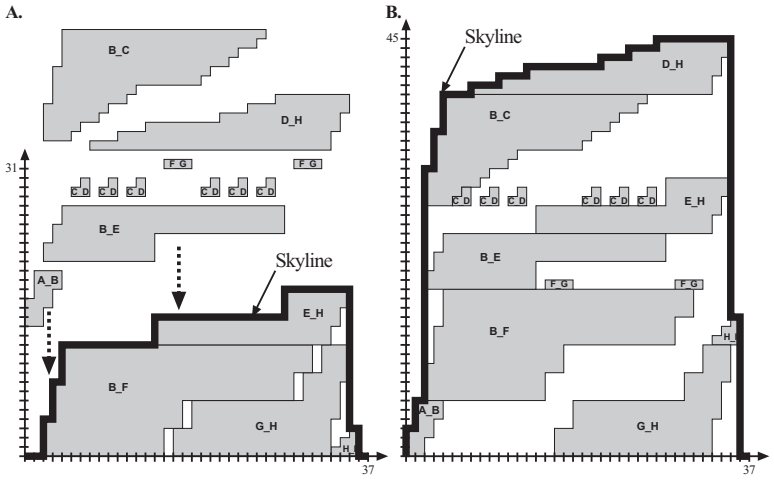


FIGURE 3.5. The impact of ordering on total buffer size for the example of Figure 3.1 under S_1 **A.** Move down of the buffers in the shown order will result in the optimal solution of Figure 3.4.F. **B.** A sub-optimal permutation: (G_H, A_B, H_I, B_F, F_G, B_E, E_H, C_D, B_C, D_H). The bold line shows the skyline.

down the buffers impacts the total buffer size. The sequence of buffers illustrated in Figure 3.5.A would lead to the allocation depicted in Figure 3.4.F. The sequence shown in Figure 3.5.B yields uncompetitive results.

To understand how far a buffer can go down we introduce another vector which is called skyline and denoted $Vsk[t]$. Here we consider buffers as solid polygons, which can stand on top of each other to construct a wall. Looking this way, skyline is the contour defined by the highest level of the constructed wall. Figure 3.5 shows two different skylines.

To construct the skyline vector we introduce skyline function which takes a Vsk and buffer B_e and also an offset o to place the buffer, and it will calculate the skyline vector $\acute{V}sk$ constructed from adding the new buffer at the point o to the existing skyline.

$$\forall e \in E \; \forall 0 \leq o \leq BBS \; \forall 0 \leq t < T \; :$$

$$\acute{V}sk = skyline(Vsk, e, o) = \begin{cases} Vsk & \text{if } S_e[t] = 0 \\ h_e[t] + o & \text{otherwise} \end{cases}$$

MDA takes a skyline vector and a buffer, and returns the lowest offset it can get from pushing down this buffer before hitting the skyline.

$$o_e = MDA(Vsk, B_e)$$

The algorithm first places the buffer on the first level of the skyline by setting the offset to $Vsk[0]$. Then moving to the right, it compares the skyline to the seated buffer to see if there is any conflict and if there is one, it will adjust the offset to remove the conflict. At the end the algorithm returns the calculated offset which does not cause any conflict with any other buffer throughout the entire program.

If we run MDA on all the buffers in a pre-defined order, and also calculate the skyline on each step and use it for the next step, we have done the buffer allocation. We call the pre-defined order a permutation of buffers and denote it with π. The function *placeAll* will then place each buffer in the same order they have in π.

The first buffer always gets zero for the offset, and it is because we are pushing down the buffers as much as possible and there is nothing in the shared-buffer yet so it goes all the way down. The very first skyline vector is $V_{zero} = [0, 0, \ldots, 0]$ which we can consider the ground. The second skyline forms when we push the first buffer down to the ground. Therefore skyline forms exactly on the top of this buffer which is the vector H. The final skyline is Vsk_{M+1} and determines the height of the wall which is actually the size of shared-buffer:

$$SBS = \max_{0 \leq t \leq T} (Vsk_{M+1}[t])$$

The size of shared-buffer depends on the sequence of buffers we are using, and *PlaceAll* itself does not guarantee that it will give us the optimal solution. However because it uses the notion of sequence in placing the buffers, it reduces the search space from $(BBS)^M$ (all the possible places for M buffers to be in an array with the size of BBS) to $M!$ (the number of different sequence of buffers that we can have). Moreover having a sequence of data as the input, is one of the fundamentals of a genetic algorithm and enables us to use them to find the optimal or near optimal solution. Algorithm 3 uses *placeAll* combined with the evolutionary part (which will be discuss in 3.6.2) to solve the buffer allocation problem.

Lastly in this Section, the following Lemma shows the MDA capability of giving us the optimal solution for the reference model:

LEMMA 3.2. *The MDA transforms the buffer allocation problem into finding the right sequence of buffers. In particular under the reference SDF model (zero initial tokens on*

SDF edges, and single-appearance task execution schedule), there exist a sequence of buffers, which gives their optimal allocation using MDA.

Extensions to other cases other than the reference model are discussed in Section 3.7.

3.6.2. Sequence Evolution using Genetic Optimization. We utilize the move down principle to construct an evolutionary genetic optimization technique. Genetic optimization is composed of several key components, including chromosome, inheritance and fitness function. Chromosome provides an abstract representation of solutions in the search space, and is normally represented as a sequence of numbers. Inheritance models the basic operations through which, chromosomes are perturbed to improve the solution quality. Typically, there are two crossover and mutation inheritance operations in a genetic optimization framework. Finally, the fitness function quantifies the "quality" of candidate solutions, and determines survival of selected candidates. Our objective is to define the notions of chromosome, inheritance and fitness, in the context of buffer sharing, and subsequently, utilize genetic optimization to solve our problem at hand.

MDA provides the ability to work on a sequence of buffers as the input (chromosome) and to allocate all of them inside the shared buffer according to their order in the sequence (Subsection 3.6.1). The size of the shared buffer is the height of the final structure, in the corresponding packing instance. We propose to use different permutations of buffers as chromosomes, or individuals of a population, and the height of the final skyline as the fitness function, in the genetic optimization framework. Consequently, the algorithm will work in the following steps:

To initialize the algorithm with a sample population, we randomly select a set of permutations. The size of the sample population is a pre-defined parameter. We used the number of buffers ($M = |E|$) to be the size of the population in our algorithm.

$$\text{Sample set} = \{\pi_1, \pi_2, \pi_3, \ldots, \pi_M\}$$

Since genetic algorithm keeps track of different lines of breeding patterns, having a larger sample population gives us the ability to keep track of more candidate solutions. On the other hand, having a very large population slows down the algorithm, and reduces the chance of finding the optimal solution in a reasonable time.

Now we can run *PlaceAll* algorithm and calculate the height of the final solution in every individual permutation in the set. We choose the height of each permutation (denoted as $height(\pi)$) to be the fitness function (denoted as $f(\pi)$) as follows:

$$f(\pi) = \frac{1}{height(\pi)}$$

For any permutation there is a chance that part of its sequence matches the sequence in the optimal solution. Basically, we would like to find these parts from different members and concatenate them, so that we can get closer to the optimum. The mechanism to recognize if we are getting closer to this goal is the fitness function. To generate new members first we need to select two of the existing members, which we refer to as parents. We select the parents depending on their fitness. The fitter individuals (shorter in height), have a higher chance of being selected. The probability of selection of an individual permutation (denoted as $p(\pi)$ is likely to change in each iteration of the algorithm due to changes to the fitness of the other members of the group.

$$p(\pi_i) = \frac{f(\pi_i)}{\sum_{j=1}^{M} f(\pi_j)}$$

In practice we can divide the interval $[0, 1)$ into M sub-intervals as follows:

$$[0, p(\pi_1)) \ , \ [p(\pi_1), p(\pi_2)) \ , \ \ldots \ , \ [p(\pi_{M-1}), p(\pi_M))$$

Two random numbers from the interval $[0, 1)$ will determine the selected permutations.

Subsequently, the parent chromosomes are used to create the children using the crossover operation. Our crossover function generates two random numbers $1 \leq p \leq q \leq M$. Then it copies the sub-sequence of the first parent from position p to q, and place it at the beginning of the child's chromosome. The sequence from p to q is the part that we would like to preserve, hoping that the same sequence exists in the optimal solution. Finally, we fill the rest of the offspring with the remaining genes (buffers) in the second parent in the same order that they appear in the second parent. The following example shows how

crossover function works:

$$p = 2 \; q = 4$$

$$\pi_{parent1} = (B_{e1}, \underbrace{\mathbf{B}_{e2}, \mathbf{B}_{e3}, \mathbf{B}_{e4}}, B_{e5}, B_{e6})$$

$$\pi_{parent2} = (\mathbf{B}_{e6}, \mathbf{B}_{e5}, B_{e4}, B_{e3}, B_{e2}, \mathbf{B}_{e1})$$

$$\pi_{child} = (B_{e2}, B_{e3}, B_{e4}, B_{e6}, B_{e5}, B_{e1})$$

Copying and matching different sequences from existing permutation may lead the process to stay in a local minimum region. To avoid this situation we can mutate the child based on the probability $p_{mutation}$, which is another parameter of the algorithm. If the child is to be mutated, then the function generates two random numbers $1 \leq i, j \leq M$, and swaps the buffers in those positions within the sequence.

$$p_{mutation} = 0.4 : \text{ the probability of being mutated}$$

$$i = 2 \; j = 4$$

$$\pi_{child} \text{ Before } = (B_{e2}, \mathbf{B}_{e3}, B_{e4}, \mathbf{B}_{e6}, B_{e5}, B_{e1})$$

$$\pi_{child} \text{ After } = (B_{e2}, B_{e6}, B_{e4}, B_{e3}, B_{e5}, B_{e1})$$

The *PlaceAll* algorithm is run on the newly generated child to calculate the height of the offspring. The child is then added to the population set. To maintain the pre-defined population of the sample set we kill (remove) the weakest (highest) member of the sample set. Therefore the offspring will be compared against the weakest member of the population, and may or may not remain in the sample set.

Iteratively, we generate new children and compare them to the existing members until the termination point where we can return the best solution found. Termination can be an acceptable size of the shared buffer (the height of the best permutation). Alternatively, the optimization can be terminated at a time limit. We selected the number of iterations as the termination criterion. We set the value to be the product of M and an iteration parameter.

Algorithm 1 Evolutionary Buffer Allocation

```
/* M : number of buffers */
/* O[1..M] : offsets */
/* f[1..M] : fitness functions */
sampleSet[ 1..M ] = createRandomSamples( M ) /* generates M different permutations
of buffers */
for i = 1 to M do
  (O[i], f[i]) = placeAll( sampleSet[i] )
end for
while !termination() do
  parents = pickParents( sampleSet ) /* randomly selects two permutations based on
  their fitness function */
  newBorn = crossOverFunction( parents ) /* applies crossover function on the two
  parents to generate the newborn */
  newBorn = mutate( newBorn ) /* mutates the newborn based on the mutation proba-
  bility */
  (O_newBorn, f_newBorn) = placeAll( newBorn )
  f_min ← min_{i=1}^{M} f[i]
  if f_newBorn ≥ f_min then
    replace newBorn by samplesSet[ min ]
  else
    discard newBorn
  end if
end while
f_best ← max_{i=1}^{M} f[i]
return O_best
```

3.7. Extension of The Reference Model

We present two extensions to the reference model that was used in our discussions so far. Specifically, we consider non-single appearance task schedules and initial tokens on the edges of SDF graphs. We highlight the implications of these two extensions for our proposed buffer allocation technique, through which, we establish its applicability beyond the reference model.

3.7.1. Non-Single Appearance Schedules. Changing the task firing schedule results in characterization of polygons with different shape and size. Under SA schedules, the producer fills in the buffer and then the consumer completely empties it (Figure 3.2.A). Therefore, the first post-consumer firing of a producer writes to an empty buffer. Under non-single appearance (nSA) schedules, however, the producer might fire after the consumer in a period, while the buffer is not *completely* empty between the two firings.

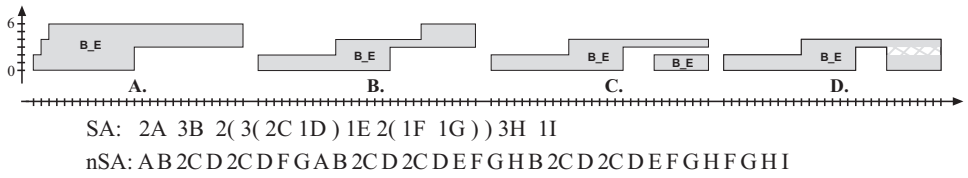


FIGURE 3.6. Geometrical characterization of buffer B_E in the example SDF of Figure 3.1, under: **A.** The given SA schedule **B, C.** The given nSA schedule assuming linear and ring FIFO implementation, respectively **D.** Conservative approximation of the ring buffer.

In addition to task schedule, the indexing strategy that the synthesizer uses to implement the buffers, impacts characterization of polygons. Buffers are typically implemented as linear or ring (circular) FIFO arrays in the synthesized software. In linear FIFOs the head and tail indices are consistently incremented in a period, whereas in ring FIFOs they might reset within the periodic schedule after reaching a maximum index.

In linear FIFOs, thus, the head index would always be greater than or equal to the tail index. In ring FIFO implementations, however, if the head index reaches its maximum value it would wrap around to the beginning to continue the write operation. As this could happen within the period of a nSA schedule, it is possible that the head index would become smaller than the tail index during the period. Lemma 3.1 does not hold under such code generation policy. Note that one must use ring buffers to bound the size of buffer e to $MT(e, S)$, under non-single appearance schedules.

Figures 3.6.A through 3.6.C depict several characterizations of buffer B_E, for the example SDF of Figure 3.1. While buffer in Figure 3.6.A is characterized under a SA schedule, Figures 3.6.B and 3.6.C illustrate the difference in characterization of the polygons under the two linear and ring buffer implementation strategies. Figure 3.7 shows the impact of linear and ring buffer implementation on the synthesized code for the example SDF of Figure 3.2, under the shown nSA schedule.

We proceed to highlight the implication of nSA schedules for our approach under three different buffer implementation strategies:

3.7.1.1. *Linear FIFO Implementation.* In linear buffer implementation, the head index is always greater than or equal to the tail index. Therefore, buffer characterization using head and tail indices as top and bottom borders yields *solid* polygons similar to the case of SA schedules, although the size of the buffer might be larger than $MT(e, S)$ (3.6.B).

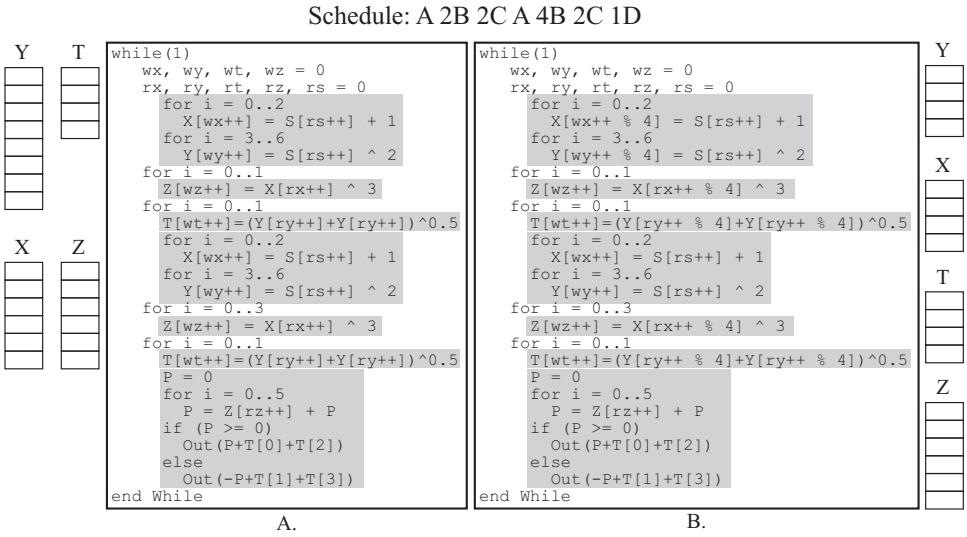


FIGURE 3.7. The synthesized code from the SDF in Figure 3.2 under the given nSA and buffer implementation **A.** linear FIFOs **B.** ring FIFOs

By solid we mean to imply that the polygons do not surround empty spaces that would be unreachable by MDA. As a result, Lemma 3.2 still holds, and MDA continues to be effective for packing a given sequence of such buffers. That is, the packing problem remains a matter of finding the right sequence of buffers presented to the MDA. Figure 3.8.B illustrates the optimal buffer allocation for our working example that is obtained by MDA under linear FIFO implementation assumption.

3.7.1.2. *Ring FIFO Implementation.* As the head index might wrap around within the periodic schedule, it could become smaller than the tail index. Thus, the characterized buffer polygons might surround empty spaces (Figure 3.6.C). It follows that in this case, the head and the tail indices do not necessarily represent top and bottom borders of the extracted geometrical object at all times, and Lemma 3.2 no longer holds. MDA would be unable to correctly pack a given sequence of such polygons, as it cannot reach the surrounded empty space.

One approach to solve this issue is to make the empty spaces within the characterized buffer polygons unavailable for sharing using bounding polygons (Figure 3.6.D). This conservative approximation enables the MDA to pack a given sequence of polygons. Although the resulting solution may lose some optimization opportunity due to the introduced inaccuracy, it is likely that the overall process would significantly reduce total buffer size. Figures 3.8.D and 3.8.E illustrate the optimal solution for our working example, obtained via hand

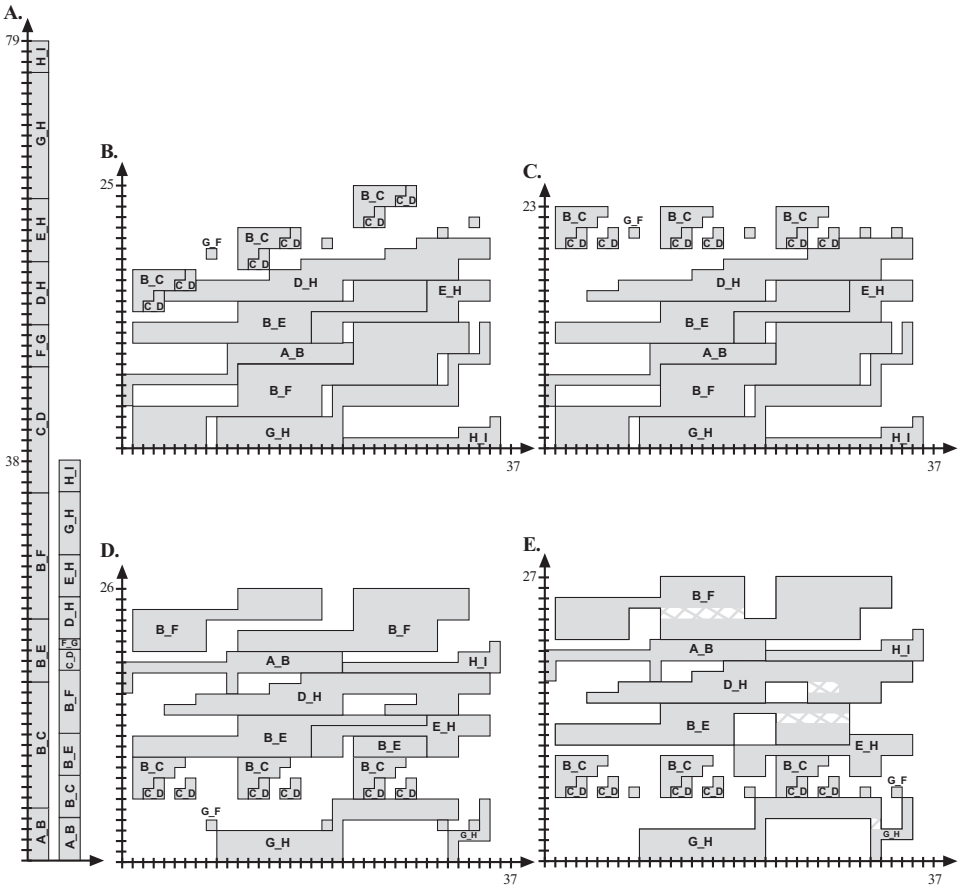


FIGURE 3.8. Buffer allocations for the example of Figure 3.1 under the nSA schedule in Figure 3.6: **A.** Baseline using linear and ring FIFO implementations **B.** Using MDA for linear FIFOs **C.** Using MDA for a hybrid implementation **D.** Hand optimized for ring FIFOs **E.** Using MDA for conservatively characterized ring FIFOs

optimization and via the MDA-generated solution after conservative approximation of the objects, respectively.

3.7.1.3. *Hybrid Implementation.* The amount of memory saving greatly depends on the ability of the characterized geometrical shapes to admit dense packing in 2-D space. A hybrid technique enables us to decide which combination of different buffer implementation strategies yields a better result. That is, implementing some buffers as ring FIFO and some as linear FIFO might be superior to adoption of only one implementation strategy for all buffers of the SDF. Searching through different combination of buffer implementations is beyond the scope of this dissertation, however, there are some intuitive hybrid approaches that are likely to reduce the memory footprint. For example Figure 3.8.C shows a packing of buffers using MDA, where a linear FIFO is used to implement a buffer unless it becomes

empty within the periodic schedule. If the buffer becomes empty during the period, we reset both head and the tail indices to the beginning of buffer, making it a ring buffer. Since our proposed buffer allocation scheme is applicable to both linear and ring FIFOs, it remains effective for hybrid implementations as well.

3.7.2. Initial Tokens. Some edges of the SDF might have initial tokens. In particular, initial tokens are necessary in cycles to avoid deadlocks in execution. For example, the model of Figure 3.9 does not admit the given schedule without at least six initial tokens on the edge C_A.

In case of initial tokens, the procedure discussed in Section 3.2 remains effective for temporal analysis of buffers at each time unit. The initial tokens cause buffers to have non-zero size at the beginning and end of the iteration, as some of the tokens produced in an iteration have to be carried over to subsequent iterations. The subtle point, however, is that the head and tail indices should maintain their value at the end of an iteration, to allow pointing to correct addresses in the subsequent iteration. That is, both indices may start an iteration from positions that are different from the previous iteration, which will result in a different geometrical characterization for the buffer. This is in stark contrast with the reference model in which, both head and tail would be reset to zero at the beginning of every iteration, and hence, the characterized polygons would be identical in every period.

We illustrate the situation with a simple example. Figure 3.9.A shows the buffer characterization for the shown SDF under the given schedule, assuming 6 initial tokens on edge C_A. In this example, both head and tail indices arrive at their initial value of zero at the end of the execution period, and hence, the buffer characterization repeats in the next period. Figure 3.9.B shows the situation, assuming 8 initial tokens on the edge. Although the head and tail indices start the period at position zero, they point to the middle of the buffer (shown with an arrow) after the first execution cycle. Since indices must continue from their latest position, the characterized polygon in the next period (Figure 3.9.C) is different from the one formed in the first period. In this example, both indices will reset to zero at the end of the second period, and thus, the buffer characterization in the odd (even) periods will be exactly the same as the first (second) period. In general, however, the

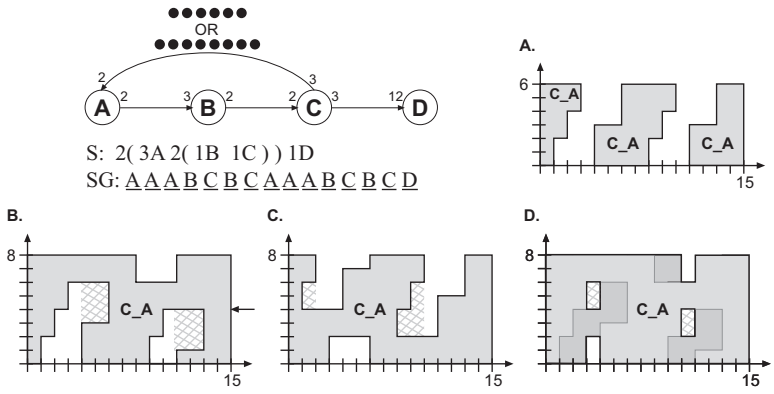


FIGURE 3.9. The impact of initial tokens on temporal behavior of the buffer C_A: **A.** The ring implementation with six initial tokens **B. C.** The ring implementation with eight initial tokens in the first and second execution periods, respectively **D.** The combination of the last two polygons using the union operation. X and Y axes show time steps and the offset within the buffer, respectively.

number of periods with distinct buffer characterizations might be more than two, although it would be finite.

Let P_e denote the number of periods with distinct geometrical characterizations for buffer e. It follows that P_e is equal to the number of different positions that the head (or tail) index takes at the beginning of execution periods. That is, after the head (or tail) index starts from a particular position, it takes exactly P_e periods for the index to start another iteration from the same position. At the beginning of the first period, the head index (writing index) points to the last initial token in the buffer. The index wraps around when it reaches the end of the buffer, and it starts the iteration from the same position after P_e periods. Hence,

$$(I_e + P_e \times TT_e) \; mod \; MT_e = I_e$$

where TT_e represent the total number of tokens produced on edge e in a period, I_e denotes the number of initial tokens on edge e, and MT_e refers to the size of the buffer. Note that the equation holds for any schedule, including nSA schedules, as the derivation solely considers the start of iterations. Solving for smallest P_e, we have

$$P_e = \frac{MT_e}{gcd(MT_e \; , \; TT_e)}$$

Correct allocation of buffers in the shared memory space must guarantee conflict-free operation throughout the execution. As buffers with initial tokens might have more than one geometrical characterization, one must avoid conflicts in all periods with distinct characterizations. One simple solution is to ignore the possibility of sharing for such buffers, and allocate them in isolation by characterizing the polygon of buffer e as a rectangle with height MT_e that spans the entire period. In effect, this would be resorting to live range analysis for such buffers.

Another way to address the issue is to conservatively approximate the geometrical footprint of a buffer as the union of its distinct characteristic polygons. Conflict-free allocation of the union polygon guarantees that the buffer remains safe in every period. Once the union polygons are generated for buffers with initial tokens, we can apply the proposed MDA-based algorithm to all buffers as before. Figure 3.9.D shows the resulting polygon for buffer C_A after combining the polygons in Figures 3.9.B and 3.9.C with the union operation. The Figure shows that even after the conservative application of the union operation to buffer C_A, some sharing opportunity exist. Note that the live range analysis scheme would not offer such opportunities as buffers with initial tokens are alive throughout the period.

3.8. Complexity vs. Optimality

The first part of our algorithm generates the polygons. The head and tail indices in each time step indicate the boundaries of polygons in that particular time step. To analyze the buffers in a given time step, therefore, buffers' head and tail indices should be updated on all of the incoming and outgoing channels of the actor that is fired. The process is iteratively carried out over all time steps to fully characterize the polygons. Let T denote the number of time steps, which is determined by the analysis granularity. The time complexity of polygon characterization is thus, $T.d$, where d is the maximum connectivity degree of the nodes in SDF graph. Clearly, $d < |V| = N$, however, practical data flow graphs are generally sparse, and d tends to be bounded by a small constant. As we discussed in 3.2, T is between N and $\sum_{v \in q_G} q[v]$ for SA schedules. If we denote the later to be Q, the complexity of polygon characterization is $O(N.Q)$.

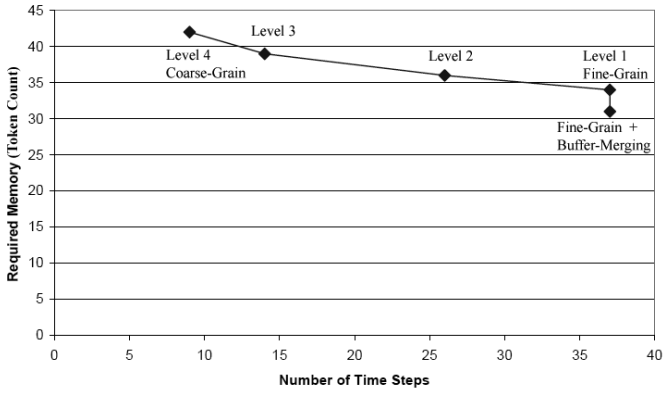


FIGURE 3.10. The tradeoff between the complexity of the buffer allocation algorithm and the minimum required memory size for the example of Figure 3.1 under schedule S_1. The reported memory size is the number of data tokens.

The time complexity of *PlaceAll* dominates the complexity of the second part of the algorithm. It is called once in every iteration of the main loop to construct valid (but not necessarily optimal) shared memory structure through packing of buffer sequences. Let G denote the total number of iterations in the main loop. MDA is the heart of *PlaceAll* and is called M times during its execution. The time complexity of MDA is linear in T. Thus the complexity of the algorithm will be $O(G.M.T)$. If we assume that G is proportional to M, the final complexity will be $O(M^2.T)$. Thus, the complexity of the second part of the algorithm is $O(M^2.Q)$.

Generally $M^2 > N$ and thus, the complexity of the second part determines the complexity of the algorithm. Similar arguments can be made under nSA schedules. The only difference is that the lower bound on T is no longer N. The upper bound remains the same, as the number of actor firings in all schedules are identical.

Buffer analysis at a finer level of granularity exposes more details, at the expense of more intensive computations. The finer details create more opportunities for improvement in the total memory size. Although an allocation of buffers can be optimal in one level of granularity, there could be more improvements, if one switched to a finer level. This tradeoff is depicted in Figure 3.10 for the example of Figure 3.1 under schedule S_1. under certain assumptions, buffer merging can be considered at the fine grain without incurring complexity penalty.

All analysis at any level of granularity will result in a valid allocation of channel buffers in the shared memory space. The best solution, from a memory size viewpoint, is obtained when the memory space is shared as aggressively as possible. This is likely to be the case at the finest resolution.

On the other hand, if the amount of memory allocated to the processor is large enough to unjustify the increase in compiler runtime, coarser grain analysis might be the superior choice, primarily because the compiler runtime might be a constraining factor in the design process. In the design development process (as opposed to the "release version" compilation), for example, memory allocation would have to be performed iteratively to accommodate algorithm development, debugging, or task scheduling and resource assignment optimizations. Thus, the compiler runtime would become critical, and it would be reasonable to trade the complexity of compilation with memory savings. Similarly, SDF compilation is sometimes necessary in a dynamic settings, where applications arrive at runtime. Examples include connection setup and migration in wireless base stations.

Conceptually, one can start the analysis from the coarse grain, and gradually move toward fine grain analysis if the target memory constraint is not met. In each trial, the violation relative to the memory constraint can serve as a clue to indicate how aggressively the analysis resolution should be refined.

3.9. Experimental Evaluation

We proceed to first report our experimental results under single-appearance task schedules. In Subsection 3.9.3, we discuss our evaluations with non-single appearance schedules.

3.9.1. Setup. We have integrated our buffer allocation techniques into the MIT StreamIt compiler [**GTK$^+$02**]. StreamIt refers to both a programming language developed for specifying the streaming applications, and a java-based compiler. The StreamIt language conforms to the SDF semantics, by modeling an application as a graph of interconnected but independent "filters" with statically-defined input and output rates that communicate via FIFO channels.

StreamIt utilizes four stream objects to hierarchically build the application graph: *filters* form the basic data processing unit, while the other three objects, *pipeline*, *split-join*, and *feedback loop* (Figure 3.11), are composite objects that contain children stream objects. The

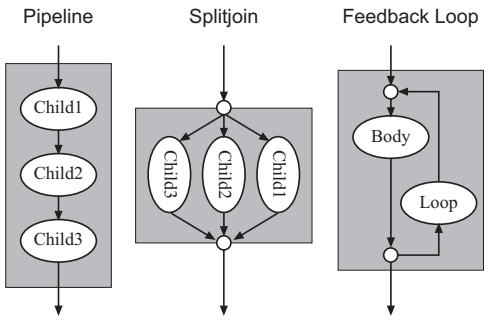


FIGURE 3.11. The composite objects of StreamIt language

children are recursively constructed out of the four different object types. In other words, the three composite stream objects (*pipeline*, *split-join*, and *feedback loop*) act as containers to build different graph structures, and the *filter* specifies data processing. The design ensures that the graph is highly structured while providing the programmers with a simple, yet flexible, set of objects to construct the stream graph for their streaming application.

We used the built-in single appearance scheduler to construct task execution order. The StreamIt scheduler is designed based on the hierarchical nature of the language. Specifically, the composite objects form virtual tasks, under which its children are scheduled. If any of the children happens to be a composite object itself, its firing in the schedule will be replaced by its own children. Consequently, only filters appear in the schedule as data processing actors. Figure 3.12 depicts the generated SA schedule by StreamIt compiler for the SDF graph in Figure 3.1.

The StreamIt compiler translates stream programs to C, which can be passed to any standard C compiler to generate executable binaries. The compiler defaults to the baseline buffer allocation scheme, in which channel in the stream graph are implemented with distinct arrays. We instrumented the compiler to allocate all the buffers within the same array, though at different indices. The baseline and instrumented synthesized codes were compiled and executed on a Unix machine to ensure that functional correctness is preserved after our transformations.

3.9.1.1. *Benchmark Applications.* We selected six different streaming kernels as our benchmarks to evaluate the proposed technique. They include two sorting algorithms, two different implementation of the fast Fourier transform (FFT), time delay estimation (TDE) and matrix multiplication kernels. These kernels frequently appear in many higher-level

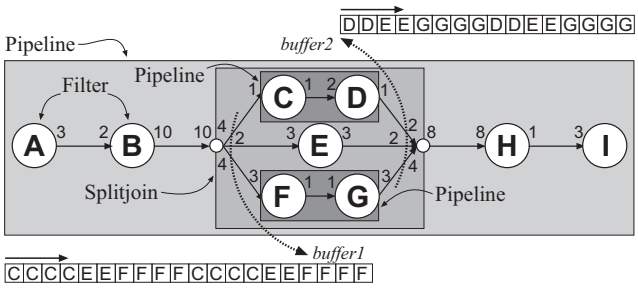


FIGURE 3.12. The split-joins share a buffer at their start and end terminals. The split-join in this picture works in round robin fashion. The letters inside *buffer1* and *buffer2* illustrate the mapping of array elements to the actors.

	# of Buffers	# of Actores	Baseline	Lifetime Analysys	Fine-Grain + Buffer-Merging	Fine-Grain + Buffer-Merging ILP	Fine-Grain	Level 2	Coarse-Grain	Coarse-Grain ILP
Bitonic Sort	119	214	1152	96	48	32	64	~	64	~
Insertion Sort	8	9	1024	256	128	128	256	~	256	256
FFT2	22	24	3072	640	384	~	384	~	384	384
FFT3	38	64	960	192	72	64	128	~	128	128
TDE	48	51	77120	23168	11776	~	12960	23040	23040	~
Matrix Mult.	10	21	5000	4000	2000	~	3000	3000	3000	~
Mean (Geometric)	27	38	3321	809	398	~	561	~	618	~

FIGURE 3.13. Benchmark characteristics and experimental results. Total buffer size under baseline, lifetime analysis, and other levels of analysis granularity is reported.

applications that are used in portable and handheld embedded systems, such as multimedia applications.

The Table in Figure 3.13 shows the benchmarks. The benchmarks are implemented in the StreamIt language and presented in the StreamIt web site [**Str**]. The second and third column of the Table list the complexity of each application in terms of number of channel buffers and actors (tasks), respectively.

Note that unlike generic SDF tasks, StreamIt filters have only one input and one output buffer. More complex inter-actor communications are modeled using split-join objects. In synthesizing split-joins, one large buffer is used to implement multiple channels that either split to or join from several actors. The sinks of a split (or sources of a join) read from (write to) the corresponding locations in the large buffer (Figure 3.12). The size of the large buffer is the sum total of the individual channel buffers, i.e., no sharing between channels is performed when allocating them in the same buffer. Consequently, the number of buffers

in the StreamIt program tends to be less than the number of actors. Figure 3.12 depicts a split-join and the buffers in its terminals.

In general, stream programs might have two different initialization and steady state execution phases. The initialization phase might be needed if a non-trivial buffer content configuration has to be created to enter the steady state phase. In this work, we have focused on the steady state buffer analysis of the stream programs.

3.9.1.2. *System and Algorithm Configuration.* The proposed genetic algorithm for buffer optimization uses a number of configurable parameters. In our experiments, we have set the iteration number of the algorithm to 1000× number of buffers in the application. In addition, the sample population in the genetic algorithm is configured to be equal to the number of buffers in the application, and the probability of mutation operation is 0.4. The experiments are performed on a Unix PC with Intel Pentium 4 CPU running at 2.80GHZ, 1024KB of cache, and 3GB of main memory.

3.9.2. Results. Figure 3.13 shows the result of baseline, lifetime analysis and our genetic algorithm based buffer allocation at four different resolution levels. The genetic algorithm is intrinsically non-deterministic. The reported results are the best buffer sizes observe out of 10 runs.

StreamIt schedules the SDF based on the hierarchy of the model. The structure of this schedule enables us to analyze the problem in different levels of granularity. Some applications have a very layered schedule in terms of the depth of nested loops, which results in more flexibility in buffer analysis. in our benchmark set, *Bitonic Sort*, *insertion Sort*, *FFT2 Sort*, *FFT3 Sort* offer only fine-grain and coarse-grain level of granularity while *TDE* and *Matrix Mult* admits an additional in-between resolution level (level 2) as well as the two ends of the spectrum.

The lifetime analysis is done according to the buffer live range analysis principle, developed by Murthy and Bhattacharyya [**MB01**]. The *first fit* heuristic is used to allocate the buffers in the shared buffer, under the same SA schedule. Note that first fit algorithm is concluded to perform well under lifetime buffer analysis model [**MB01**].

We also generated ILP instances for the benchmark application, according to the formulation developed in Section 3.4. Due to the exponential growth of the ILP complexity

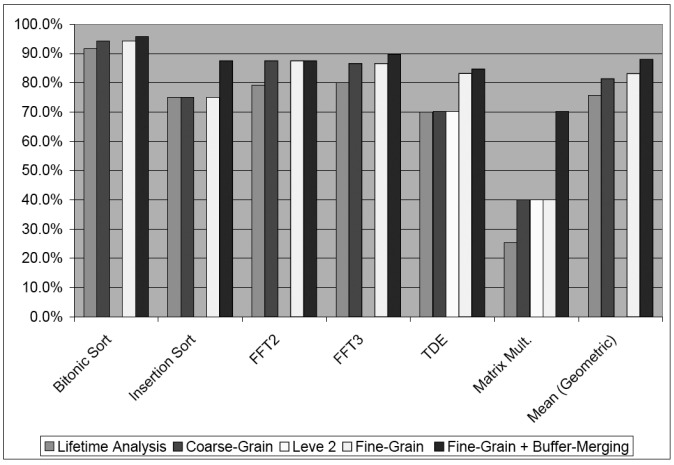


FIGURE 3.14. Savings in total buffer size over baseline allocation.

with respect to time steps, number of buffers, and the expected size of the shared buffer memory for the problem, our system was unable to load a number of large sized ILP instances. Examples include *FFT2* under fine grain plus buffer-merging mode, *Bitonic Sort* under coarse grain analysis, and *TDE* and *Matrix Mult* in both cases.

None of the ILP instances could be loaded and/or solved within reasonable time under fine grain or level 2 models for any of the applications. The ILP approach is clearly not scalable, nevertheless, it is helpful in demonstrating the optimality gap for the selected benchmarks. Our contention is that the obtained ILP solutions, in the few cases that were feasible, validate the effectiveness of our heuristic optimization.

Figure 3.14 visualizes the performance of lifetime analysis and different granularity levels in our proposed buffer allocation scheme over the baseline scheme, in terms of savings in total buffer size. On average, the proposed evolutionary optimization algorithm reduces the buffer size by 88% under combined fine grain and buffer merging model, and by 81% under coarse grain analysis.

Varying the analysis resolution trades off total buffer size with optimization latency (compiler runtime). The Table in Figure 3.15 shows the complexity of different levels of granularity in terms of the number of time steps. In addition, it shows the time spent for both the genetic optimization-based buffer allocation, and the entire compilation process, in seconds.

Figure 3.16 puts the relevant data in Figures 3.13 and 3.15 in the same chart to highlight the complexity-quality tradeoff in buffer allocation. In this graph, the conventional life range

	Lifetime Analysis Compile Time	# of Time Steps Fine-Grain	Fine-Grain GA Time	Fine-Grain Compile Time	# of Time Steps Level 2	Level 2 GA Time	Level 2 Compile Time	# of Time Steps Coarse-Grain	Coarse-Grain GA Time	Coarse-Grain Compile Time
Bitonic Sort	17.6	340	59	79.4	~	~	~	214	40	58.3
Insertion Sort	1.9	263	0.3	3.1	~	~	~	9	0.1	2.9
FFT2	3.2	446	2.7	6.1	~	~	~	24	0.5	4.1
FFT3	5.2	175	3.5	8.9	~	~	~	64	1.8	6.2
TDE	6.4	17204	454	489	1987	54.5	59.8	51	2.5	8.5
Matrix Mult.	3.1	2712	4.2	10.2	137	0.3	4	20	0.1	3.3
Mean (Geometric)	4.7	829	8.3	20.1	~	~	~	38	1	7

FIGURE 3.15. Analysis complexity in terms of the number of time steps, buffer allocation runtime (GA time), and the overall compilation runtime (compile time). Runtimes are reported in seconds.

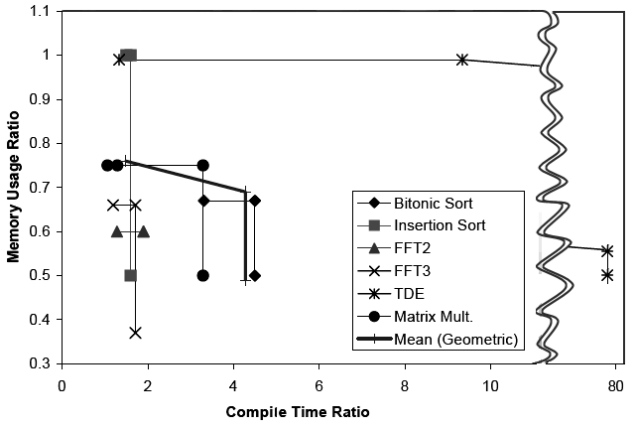


FIGURE 3.16. The tradeoff between compile time and the quality of the solution in different levels of granularity. The comparison is made based on the compile time and the memory usage in lifetime analysis for each application

analysis is the reference of comparison. The X axis shows the ratio of compile time, and the Y axis represents the ratio of total buffer size relative to the reference.

It is important to strike a balance between the two competing complexity and quality elements in practice. However, the Figure shows that the "sweet spot" is application-dependent, and occurs at different resolutions for different benchmarks. In case of TDE, for example, the 60 times increase in runtime might not justify the savings in the buffer size, under certain assumptions. However, for other benchmarks the fine grain analysis offers very good tradeoff, specially when combined with buffer merging.

Moreover, the Figure shows that for some applications such as $FFT2$, there is no gain in increasing the resolution of analysis. For such application, coarse grain analysis leads to results that cannot be improved further by finer analysis of the buffers.

	Time Steps	Linear Baseline	Linear Shared	Ring Baseline	Ring Shared	Hybrid Baseline	Hybrid Shared
Coarse-Grain	30	1021	412	744	372	851	420
Level 6	33	1021	411	636	408	743	456
Level 5	41	1021	411	546	390	635	440
Level 4	87	1021	334	312	282	521	366
Level 3	163	1021	316	178	153	300	219
Level 2	484	1021	302	58	53	248	208
Fine-Grain	612	1021	302	58	53	227	207
Fine-Grain + Merging	612	1021	302	58	53	227	198

FIGURE 3.17. Total buffer size obtained using different analysis granularities and buffer implementations, under a highly optimized nSA schedule [**KMB07**].

3.9.3. Non-Single Appearance Schedules. To demonstrate the validity of our technique even when applied to nSA schedules, we applied our technique to CD-DAT application reported in [**KMB07**]. The technique presented in [**KMB07**] generates a highly optimized nSA schedule to reduce the total buffer size.

We analyzed the application in different levels of granularity, and applied our buffer allocation method under both linear and ring buffer implementations. Figure 3.17 illustrates the results. In each level of granularity, the non-shared buffer size (baseline) and also the shared buffer size are reported under different buffer implementations. The hybrid implementation policy is similar to Figure 3.8, where buffers are implemented as linear FIFOs unless they become empty during the period.

The memory footprint reported in [**KMB07**] for this application is 58, which is equal to the baseline memory requirement in our technique using ring FIFOs (in level 2, Fine-Grain, and Fine-Grain + merging). Our technique further improves the buffer size to 53 by sharing the buffer space under the generated nSA task firing schedule.

The baseline memory requirements in Figure 3.13 is based on the StreamIt output, which corresponds to the non-shared buffers at coarsest-granularity. The baseline in Figure 3.17 is the calculated non-shared memory size in different levels of granularity. Note that some of the memory savings in each level of granularity moving from coarse-grain to fine-grain merely come from more detailed characterization of buffers. Except for linear FIFOs the baseline memory size in each level shows this reduction.

In linear FIFOs the size of each individual buffer is always equal to the total number of tokens exchanged on the corresponding edge in a period. The ring buffers, however, would be characterized differently under different analysis granularities. Thus, more accurate

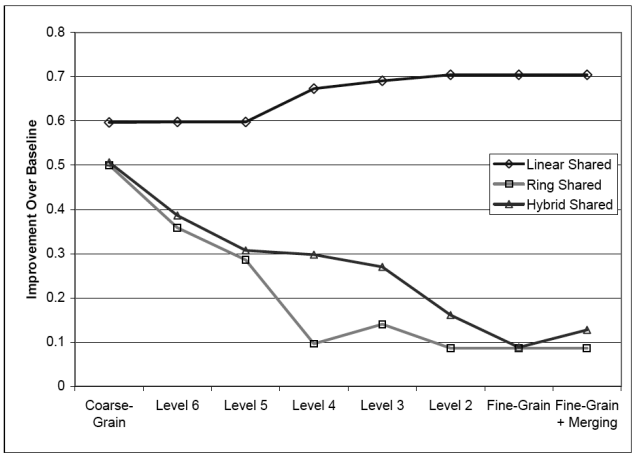


FIGURE 3.18. The improvement of using our sharing algorithm over the baseline memory size.

analysis of buffer requirement alone could lead to significant savings in total buffer size, regardless of the buffer sharing method.

Figure 3.18 illustrates memory savings obtained using our sharing technique over the baseline memory size in each level. One point worth noting in this Figure is that the amount of saving from sharing reduces moving towards finer grain levels. It is because the generated nSA schedule itself has aggressively reduced the buffer sizes, leaving less opportunity for memory savings via sharing. However, 9% improvement in finest-granularity is still significant in some application spaces, and shows that our post-scheduling technique can even complement memory-driven scheduling policies.

CHAPTER 4

Processor Mapping for Circuit-Switched GALS-Based Manycores

As the number of processors per chip grows, high-speed communication between cores becomes more challenging. Although packet-switched Network-on-Chip (NoC) architectures [**BJM+05, DRGR03, MNTJ04**] offer modular and design reusable solutions, their reliance on a single voltage-clock domain becomes a limiting factor for both performance and power reduction. Circuit-switched Globally Asynchronous Locally Synchronous (GALS) interconnects [**MVK+99, OMCM07**] offer a promising alternative to improve both factors [**TCM+09, CSC06**].

While domain-specific many-cores promise large gains in performance and energy efficiency, development of application software for their utilization remains a major challenge. Part of the difficulty lies in providing abstractions and methodologies for productive development of concurrent tasks that faithfully implement the application specification [**HFGE12, HG10a, TLA03, PHLB95, CHJ07**]. Furthermore, there is a pressing need for tools to efficiently map application concurrent tasks to platform resources. In this Chapter we present our work and results on the latter problem considering the requirements of a GALS communication paradigm.

Specifically, we study the problem of mapping a given application task graph to the processors of a given CMP platform subject to platform constraints (e.g., limited interconnect resources). In our study, the task graph is composed of concurrent software modules (tasks) that communicate via sending and receiving messages over point-to-point communication channels to implement the application functionality. In this context, a valid solution would need to determine a feasible assignment of processors to execute application tasks, and also a feasible assignment of interconnect resources to correctly implement inter-task communication channels. The quality of processor mapping, judged by metrics such as the

longest or total inter-task communication links, impacts application performance or energy dissipation.

Processor mapping finds applications at both design time (offline) and run time (online). While the quality of mapping solutions is the primary objective in design time scenarios, the run time of the mapping algorithm becomes an important criterion in online mapping. Consequently, we aim to devise extensible and scalable mapping algorithms that can 1)scale to task graphs/platforms with 1000+ tasks/cores, and 2)provide a judicious balance between solution quality and tool run time. We introduce a mapping algorithm called BAMSE (BAlanced Mapping Space Exploration) to tackle this problem. Bamse is also the name of a highly popular Swedish cartoon character for children.

4.1. Target Platform and Application Model

In this Section, we present the basic architectural features of AsAP2 processor as an example GALS chip multiprocessor (CMP), and also the target platform in our work. Although our approach is generic in nature and potentially applicable to other manycore GALS architectures, the focus of this work is on AsAP-like platforms with statically allocated non-pipelined interconnect architecture. Such platforms offer a promising tradeoff between energy efficiency and programmability for selected applications [**TTB10**]. Discussing AsAP2 as an example would bring clarity and justification to some of the decisions we make in designing the algorithm.

AsAP2 [**TCM+09**] is an academic many-core GALS processor and contains 164 programmable cores, 3 fixed function cores and 3 fixed memory modules that are interconnected via mesh topology. Each core in AsAP2 is connected to a router, and each router is connected directly to its four nearest neighbor routers with two unidirectional links in each direction. Longer communications are possible by connecting a series of links between cores. In theory, each core can communicate with any other core on the chip, however, long communications highly affect the nominal frequency of the source core even if the volume of the communication is very low [**TTB10**] (Figure 4.2). It is clear that the adverse effect on source core clock rate is monotonically proportional to the inter-core link distance.

Figure 4.1 illustrates the architectural specifications of AsAP2 [**TCM+09**]. Figure 4.2 shows the effect of interconnect distance between communicating processors and the clock

frequency of the source processor in this particular CMP platform [**TTB10**]. The drop in source core frequency is due to the fact that AsAP2 uses circuit-switched interconnection for inter-core communication, in which the clock signal of the source core is sent along with the data to maintain communication synchrony [**TTB10**]. Note that the synchronization mechanism, and its adverse effect on source core clock rate, disregards the communication bandwidth between the two cores. Thus, even an infrequent low throughput control signal would slow down the clock rate at the source core, if it has to travel far to a destination. The detailed discussion on the hardware implementation of AsAP2 is beyond the scope of this dissertation and is presented in [**TCM+09**].

Another limiting factor in circuit-switched interconnects is the limited network resources. In AsAP-like circuit-switched architectures, links are statically allocated between two communicating cores at the programming phase after reset when the application is loaded to the processor. Therefor, these links cannot be shared by other inter-core connections. This is in contrast to packet-switched networks in which, the physical resources can be shared and the limitation reduces to a constraint on total bandwidth allocated to links. Although in this approach finding deadlock free mappings will not be of any concern (since the resources are not shared), it can also reduce the number of feasible solutions to a great degree. In our target platforms, even low bandwidth control-oriented communications permanently occupy the resources allocated to the connection, and render them unavailable for allocation to other inter-core communications.

For example, in AsAP2 there are only two bidirectional links between any two neighboring cores. The restricted resources are due to the emphasis on simultaneous extreme energy efficiency and programmability philosophy of the platform design, which stresses the significance of link resource management during task mapping. Our motivation to work on this problem is partly due to the early observation that communication resource constraints would sometimes render finding of a feasible mapping, regardless of the quality, prohibitive for application developers.

The platform is particulary efficient in implementing embedded streaming applications, which are primarily characterized by the requirement to process a steady stream of input data as they are presented to the system. Such applications, are well modeled using task

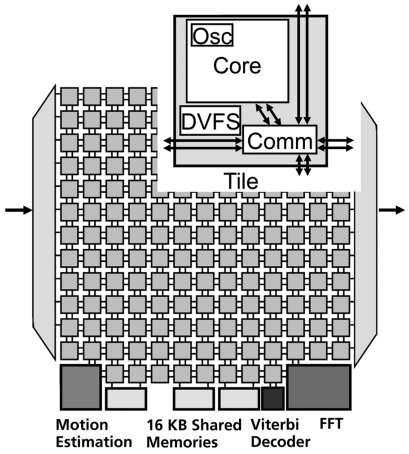


FIGURE 4.1. Block diagram of AsAP2 architecture [**TCM**$^+$**09**]. Each bidirectional connection depicted in this Figure is composed of two separate unidirectional connections in opposite directions

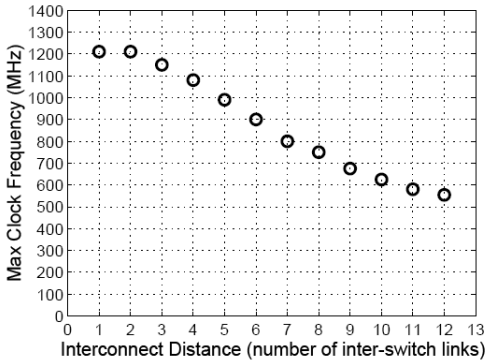


FIGURE 4.2. Measured maximum source core clock frequency for interconnect between AsAP2 processors of various distances [**TTB10**].

graphs in which, graph nodes model tasks and graph edges represent inter-task communication channels. In our discussion, we assume that application tasks are already allocated to processors in the system and will be executed on self timed schedule [**GGS**$^+$**06**]. That is, graph nodes (tasks) are viewed as virtual processors that need to be mapped to physical processors existing on the chip, and all virtual processors continue execution as long as they have input data and space available on their input and output queues, respectively.

4.2. Problem Statement

Both application and hardware platform are represented in the form of graphs:

(4.1) Task Graph: $G = < V, E >$

(4.2) Hardware Graph: $H = < C, L, CAP_L >$

In task graph G, V denotes the set of vertices, which model tasks, and E is the set of edges, which represents inter-task communication. Unless otherwise notes, we liberally use the notion of task and inter-task communication in G to refer to the set of application tasks that are assigned to the same virtual processor (i.e., a coarse-grain task), and the corresponding inter-processor communication, respectively.

The hardware graph H consists of C, which represents a set of available cores on the chip, and a set of links L, which is a subset of $C \times C$. L models possible direct physical links between cores. Each core is connected to its own router, which is responsible for receiving data from and sending data to other cores. These routers are also connected to their neighbors, and can be statically configured to implement longer communication links. Since cores have their own dedicated routers, the term inter-core communication loosely refers to the corresponding inter-router communication. CAP_L is a function that assigns a capacity number to links in L.

The mapping solution is characterized by two sets, S and R, which give the mapped processor and the allocated links for inter-task communication, respectively:

$$map(G, H) \rightarrow < S, R > \tag{4.3}$$

$$S \subset V \times C \tag{4.4}$$

$$R = \{\ path_{ij} \in P(L) \mid e_{ij} \in E\}\ P(L)\text{: power set of } L \tag{4.5}$$

All tasks must be assigned to cores, and a core can execute at most one task. Formally:

$$\forall\ v \in V\ \exists\ c \in C\ :\ (v, c) \in S \tag{4.6}$$

$$(v_1, c) \in S \text{ and } (v_2, c) \in S \Longrightarrow v_1 = v_2 \tag{4.7}$$

$$(v, c_1) \in S \text{ and } (v, c_2) \in S \Longrightarrow c_1 = c_2 \tag{4.8}$$

$path_{ij}$ refers to a lean subset of links that connects v_i to v_j in the mapping solution. The notion lean implies that all links in the subset are essential to the path, and no single link can be eliminated while the connectivity is maintained. In this work, we restrict ourselves to consider paths whose length ($|path_{ij}|$) defined as the Manhattan distance in the mesh

is minimal. This is generally not a requirement, although our experiments (Section 4.5) showed that longer paths are unnecessary from a practical viewpoint.

The mapping solution must satisfy capacity constraints:

$$\forall\ l \in L : CAP(l)\ \leq\ \Sigma_R \text{ paths that contain } l \tag{4.9}$$

That is, at most $CAP(l)$ paths can use the link l in a valid mapping solution.

The ultimate objective is to find the mapping that gives the best performance or energy profile. It is hard to accurately estimate performance and energy at the mapping level, however, it is evident that they are both adversely effected by longer connections (Figure 4.2). We use three attributes of the mapping solution as proxies for performance and energy, and use them as the optimization objective. Specifically, we use the Longest Connection (LC), Total number of Connections (TC), and bounding box Area(A) of the cores used in a mapping solution:

$$LC = \max_R\ |path_{ij}| \tag{4.10}$$

$$TC = \Sigma_R\ |path_{ij}| \tag{4.11}$$

$$A = Area(S, L) \tag{4.12}$$

The mapping problem as defined has a multi-objective optimization criteria. We use the tuple (LC, TC, A) to denote the cost of a mapping solution. Given the relative importance of the metrics, different candidate solutions must be compared lexicographically. That is, the longest connection (LC) has the highest priority for optimization regardless of total connection (TC) and area(A). In comparison of two mappings with equal LC, the one with smaller TC value would be considered to have a smaller cost regardless of the areas.

4.3. BAMSE Algorithm

We proceed to present our algorithm, called BAMSE, for solving the formulated mapping problem. BAMSE takes a constructive approach to mapping optimization, and incrementally maps application tasks onto the cores of the given hardware platform. The basic idea is to maintain a list of partial mapping solutions (initialize to empty), and incrementally augment the partial mappings by mapping a new task to an available core, while ensuring that

only a small number of promising partial mapping candidates are maintained from iteration to the next. In the remainder of this Section, we discuss these steps in detail. Specifically, we discuss iterative selection of Tasks for mapping (Task Selection); augmentation of partial mappings by finding suitable cores for mapping the task at hand (Core Selection); and judicious maintenance of a small subset of partial mappings to avoid exponential growth of retained partial solutions (Mapping Selection).

4.3.1. Task Selection. Task Selection is the process by which tasks are sequentially labeled for incremental mapping. Since the goal is to map a task as close as possible to its connected tasks, Breadth First Search (BFS) is an intuitive choice for ordering of the tasks. In BFS ordering the immediate children of a node are favored over farther nodes in the graph, however, standard BFS is silent on the tie breaking strategy when it comes to children of a node.

To order the tasks, we use the principle leveraged by Cuthill-McKee variant of the BFS algorithm [**CM69**], which heuristically aims to reduce the bandwidth of the resulting sequence of tasks. Specifically, the tasks are ordered in BFS order, while the children of a task are themselves visited in increasing order of their degree in the graph. In this context, the term "bandwidth" is historically used to denote the maximum distance (the number of tasks) between any parent and its children in the sequence, and should not be confused with bandwidth of communication channels. To avoid confusion, we use "Maximum Distance to Children (MDC)" to refer to bandwidth in graph theory, and restrict the use of bandwidth to communication performance discussions.

Figure 4.3.P1.a shows the generated task sequence using Cuthill-McKee BFS, while another variant of the BFS is used to generate the task sequence shown in Figure 4.3.P1.b. The difference is that in the former, node C is selected as the first child of node A over node B due to its smaller degree, whereas in the latter, node B has been selected as the immediate child after node A. These child ordering policies lead to $MDC = 2$ for the Cuthill-McKee BFS (only nodes H and D stand between node B and its farthest child node F), and $MDC = 3$ for the basic BFS algorithm. Intuitively, the task sequence with smaller MDC gives the advantage of visiting the other tasks connected to the current task

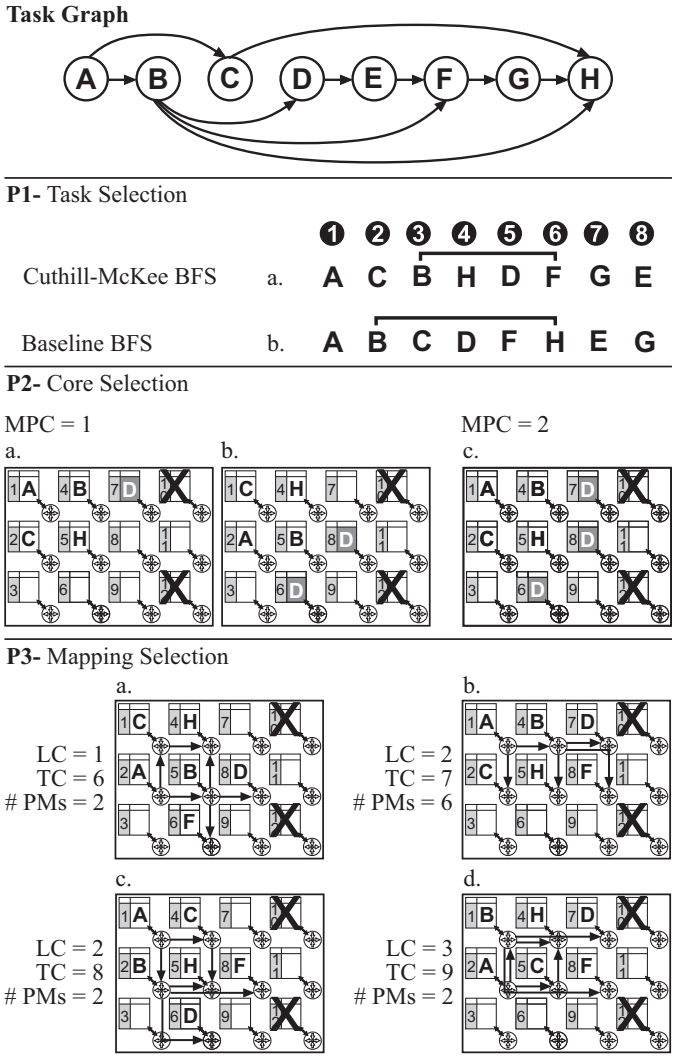


FIGURE 4.3. Snapshots of BAMSE steps on an example task graph. In the Task Selection step (P1), the difference between the Baseline and Cuthill-McKee BFS algorithms is depicted. In the Core Selection step (P2), the cores in gray are the selected cores and are considered for the current placement. The effect of different MPC numbers is also shown. And finally, in the Mapping Selection step (P3), four sample partial mappings are shown in four different cost profiles (Longest Connection and Total number of Connections). Each of the sample mappings are representative of some number of Partial Mappings belong to the same cost profile. This number is also shown as #PMs. The total number of partial mappings in all four cost profiles is 12.

earlier in the process, which heuristically results in mapping them closer to the current task.

4.3.2. Core Selection. A partial mapping (PM) is a mapping for the first k tasks in the sequence, where $k \neq |V|$. Let, v_{next} refer to the next task, namely task $(k+1)$, in

the sequence. Since the task sequence is created by BFS of a connected graph, the parents of a task are visited prior to the task itself. Thus, in any partial mapping there is a non-zero number of cores (mapped tasks) that are connected to the next task. Let "connected mapped cores" refer to the set of such cores.

In order to assign v_{next} to a core, the core selection process identifies a number of unoccupied cores as potentially good matches based on their distance to the connected mapped cores. The lower bound on expected number of potential matches is a configurable parameter, called "minimum number of potential candidate cores (MPC)". Smaller values of MPC would force the algorithm to behave more greedily, while the larger values tilt the balance towards more thorough search of the solution space.

To create at least MPC potential candidate cores, the neighboring set of each connected mapped core is created in levels. From one level to the next, the acceptable radius that defines neighborhood in Manhattan distance is incremented. The intersection between the neighboring sets of all connected mapped cores determines the potential candidate cores at a particular level, since mapping v_{next} to any of them would not create a link longer than the acceptable radius. The neighboring radius is incremented until enough number (at least MPC) of potential candidate cores are generated. Once the list of potential candidate cores are generated, existing partial mappings are augmented by mapping v_{next} to all of them. That is, for every existing partial mapping at least MPC augmented mappings are created.

In Figure 4.3.P2, tasks A, B, C, and H are assumed to have been mapped in previous iterations. The task that is being mapped in the current iteration (v_{next}) is D. Task B is the only task connected to D among the mapped tasks, thus the connected mapped cores set in the partial mapping 4.3.P2.a is $\{core4\}$, and in the partial mapping 4.3.P2.b is $\{core5\}$. These two partial mappings are only shown as examples, and potentially, there exist many other partial mappings at this iteration. The potential candidate cores for node D for each partial mapping are shown in gray color.

If $MPC = 1$, the closest available cores to the connected mapped cores set are explored until at least one possible candidate is found. At radius one, partial mapping 4.3.P2.a has one potential candidate core, however there are two potential candidate cores for the partial mapping 4.3.P2.b. If MPC is 2, we would still have enough candidate cores in the case of Figure 4.3.P2.b. However, the number of potential candidate cores in 4.3.P2.a would not be

enough (there is only one in the set). Therefore, the neighborhood radius is incremented, and farther neighbors are explored to find at least MPC total candidates. Figure 4.3.P2.c illustrates the result of expanded neighborhood.

The process process does not favor any of the candidates over the others, and it accepts all potential candidates. The term "minimum number" in MPC underscores that the number of potential candidate cores for augmentation of a partial mapping can be greater than or equal to MPC.

A subset of processor cores might be unavailable due to various factors that are discussed in Section 4.3.5. Unavailable cores are provided to the algorithm as part of platform resource description. At each step of finding potential candidate cores, the candidates are compared to the list of unavailable cores and eliminated from the set if they match the list. After this elimination the number of remaining potential candidate cores is compared to MPC to decide if incrementing neighborhood radius and exploration of farther neighbors is necessary.

4.3.3. Mapping Selection. In the first iteration of BAMSE, the first (start) task of the sequence is mapped to a core. Given the BFS nature of our task selection process, we ensure that the start task interfaces to the application input. The location of the start task might be restricted to a subset of all cores, as it has to interface with the input data stream. For example in AsAP2, the cores on the leftmost column of the chip have access to the input pins of the chip. It follows that the list of partial mappings is initialized with such possible mapping of the start task. Mapping of the start node creates a set of partial mappings to start the process. In subsequent iterations existing partial mappings are augmented.

For a given partial mapping in a subsequent iteration, the task under consideration can potentially be mapped on any of the cores in its corresponding potential candidate cores set. As a result, multiple augmented partial mappings are created from an existing partial mappings, and are added to a list, called "mapping list". The mapping list contains all points in the solution space that might have a chance to evolve into the final solution. To avoid state explosion, the mapping list is sorted in ascending order based on the cost of each partial mapping. The size of the list is also limited by the configurable parameter "Window Size" (WS). If the mapping list has WS partial mappings, each newly generated augmented partial mapping with a cost greater than the last partial mapping in the list

(i.e., highest cost) is dropped; otherwise the new partial mapping is placed in the list based on its mapping cost, and the last mapping is removed from the list to maintain its WS size. Smaller values of window size force the algorithm to be more greedy, while larger values tilt the balance towards more thorough exploration of the space at the expense of longer algorithm runtime.

Figure 4.3.P3 shows the number of possible partial mappings, when mapping task F. There are 12 partial mappings in this stage, and they all fall into one of the four cost profiles shown in Figures 4.3.P3.a, b, c and d. For each cost profile, only one of the partial mappings is depicted as a representative of the group. The number of partial mappings ($\#PM$) in each category is also reported.

In this example if window size (WS) is 2, only the partial mappings in Figure 4.3.P3.a are passed to the next iteration. The comparison criteria is the multi objective cost function presented in Section 4.2, which uses longest connection (LC) as the primary, and total number of connections (TC) as the secondary cost component. If window size is set to 8, all partial mappings in Figures 4.3.P3.a and b are kept in the mapping list. In case window size is 12, all of the partial mappings will be passed to the next iteration, where their augmentation with the next node (G) is considered.

4.3.3.1. *Look Ahead Technique (LAT).* In practice many partial mappings at the end of the list are likely to have identical cost tuples. In Figure 4.3.P3, for example, if window size is 4 then some of partial mappings in Figure 4.3.P3.b must be dropped. In this Section, we introduce a tie-breaking technique that enables us to differentiate between such partial mappings with identical costs, based on the likelihood that they will lead to superior solutions down the road.

A naive way to solve this problem would be to select an excessively large window size to ensure that most of promising partial mappings are maintained in the list throughout the run time of the algorithm. Very large window sizes, however, have adverse implications on algorithm runtime and memory requirements. Rather than increasing the window size, our approach to overcoming this problem is to look ahead in the task sequence, and quickly estimate the cost of augmented partial mapping that would result from a given partial mapping at the current iteration of the algorithm. For a given partial mapping, our "Look Ahead Technique" (LAT) quickly maps a few more of upcoming tasks. The anticipated cost

of such augmented partial mappings, called "secondary costs", is used as the tie-breaker to sort the existing partial mappings.

Secondary costs are merely needed as a tie-breaking mechanism to better sort the partial mappings at the end of the list. Thus, it is reasonable to use a quick greedy technique to map the upcoming nodes, and to establish the secondary costs. In particular, we choose to run BAMSE algorithm with $WS = 1$ and $MPC = 1$ settings for a few more steps for partial mappings that need tie-breaking. Note that the look ahead technique is only performed to estimate secondary costs, and estimated mappings are not actually committed.

We use a configurable parameter called "Forward Number" (FN) to indicate the number of upcoming tasks that are mapped by the look ahead technique. Intuitively, a good choice of forward number should allow us to consider all children of the current task in the look ahead phase. Recall from our discussion in Section 4.3.1 that the maximum distance between a task and its children in the task sequence is readily obtained by traversing the sequence, and is called maximum distance to children (MDC). Thus, we set the forward number to the sequence MDC.

The use of look ahead technique enables us to better sort the partial mappings in the list, and effectively reduces the required window size to obtain quality solutions, when compared to the baseline approach. Although it increases the complexity of the algorithm in the asymptotic sense, dealing with a smaller window size has positive effect on the runtime of every iteration. Thus, its true impact on the overall algorithm runtime is a balancing act.

4.3.3.2. *Redundant Mappings Elimination Technique (RMET).* Although no two partial mappings in the mapping list are identical, not all of them lead to different mappings in terms of solution quality. For example, the two partial mappings shown in Figure 4.4 are different, however, they yield equivalent augmented mappings of as the algorithm progresses. As such, one of them is redundant. Accurate identification of redundant partial mappings is an instance of graph isomorphism, which is a well-known NP-hard problem. In order to avoid the computational cost, we settle for an approximate test, which in practice identifies redundant partial mappings with sufficient accuracy. Specifically, we use the cost of a partial mapping, and the locations of its terminal cores as a measure of redundancy among partial mappings. The terminal cores refer to the set of cores in a partial mapping

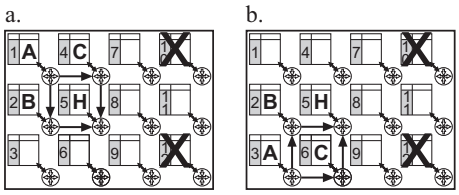


FIGURE 4.4. Both partial mappings lead to equivalent augmented mappings. The partial mapping b is basically a reflection of the partial mapping a.

that are connected to unmapped upcoming tasks in the application (set $\{B, H\}$ in Figure 4.4). Intuitively, if two partial mappings have the same cost and their cores with "open connection" are at the same locations, it is very likely that one of them is redundant. Elimination of a redundant partial mapping allows us to utilize the limited space on the mapping list more efficiently.

4.3.4. Integrated Link Assignment. Due to limited network resources on many of the circuit-switched platforms, not all task mappings would result in feasible implementations with valid link assignment between connected cores. In our experiments with AsAP, link assignment was a very constraining factor for some benchmark applications. We conclude that it is more efficient to integrate link assignment in the task mapping process, to avoid generation of infeasible mapping solutions. Since link assignment needs to be applied to the many partial mappings that are considered by BAMSE, an important design preference is fast runtime over thorough exploration of the search space.

To this end, we developed a XY link assignment algorithm that incrementally assigns links to surviving augmented partial mappings. Our XY link assignment technique only considers paths that entirely lie in the bounding box of two connected cores. In mesh interconnects, the length of all such paths is exactly the Manhattan distance between the two cores. A book-keeping table records the occupied link resources for each partial mapping. In each iteration of BAMSE, an augmented partial mapping inherits the table from its parent partial mapping, and incrementally adds new information on allocated links to the table. Subsequently, the capacity of the allocated links are updated. Infeasible partial mappings, i.e., those that cannot successfully establish all required inter-core communications with remaining link resources, are eliminated from the mapping list.

4.3.5. Unavailable Cores and Fixed Functions. Due to a number of reasons, a subset of cores might become unavailable for task mapping. In online or multi-application mapping, for example, some cores could be unavailable because another application is mapped to them. Another example would be core failure, which is probable when one deals with 1000+ core chips. Imperfect manufacturing yield and exhaustion of on-chip electronic components over time due to issues such as, aging or excessively-high local temperature, are examples of cases that could lead to core failure. Failed cores become unavailable in the hardware graph, and the mapping technique must avoid using them.

BAMSE can be readily extended to handle unavailable cores. Specifically, unavailable cores are considered during construction of potential candidate cores in the core selection stage of the algorithm. In the example discussed in Section 4.3.2, the set $\{core10, core12\}$ are unavailable for mapping. The set of unavailable cores are modeled with the $noUseList$ list in algorithm 2.

Another practical requirement is to map specific tasks of the application to certain cores that contain special resources, e.g., custom accelerators or memory resources, or other unique capabilities. For example in AsAP2, only the cores on the first (last) column of the chip can be connected to input (output) pins of the chip. In addition, AsAP2 has six tiles that implement the following functions (Figure 4.1): motion estimation, Viterbi decoder, fast Fourier transform, and 3 shared memory modules. These resources have fixed locations on the chip, which must be taken into consideration when corresponding tasks are mapped to cores. In addition, a designer may want to map particular tasks of an application to specific cores of the platform for other reasons, such as proximity of an output task to a particular output terminal used for data analysis or consumption.

We propose to extend BAMSE to handle such constrained choices. Let fixed function refer to tasks that have constraints on matching cores. Intuitively, it is efficient to map fixed functions early in the process, so the subsequent connected tasks would be mapped to adjacent cores. Otherwise, one would have to maintain a prohibitively long list of unpromising partial mappings to visit the same points in the search space. As such, we extend the node selection process to initialize the BFS queue with the fixed functions. Note that the input task is a fixed function on AsAP2, due to its constraint on interfacing with the input pins. Subsequently, the aforementioned BFS-based task sequencing scheme orders the remaining

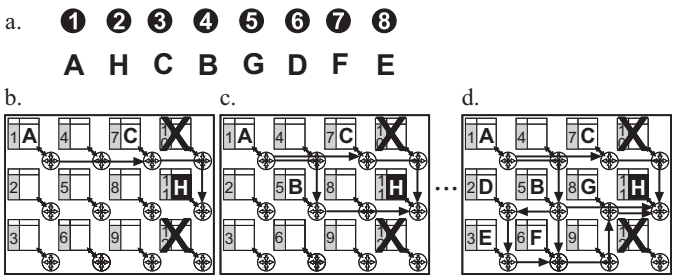


FIGURE 4.5. The effect of fixing task H on $core11$ in the mapping steps of the same task graph depicted in Figure 4.3. a. Since H is fixed and already placed, the task sequence changes in order to give more priority to the task H in the sequence. b. and c. Example partial mappings created from mapping nodes C and B, respectively. d. An example final mapping generated from the new task sequence.

tasks by visiting the children of the existing tasks in the queue. The core selection process naturally honors the dictated constraints, and otherwise the algorithm operates as described before.

For example, let us assume that the output task H of the task graph depicted in Figure 4.3 is constrained to be mapped to $core11$. Figure 4.5.a illustrates the resulting task sequence. Figures 4.5.b and 4.5.c show sample partial mappings after mapping tasks C and B. Figure 4.5.d shows the final mapping generated by BAMSE. Note that both tasks B and C have connections to tasks A and H, and the surviving cores for both tasks have the minimum distance of two to the fixed functions. The pseudo-code of BAMSE algorithm including the look-ahead technique (LAT) and redundant mappings elimination (RMET) is given in Algorithm 2. The algorithm performs under the general assumption that the input task graph does not violate any immediate feasibility constraint. For example, we assume that the maximum connectivity degree in the task graph does not exceed the architecture connectivity limitation. That is, to use AsAP2 as an example, there exists no task connected to nine other tasks. Similarly, we assume that the number of input or output tasks are not more than the number of input or output cores on the chip. Note that checking for these conditions is rather trivial, so 'assumption' is practically the same as 'ensuring' that graph does not violate immediate feasibility constraints.

4.4. Complexity vs. Quality

We start with some abstraction in analyzing the runtime of the algorithm. Let us assume the run time of allocating one core to a task (including link assignment) is T_{core}. For each

Algorithm 2 *BAMSE*

Input: G, H, $sNode$, $sCoreList$, WS, MPC, $noUseList$, $FFList$
Output: $finalMapping$
{$sNode$: Start Node , $sCoreList$: list of cores for $sNode$, $FFList$: List of fixed functions}
$inML \leftarrow \phi$
$TSet \leftarrow \phi$ {$TSet$: Terminal Set}
for $i \leftarrow 1$ to $sCoreList.size$ **do**
  $assignCoreToNode(inML[i], sNode, sCoreList[i], FFList)$
end for
$TS \leftarrow BFS(G, sNode, FFList)$ {TS : Task Sequence}
$FN \leftarrow calculateFN(G, TS)$
{FN : the number of extra nodes to map for each PM}
for $i \leftarrow 2$ to $TS.size$ **do**
  $nextNode \leftarrow TS[i]$
  $outML \leftarrow \phi$
  $TSet \leftarrow findTSet(G, TS, i)$
  for $j \leftarrow 1$ to $inML.size$ **do**
    $PM \leftarrow inML[j]$
    $listPMs \leftarrow findNextPMs(G, H, PM, nextNode, MPC, noUseList)$
    if redundant($TSet$, PM, $outML$) **then**
      continue from the start of the loop
    end if
    $endLoop = i + 1 + FN$
    if $endLoop \geq TS.size$ **then**
      $endLoop = TS.size$
    end if
    for $k \leftarrow 1$ to $listPMs.size$ **do**
      $PPM \leftarrow listPMs[k]$ { predicted partial mapping }
      for $l \leftarrow i + 1$ to $endLoop$ **do**
        $sNextNode \leftarrow TS[l]$ { secondary next node }
        $listPPMs \leftarrow findNextPMs(G, H, PPM, sNextNode, MPC = 1, noUseList)$
        $PPM \leftarrow bestCost(listPPMs, listPPMs.cost)$ { return the mapping with the best cost }
      end for
      $listPMs[k].sCost \leftarrow PPM.cost$ { set the secondary cost of current PM }
    end for
    $outML \leftarrow addSorted(listPMs, listPMs.sCost)$ { add resulting PMs to the sorted list $outML$ based on their secondary cost}
  end for
  $inML \leftarrow outML$
end for
$finalMapping \leftarrow inML[1]$
return $finalMapping$

partial mapping from the mapping list, $|PCC|$ (Potential Candidate Cores) number of cores are selected for mapping a task. Clearly $|PCC|$ cannot be larger than $|C|$ (total number of cores in the architecture). Recall from Section 4.3 that MPC is a lower bound on $|PCC|$. At each step of the algorithm, there exists at least one, and at most WS (window size)

partial mappings in the mapping list. Thus, we have:

$$MPC \leq |PCC| \leq |C| \tag{4.13}$$

$$1 \leq |mappinglist| \leq WS \tag{4.14}$$

The process above is repeated for all $|V|$ tasks in the application task graph. Therefore, using asymptotic complexity operations O and Ω, the overall runtime of the main algorithm (T_{main}) is as follows:

$$T_{main} < O(WS \times |V| \times |C| \times T_{core}) \tag{4.15}$$

$$T_{main} > \Omega(MPC \times |V| \times T_{core}) \tag{4.16}$$

Larger values of WS and MPC increase the algorithm runtime, albeit in different ways. In addition, more partial mappings are explored, which should *generally* lead to better final mappings. In the merely-hypothetical impractical case that $MPC = |C|$ and $WS = |C|^{|V|}$, the entire search space would be exhaustively explored to find the optimal solution.

As discussed in Section 4.3.3.1, the look ahead technique (LAT) is an improvement over the baseline mapping scheme, which greedily runs BAMSE for FN upcoming tasks with parameters $WS = 1$ and $MPC = 1$. Following the same principles in calculating T_{main}, the runtime of LAT (T_{LAT})is:

$$T_{LAT} < O(FN \times |C| \times T_{core}) \tag{4.17}$$

$$T_{LAT} > \Omega(FN \times T_{core}) \tag{4.18}$$

Putting it all together, the total run time of the algorithm (T_{total}) would be as follows. Note that redundant mapping elimination technique (Section4.3.3.2) does not asymptotically affect the runtime, if proper hashing techniques are used in its implementation.

$$T_{total} < O(FN \times WS \times |V| \times |C|^2 \times T_{core}^2) \tag{4.19}$$

$$T_{total} > \Omega(FN \times MPC \times |V| \times T_{core}^2) \tag{4.20}$$

Although increasing the configuration parameters is generally going to improve the quality of the final solution at the expense of longer algorithm runtime, the relationship is not strictly

TABLE 4.1. BENCHMARK APPLICATION SET SPECIFICATIONS: D is the task graph degree, and MDC is the maximum distance between a parent and its children after sequential ordering of tasks using BFS.

Application Name	# Tasks	# Channels	D	MDC
Viterbi decoder	30	35	3	4
802.11a baseband receiver	25	40	6	9
small AES	59	79	3	4
large AES	137	176	6	8
H.264/AVC encoder	115	165	7	24

monotonic due to the discrete nature of mapping optimization problem. In finding the proper value of parameters, a simple, though helpful, observation is that if a partial mapping whose augmentation could lead to an optimal final mapping survives at each iteration, the optimal solution could be successfully generated. The issue is further discussed in Section 4.5.4.

4.5. Experimental Evaluation

In this Section, we present our empirical study, whose results showcase the effectiveness of BAMSE in rapid generation of quality mapping solutions.

4.5.1. Setup. We use AsAP2 manycore chip as the target platform for mapping a number of applications. The relevant architectural features of AsAP2 are highlighted in Section 4.1.

4.5.1.1. *Benchmark Applications.* To evaluate the proposed technique we selected five different streaming applications as our benchmarks. The benchmark applications, which were previously reported in a number of technical publications, are developed, manually optimized (including task mapping) and validated for correct functionality. They include Viterbi decoder [**Wor07**], wireless LAN 802.11a baseband receiver [**TTB08**], two different implementations of Advanced Encryption Standard (AES) encryption algorithm [**LB11**], and H.264/AVC video encoder [**XLB11**] kernels. These kernels frequently appear in many higher-level streaming applications that are widely used in many embedded systems.

Table 5.2 reports the number of tasks, number of inter-task communication channels (i.e., task graph edges), task graph degree D (maximum number of connected tasks to a task), and MDC (maximum distance between a parent and its children after tasks are sequentially ordered) for all of the applications. Wireless LAN 802.11a baseband receiver

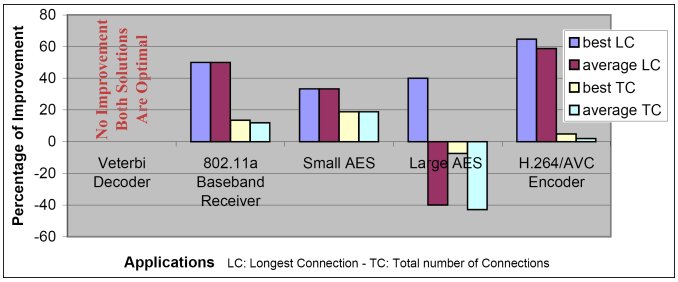


FIGURE 4.6. Improvement of BAMSE over manual mapping in longest connection (LC) and total connections (TC).

uses FFT hardware accelerator, and H.264/AVC encoder utilizes both motion estimation and FFT hardware accelerators of AsAP2 as fixed function nodes (Section 4.3.5).

4.5.1.2. *System and Algorithm Configuration.* BAMSE algorithm uses two configurable parameters: WS and MPC. In our experiments, we run the algorithm with 2400 different combinations of these parameters. Specifically, WS = 1, 2, ..., 300 and MPC = 1, 2, ..., 8. Each of these configuration points (WS, MPC) represents a level of greediness/thoroughness characteristics of the algorithm. One objective is to study BAMSE's greediness and thoroughness characteristics, such that we can strike a balance between runtime and mapping solution quality.

The objective of the mapping is to minimize the multi-objective cost function presented in Section 4.2, with the priority order of longest connection (LC), total connection (TC), and area. Area as the third objective did not make any significant difference in any of the benchmarks, and is not presented in the results. That is, the first two metrics provided sufficient resolution to differentiate among competing mapping solutions. The experiments are performed on a Unix PC with Intel Xeon CPU running at 3.07GHZ, 8192KB of cache, and 6GB of main memory.

4.5.2. Results. Figure 4.6 illustrates the improvement of BAMSE over manual mapping in longest connection and total connections of the mapped task graph. All applications were developed and mapped before the start of our mapping project, and the developers had the incentive to improve the mapping result as it simplified their work and impacted their reported performance and throughput Figures in their published work [**Wor07, TTB08, LB11, XLB11**]. Therefore they are representative of what manual mapping can achieve. This is contrast to the predominant notion of comparison with manual

optimization in CAD community in which, manual results are derived in parallel to automated results. In fact, our work was motivated by the programmers' feedback that task mapping is tedious, error-prone and unscalable to large graphs. They thought their manual mappings for published applications are quite optimized when they posed the problem to us.

In principle, hand optimized mapping should give the optimal result, however, the sheer size and complexity of solution space (large applications with 100+ tasks and 150+ links, and architecture interconnect constraints) prevent humans from efficient exploration of the search space. Out of 2400 configuration settings, both best case and average case results are depicted in the Figure. The improvements in longest connection (LC) are as high as 65%. In most cases, the second objective of total connection (TC) is also improved. Given the priority-based definition of cost function, it is not necessarily possible to minimize both LC and TC in the absolute sense. In fact, sometimes reducing one increases the other as a compromise. However, Figure 4.6 shows that for most applications manual mappings are improved in terms of both objectives. The average result of both objectives is also shown in this Figure.

For comparison, we also generated integer linear program (ILP) instances of the simplified (excluding link assignment) mapping problem. The objective of ILP formulation is to minimize the first two component of the mapping cost, i.e., longest connection and total connections. Details of ILP formulation of the mapping problem is not presented, for brevity. Although ILP is a well-known NP-hard problem and its runtime scales very poorly with input problem size, it is potentially helpful in establishing lower bounds on possible solution quality, and in quantifying the gap between the generated solutions and the optimal mappings.

In order to accelerate the ILP solver runtime, we occasionally leveraged knowledge of the problem to expose a smaller search space to the solver. Specifically, for smaller applications it is sometimes evident that the optimal solution would only use cores in a small region of AsAP2. In such cases, a smaller instance of the hardware mesh was used in generation of the ILP instance. In case of our study, in particular, a 6x6 mesh of cores was used as the target platform for *Viterbi decoder* and *802.11 baseband receiver*. We tried to solve the ILP

instances using CPLEX, a commercial grade ILP solver. The solver was allowed to run for a maximum of 10 days.

In addition to manual mappings and ILP solutions, we also implement CastNet algorithm [**Tos11**]. CastNet is a fast constructive algorithm specific to packet-switched NoC platforms with the ultimate goal of reducing the energy consumption of the system. CastNet uses the bandwidth information from the task graph and the distance between the connecting cores in the mapping solution to calculate the energy consumed for communication in the mapped application. As it was discussed in Section 4.1, bandwidth is not an important measure in the context of circuit-switched GALS architectures. In fact, BAMSE never considers bandwidth in its cost function calculation. In order to adapt the benchmark applications for the CastNet algorithm and at the same time maintain the constraints and requirements of the AsAP2 architecture, we equally assign a fixed bandwidth on each edge of the task graph.

Table 5.4 shows the comparative study data. In addition to the longest connection and total connection of mapping, algorithm runtime is reported. As expected, manual mapping can generate good results for small graphs, however, the approach does not scale to somewhat more complex task graphs. In cases that we could solve the ILP formulation, BAMSE indeed had found the optimal solution dramatically faster. For the 3 more complex applications the ILP solver did not finish after 10 days. Interestingly, in these cases the best solution found by ILP after 10 days is far inferior to BAMSE results generated in fraction of the time.

The CastNet results in Table 5.4 suggest that although the mapping problems in circuit-switched and network-switched architectures are somewhat similar, the impact on the outcome can be dramatic if the subtle differences between these two architectures are not taken into account. For example, one of the main reasons that CastNet and similar algorithms like it perform so poorly when it comes to mapping task graphs for AsAP-like architectures is that their methodology relies greatly on the bandwidth information of the graph, whereas, in AsAP-like architectures bandwidth looses its significance in the problem. When the applications are given to CastNet with equal bandwidth on all communication channels (each edge of the task graph), the algorithm performs almost as it is making random choices at

TABLE 4.2. BAMSE VS. ALTERNATIVES: Results for longest connection and total number of connections are reported. A smaller hardware platform (a 6x6 mesh of cores) is used for generation of ILP* instances to accelerate the solver runtime. The ILP** numbers are obtained by terminating the solver after 10 days, and are not optimal.

Application		Long Conn	Total Conn	Time
Viterbi Decoder	Manual	1	35	-
	BAMSE	1	35	1 (sec)
	CastNet	8	100	< 1 (sec)
	ILP*	1	35	46 (hours)
802.11a Base-band Receiver	Manual	6	58	-
	BAMSE	3	51	13 (sec)
	CastNet	8	79	< 1 (sec)
	ILP*	3	51	58 (hours)
Small AES	Manual	3	106	-
	BAMSE	2	86	2 (sec)
	CastNet	11	228	< 1 (sec)
	ILP**	3	105	10 (days)
Large AES	Manual	5	254	-
	BAMSE	3	273	170 (sec)
	CastNet	16	995	< 1 (sec)
	ILP**	5	328	10 (days)
H.264/AVC Encoder	Manual	17	353	-
	BAMSE	6	336	273 (sec)
	CastNet	16	702	< 1 (sec)
	ILP**	7	288	10 (days)

each step. It is because bandwidth which is an important deciding factor in CastNet does not make enough distinction between existing choices at each step.

Another contributing factor which amplifies the effect of lacking bandwidth information is the size of the benchmark graphs. When dealing with large graphs, any early mistakes or bad choices at the beginning of the mapping process can lead to a great departure at the end. The reported results in Table 5.4 demonstrate the effect of these two important factors between CastNet and BAMSE and in general highlight the need for a new approach specific to solving the mapping problem for circuit-switch GALS architectures.

4.5.3. Impact of Core Failures. To demonstrate BAMSE ability to handle mappings in the existence of unavailable cores, the following three scenarios are provided. In one scenario all cores are available, and in the other two some number of cores are failing thus are avoided in the mapping process. The failing cores are selected randomly and kept the same in different applications. In the first failing case 4 cores are failing which is almost 2% of AsAP2 cores, and In the second case 9 are failing which is almost 5% of the cores. In each case we run BAMSE with 50 different configuration points as follows and choose

TABLE 4.3. CORE FAILURES: Results for longest connection and total number of connections in three different core failure scenarios are reported.

Application		0 core No Failure	4 cores 2% Failure	9 cores 5% Failure
Viterbi Decoder	Long Conn	1	1	1
	Total Conn	35	35	35
802.11a Baseband Receiver	Long Conn	3	3	3
	Total Conn	51	53	53
Small AES	Long Conn	2	2	2
	Total Conn	86	87	88
Large AES	Long Conn	3	3	4
	Total Conn	269	285	298
H.264/AVC Encoder	Long Conn	7	8	8
	Total Conn	310	343	340

the best results: $WS = 30, 60, ..., 300$ and $MPC = 1, 3, ..., 9$. The results are reported in Table 5.3.

The discrepancy between the results in Table 5.4 and 5.3 in the case when none of the cores are failing is because of the difference between the number of times the algorithm is run with different configuration points (2400 in Table 5.4 vs. 50 in 5.3).

The discrete nature of mapping optimization problem combined with randomly selected failing cores makes it impossible to predict the outcome of these scenarios. Moreover, no other mapping technique handles such cases to be able to compare our results against their mappings. However, the closeness of the results between the constrained failing cases and non-failing cases proves the effectiveness of our approach.

4.5.4. Impact and Selection of Configuration Parameters. In this Section, we discuss our experimental results on the impact of configuration parameters WS and MPC on the algorithm runtime and mapping quality. Since cost of the optimal mapping solution is not generally known, we measure the quality of a particular mapping with respect to the best mapping that we could find in the target parameter space. Namely, let the *Relative Cost* of a particular mapping be the normalized increase in its mapping cost, relative to the best mapping found in the explored configuration parameter space:

$$Relative\,Cost^{app}_{(ws,mpc)} = \frac{mapping\,cost^{app}_{(ws,mpc)}}{min\,mapping\,cost^{app}} - 1 \tag{4.21}$$

Mapping cost is the multiobjective cost function presented in Section IV, with the priority order of longest connection (LC) and total connection(TC) respectively.

The small charts in Figure 5.16 show the algorithm runtime at each configuration point (WS, MPC) in the parameter space. Although the general direction relationship in runtime with increase in both parameters is evident, the relationship is not strictly monotonic. This is primarily due to the difference in the time spent on link assignment, which tends to vary differently from one partial mapping to another. Within the link assignment algorithm, BAMSE recursively resolves conflicts as it allocates links to paths between connected cores. The conflict frequency and degree of congestion, which determine the frequency and depth of recursive calls, are highly benchmark dependent. Nevertheless from a high level viewpoint, the general trend is a direct linear relationship with WS and MPC parameters.

The aforementioned relationship trend between runtime and parameters holds fairly consistently for solution quality as well. The charts in Figure 5.16 illustrate the relationship between the $Relative\,Cost$ of generated mapping solutions, and the window size (WS) for different MPC parameters. The charts for Viterbi and 802.11a baseband receiver applications are not presented in this Figure for brevity. However, small AES application well represents these two applications as well since they all consist of relatively small task graphs compare to the other two bigger applications. Similar to runtime, the data show a general trend of cost decrease with growth of configuration parameters, although the trend is not strictly monotonic, due to the discrete nature of the underlying search space.

For example, in case of small AES applications, all of (WS, MPC) configuration points larger than a small threshold result in the best (known) mapping. In such cases, if the WS are large enough (e.g. the break point of $WS > 40$ in small AES), the best mapping solution is universally found. In more complex applications on the other hand, the anticipated threshold values might be prohibitively large, in terms of computation time and memory requirements.

Consequently, an important objective is to select the configuration parameters such that a reasonably optimized solution, as compared to the best mapping solution that exists in the configuration parameter space, is generated. Clearly, one would like to accomplish this without explicit knowledge of the task graph structure, its relevant properties and without exhausting all points in the configuration parameter space. Given the discrete nature of the problem, it is not possible to guarantee optimality in the target parameter space without exhausting all of their possibilities. Thus, we expect the users to define their acceptance

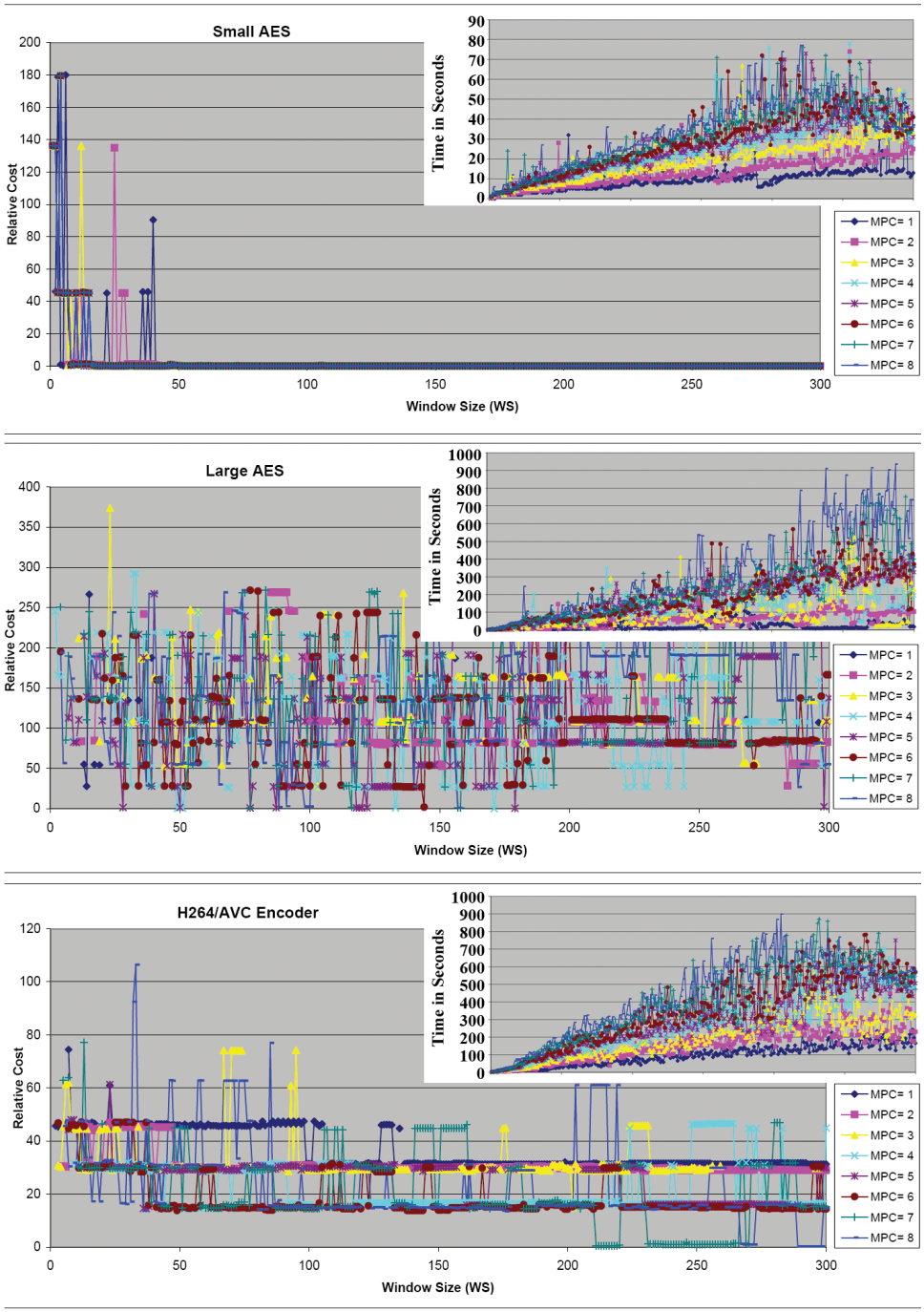


FIGURE 4.7. The effect of increasing WS on the quality of mapping solutions for different MPC and benchmark applications. Relative cost numbers are calculated using equation 4.21, and quantify the normalized distance from the best known solution in the (WS, MPC) configuration parameter space. The effect of increasing WS on BAMSE runtime for different MPC values and benchmark applications is also given in the smaller charts. The recursive and application-dependent nature of link assignment is the primary reason for runtime fluctuations.

threshold for degradation in the quality of the final mapping solution. This decision can be made by users based on various criteria such as required throughput, energy budget, and the reasonable tool runtime.

We propose to randomly select different configuration parameters from the target space, run BAMSE with the selected parameters, and output the best solution found in the trials. As the number of trials increases, the probability of reaching an acceptable solution quickly improves. Specifically, the failure to find an acceptable solution requires failure in all of the trials, whose probability quickly decreases for realistic data sets, e.g., those illustrated in Figure 4.8.

Specifically, let θ be the threshold for the acceptable degradation in mapping quality, and $N(\theta)$ represent the number of solutions in the target parameter space, whose cost is not more than $1+\theta$ times larger than the best solution in the parameter space. The probability of not finding an acceptable solution within this acceptance level can be calculated using equation 4.22.

$$P_\theta^k = 1 - \prod_{i=1}^{k} \frac{N - i + 1 - N(\theta)}{N - i + 1} \tag{4.22}$$

where k is the number of trial runs, and N is the total number of points in the parameter space. Assuming a lower bound on the number of acceptable solutions in the space, $N(\theta)$, one can calculate the number of trials that lead to generation of an acceptable solution with the desired confidence level.

For three different acceptance thresholds θ, Table 5.1 illustrates the number of acceptable solutions in the parameter space $N(\theta)$. For each solution, the mapping cost in terms of the longest and total connection are reported for comparison. The value $\theta = 0$ corresponds to the optimal solution in the target parameter space, which has 2400 points. Note that as the acceptance threshold is relaxed the number of acceptable solutions grows, and hence the number of trials required to find an acceptable solution with 95% confidence is decreased. Figure 4.8 visualizes the relationship between number of runs (k), acceptance threshold (θ), and the probability of obtaining an acceptable solution for the benchmark applications.

As a design decision, the user can decide the number of trial runs and the range in which the tool should explore the parameters. Given the relatively short runtime of BAMSE,

TABLE 4.4. MAPPING QUALITY AND THE NUMBER OF ACCEPTABLE SOLUTIONS ($N(\theta)$) UNDER DIFFERENT THRESHOLDS (θ): The longest and total connection values correspond to the lowest quality mapping that met the given threshold.

Application	Threshold (θ)	Long. Conn.	Total Conn.	#of sol. ($N(\theta)$).	95% conf. Trials
Viterbi decoder	0%	1	35	2154	2
	10%	1	35	2154	2
	50%	1	35	2154	2
802.11a baseband receiver	0%	3	51	162	43
	10%	3	53	1983	2
	50%	3	58	2385	1
Small AES	0%	2	86	2248	2
	10%	2	97	2334	1
	50%	3	100	2368	1
Large AES	0%	3	273	1	2280
	10%	3	328	10	621
	50%	4	343	155	45
H.264/AVC encoder	0%	6	336	4	1265
	10%	6	356	10	621
	50%	9	456	2150	2

multiple runs are likely to be justified to improve the mapping quality in offline mapping scenarios. In the time constrained online mapping use cases, tool runtime tends to be a hard constraint, and hence, a small number of runs are appropriate.

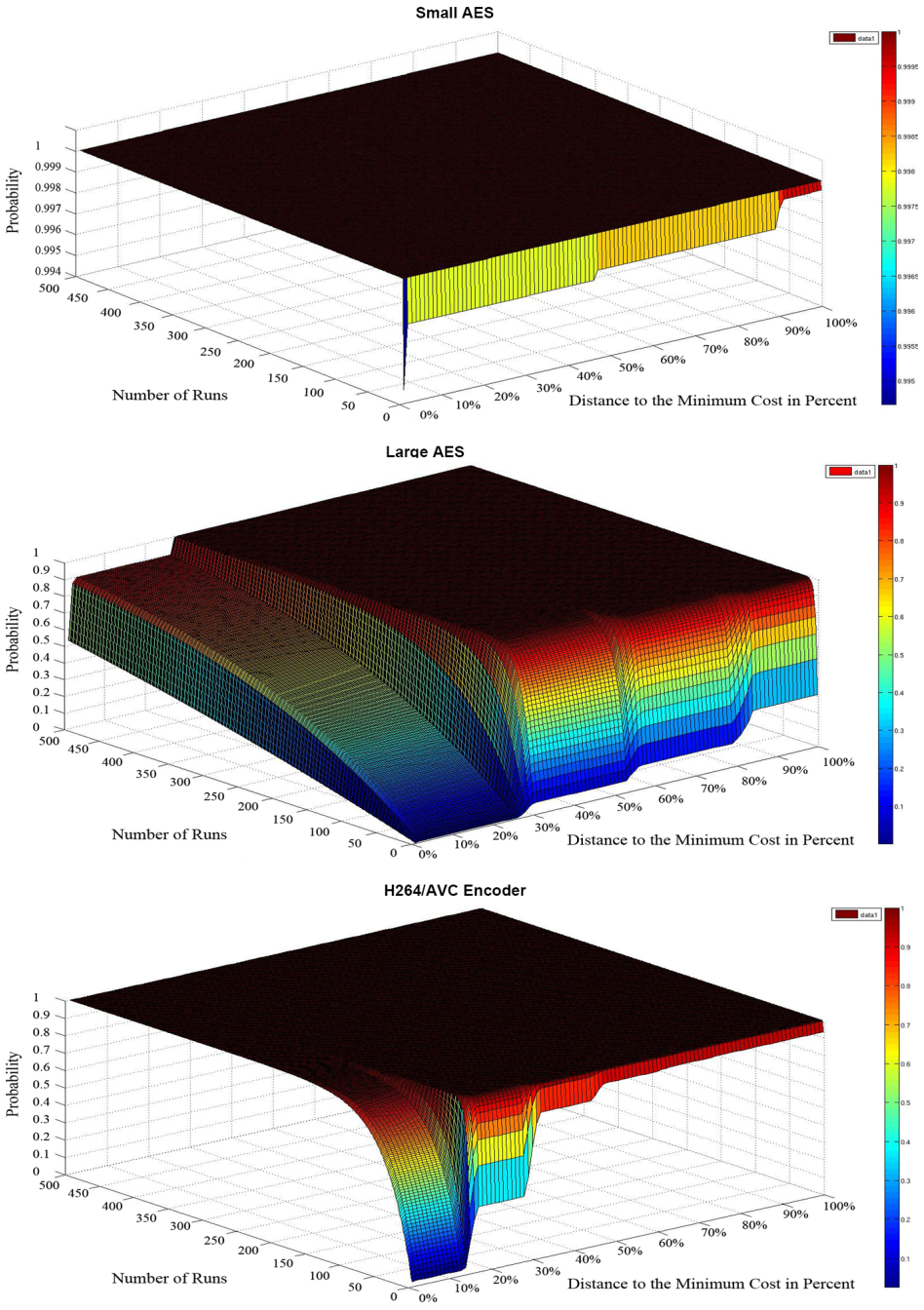


FIGURE 4.8. The probability of generating an acceptable solution under a given acceptance threshold and different number of random parameter trials. 0% distance represents the solutions with the same cost as the best solution, and 100% distance represents the solutions twice as costly as the best solution.

CHAPTER 5

Memory Access Analysis and Optimization

The software synthesis process involves a number of algorithmic steps, such as task scheduling and buffer allocation, and culminates in code generation. The generated code can be passed to a standard compiler to generate executable binaries. The synthesized software tends to follow specific styling conventions. For example, inter-task data communication is typically implemented via buffer arrays that are written to (read from) by the data producer (consumer).

In principle, specifying the application as a set of tasks and inter-task communication channels promises portability in regard to the final hardware platform. However, research shows that synthesizing software from a large SDF graph for a platform with small number of processors will result in performance loss, compared with synthesizing a smaller graph of the same application when the same platform is targeted [**HFGE12**]. It is partly because the overhead of coordinating among different tasks is justified only if sufficient amount of parallelism exists in the platform.

When the number of processing units (cores) are considerably less than the number of tasks in the graph, a large number of tasks are forced to be executed on a single core. On the other hand, the nature of dataflow streaming applications involves repeated passing of data from one task to another. This tends to result in redundant array accesses in the synthesized code if some tasks merely reorder, duplicate or drop (e.g., down sampling) data tokens[1], and pass them downstream to other tasks.

Since the behavior of SDF graph is statically analyzable, one can trace array memory accesses to characterize redundant array accesses to a large extent. In this Chapter, we demonstrate the issue through examples and experiments, and present an algorithm called RACE (Redundant Array Copy Elimination), which exploits the statically analyzable nature of SDF execution to identify and optimize such cases. RACE is a source-to-source

[1]as opposed to change the data values via arithmetic and logic operations

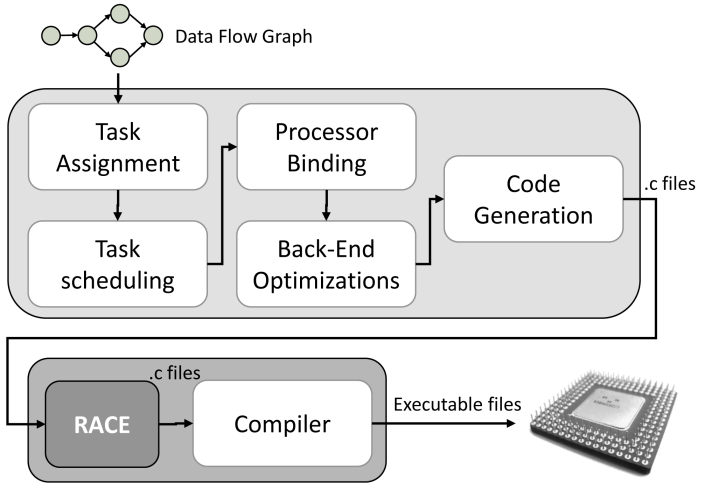


FIGURE 5.1. The flow of automatic software synthesis for SDF modeled streaming applications based on given application graph and target many-core model. The block RACE is our contribution to this flow discussed in this Chapter.

optimization algorithm which transforms the given generated C code into an optimized version in the same native programming language (C code). The optimization will redeem some of the performance loss caused by synthesizing software for a platform with a few number of processors from a large data flow graph. Figure 5.1 is the modified version of Figure 2.3 and demonstrates the place of RACE optimization within the other algorithmic steps in the flow of the software synthesis process for SDF modeled streaming applications.

5.1. Motivating Example

Figure 5.2 demonstrates a toy application modeled by a SDF graph. In this Section, we show the types of optimizations discussed in this Chapter using this example. In Figure 5.2.a, edges of the graph represent inter-task (actor) data dependency. They are annotated with the names and production/consumption rates that implement inter-task dependency in the synthesized software. The name of each task is also provided in a black label adjacent to the task.

Figure 5.2.b represents a single appearance schedule of the tasks in the graph. The number next to each task in the schedule represents the repetition factor of the task. This repetition factor ensures that the required input data tokens to the connected tasks are produced, and the data tokens ready on the input buffers of the running task are consumed before moving on to executing the next task in the schedule.

Figure 5.2.c demonstrates the C code generated by the software synthesis process. The parts in gray are generated directly from the code inside each task, and the for-loops wrapped around them provide the required repetition by the given schedule. Note that tasks B and C facilitate parallel processes if multiple processors are available. In this example, they are both run sequentially on a single core processor along with other tasks.

Figure 5.2.d shows the paths on which two data elements of the source array travel to arrive at their final destinations (before their value changes). Lastly, some of the final destinations of the source data elements are shown in Figure 5.2.e. As it can be seen in Figure 5.2, if arrays P and Q are reconstructed directly from M, then array N and all of its associated assignments can be removed from the code to optimize its performance.

5.2. Problem Statement

The problem we are tackling in this Chapter is to reduce the amount of memory access caused by intensive data transfer, reorder, duplicate, drop (ex. down sampling) which are commonly transpired in signal processing and streaming applications. The problem in its general form is to difficult to address. However, as it was discussed in Section 5.1, SDF modeled applications possess characteristics that can be used to solve the problem. To this end, we assume the following criteria in the target applications:

1- The location of the data tokens within the communication buffers is not coupled with the values of these tokens, i.e. the indices of the array variables are statically resolved in the code.

2- Each iteration of the application is representative of other iterations in terms of the location of the data tokens within the communication buffers. This rule ensures that analyzing an application for only one iteration guarantees the solidity of the optimization in other iterations.

In Section 5.6 we illustrate the practicality of these assumptions in the real-life streaming applications.

5.3. Memory Access Modeling

We first explain the problem in the scalar form to introduce some of the concepts and the terminology used in the remainder of this Chapter. Please note that this problem in its

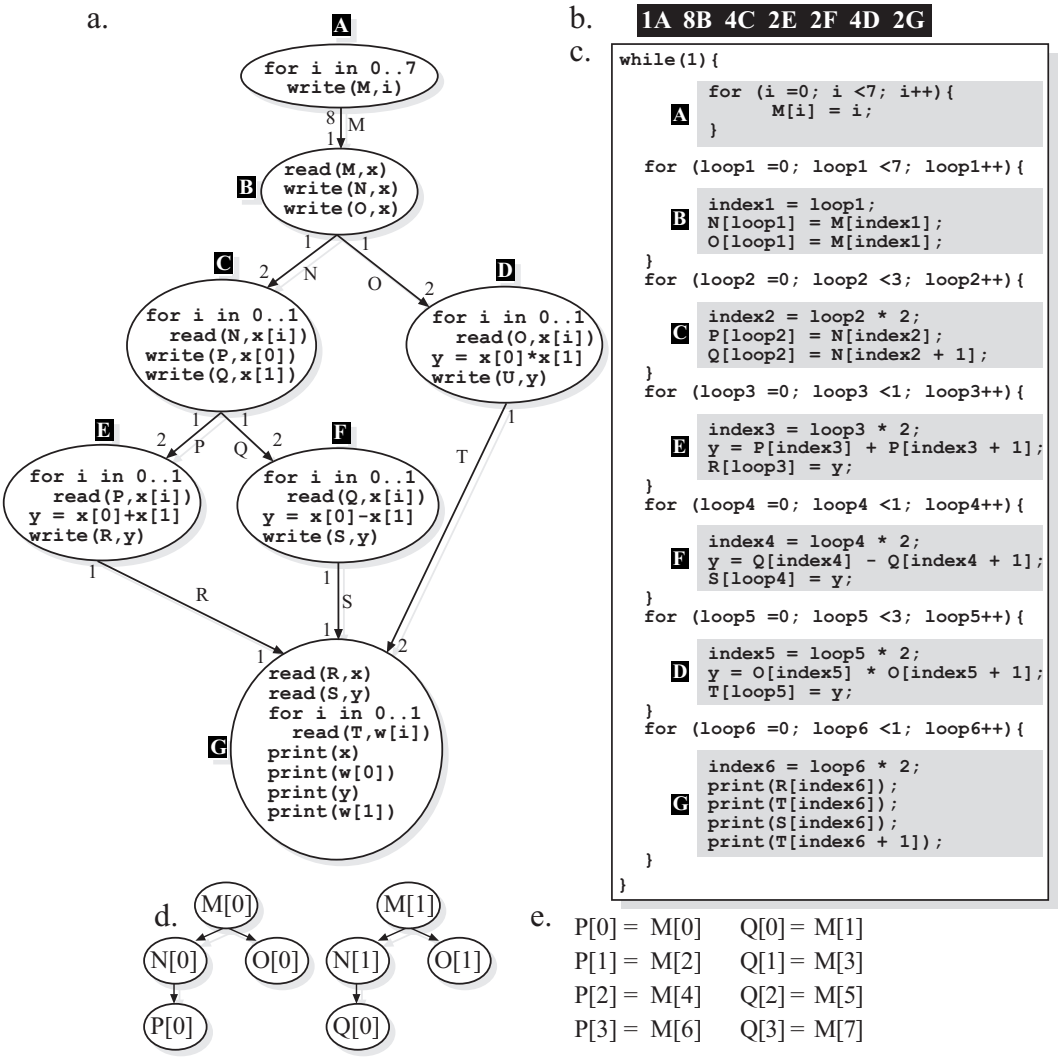


FIGURE 5.2. a. A toy example SDF graph. The edges of the graph are annotated with the names and production/consumption rates that implement inter-task dependency in the synthesized software. The name of each task is also provided in a black label adjacent to the task. b. A single appearance schedule of the tasks in the graph. c. The synthesized code generated from the given SDF graph under the given SA schedule targeting a uniprocessor platform. The snippet codes in gray are directly reflected from the tasks of the SDF graph, and the loops implement the required repetition factor for each task. d. Two examples paths that data elements take to arrive at their final destinations. d. The mapping between the source array and two destination arrays.

scalar form may appear to be unnecessary since the conventional compilers are equipped with better ways to solve it, either through register allocation or other known techniques. However, the introduction of the scalar from is just a way to ease the readers to follow the main concepts on the more complex form of the problem introduced in the second half of

this Section where the array form is discussed. The real life applications and the solution introduced in this Chapter are within the array form.

5.3.1. Scalar Variables. We first define Run Time Dependency Graph (*RTDG*). In addition to data dependency information between different variables in the program the RTDG also contains run time information. This information is gathered by following the control flow from a high-level execution of the program under the assumption that the control flow of the code, e.g. loop variables or the condition in conditional instructions, does not depend on any of the input variables. The majority of SDF modeled applications follow this assumption since this programming model is most effective when the control of the program only relies on the availability of the input data and not the conditional structure of the code.

DEFINITION 5.1. *The Run Time Dependency Graph (RTDG) is a graph with a set of vertices and two sets of edges and is defined as follows:*

$$G = < V, E, M >$$

$$V = \{v : a^{\alpha} \mid a \in \ VS, \ \alpha \in \mathbb{Z}\}$$

$$E = \{e : a^{\alpha} \rightarrow b^{\beta} \mid a, b, ... \in VS, \ b := f(a, ...)\}$$

$$M = \{m : a^{\alpha} \rightarrow a^{\alpha+1} \mid a \in VS\}$$

VS is the set of all variables in the code and α is the α^{th} time the variable a has changed value during execution of the program and is called *change count* of variable a. We use the notation ":=" to refer to the assignment operation in high-level languages, thus "$b := f(a, ...)$" means the value of $f(a, ...)$ which is a mathematical expression that relies on variable a and possibly other variables, is assigned to variable b. E is the set of dependency edges between variables, and M is the set of alteration edges. Any time that an assignment instruction is reached, a new instance of the variable accepting the new value is created as a node in the RTDG. The change count of the new instance is incremented by 1 from the last instance of the same variable or assigned zero if this is first time the assignment is reached. The new generated node is also connected to the last instance with an alteration

edge. Figure 5.3 shows the RTDG for the code shown on the right hand side of the picture. The code is assumed to have before and after pieces.

One assignment in the code can create a series of nodes in the RTDG if the assignment exists in a loop and is visited multiple times during the execution of the program.

Each node $a^\alpha \in V$ of the graph G also stores the O_{a^α} and R_{a^α} information as described below:

DEFINITION 5.2. *R_{a^α}: The assignment instruction in the code that changes a for the α^{th} time. Note that multiple values of α might have the same R_{a^α} if the assignment instruction is iteratively executed in a loop.*

DEFINITION 5.3. *O_{a^α}: The Instruction Order which is a number defining a global order on the instructions. We use the notion of instruction order as the n^{th} instruction in the representative execution on an abstract single issue architecture. Any time that a changes, the associated instruction order of the corresponding instruction at that instance is kept in O_{a^α}.*

In the above definition of the RTDG, each node in graph G represents a memory transaction not differentiating between registers and main memory. The objective of the problem is to minimize $|V|$ by eliminating some of the assignment instructions from the code without changing its functionality.

DEFINITION 5.4. *Parents and Children sets of a node in the graph G: The set of nodes that directly contribute to the new value of the variable instance a^α in an assignment instruction is called the "Parent Set" of a^α ($PS(a^\alpha)$). The "Children Set" of a^α ($CS(a^\alpha)$) is the set of nodes whose values are directly influenced by the node a^α in assignment instructions.*

In Figure 5.3, $PS(b^\beta) = \{a^\alpha\}$ and $CS(b^\beta) = \{c^\gamma, c^{\gamma+1}, c^{\gamma+2}\}$.

DEFINITION 5.5. *We define simple and complex assignments as follows:*
An assignment instruction with only one variable on the right hand side is referred to as a simple assignment. The corresponding execution of a simple assignment to a^α in RTDG will have $|PS(a^\alpha)| = 1$. The assignment is complex otherwise.

In Figure 5.3 "$b := a$" is a simple assignment and "$c := f(b, x)$" is a complex assignment.

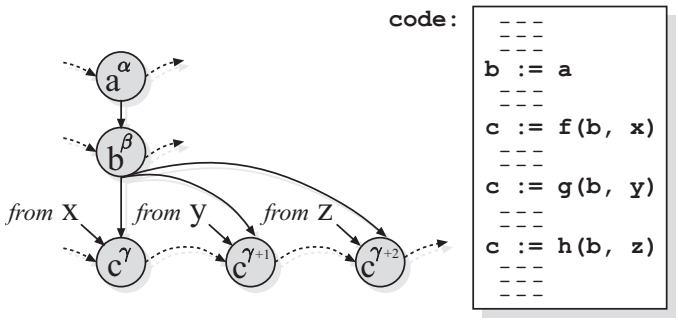


FIGURE 5.3. Example of the Run Time Dependency Graph (RTDG) created from the given code. Solid arrows represent the dependency edges and dashed arrows demonstrate the alteration edges. Since the code is assumed to have before and after pieces, some of the arrows do not show start or stop points.

In general, the term "copy" is only used when simple assignments are in discussion as one variable is copied over another. Complex assignments suggest a form of computation in the code. Therefore we only target simple assignments for the redundant memory access elimination process in this work. In Figure 5.3, assume that the simple assignment "$b := a$" is the target instruction for elimination. The intuition is that we could use a in any instruction that is using the content of variable b from this point on in the code until b receives a new value. In this example, b can be replaced by a in "$c := f(b, x)$" instruction.

LEMMA 5.1. *If the instruction R_2 ($c := f(b, x)$) is executed after the instruction R_1 ($b := a$) in the code, the variable b can be replaced by the variable a in R_2 if the following conditions are met. α, β, and γ are assumed to be known change counts for each variable.*

$\forall \gamma \leq \gamma_{max}$: *(γ_{max} represents the last time c changes in instruction R_2)*

I. $PS(b^{\beta}) = \{a^{\alpha}\}$ *R_1 is a simple assignment*

II. $O_{a^{\alpha}} < O_{c^{\gamma}}$ *R_1 is executed before R_2*

III. $O_{a^{\alpha+1}} > O_{c^{\gamma}}$ *a doesn't change before R_2 is executed*

In the example presented in Figure 5.3, b^{β} is the only child of a^{α}, thus, the assignment is simple (condition I in Lemma 5.1). The instruction order of a^{α} is also less than that in c^{γ}, $c^{\gamma+1}$, and $c^{\gamma+2}$ where c is being assigned new values by f, g, and h functions respectively (condition II in Lemma 5.1). Lastly, under the assumption that a is not changed during the time that variable b is being used, we can also state that $O_{a^{\alpha+1}}$ which represents the next

time a is changed (not shown in the code) is bigger than O_{c^γ}, $O_{c^{\gamma+1}}$, and $O_{c^{\gamma+2}}$ (condition III in Lemma 5.1). Therefore, in this example b can be replaced by a and the target assignment "$b := a$" can be eliminated.

The change count values α, β, and γ are used in this example as an indication that we do not have any prior knowledge about previous assignments, in which, the values of the variables a, b, and c might have changed.

DEFINITION 5.6. *Chain Assignment: When a series of nodes in the RTDG are connected from one to another by only one dependency edge on each connection (simple assignments), they are collectively called a chain assignment. The first node and the last node of a chain assignment are denoted the source and destination of the chain, respectively.*

Chain assignments represent a series of redundant operations that can be optimized. We can eliminate the intermediate assignments in a chain if we could verify that the three conditions in Lemma 5.1 are applicable between the source and the destination of the chain. The rules I and II in Lemma 5.1 are embedded in the given definition of a chain assignment, thus, rule III in Lemma 5.1 is the deciding condition.

LEMMA 5.2. *If $a^\alpha \rightarrow b^\beta \rightarrow ... \rightarrow p^\rho$ is a chain assignment, then the variable p can be replaced by the variable a in the next instruction(s) consuming p if the following condition is met (ρ_{max} represents the last time p changes in the corresponding instruction in the chain):*

$$\forall \rho \leq \rho_{max} :$$

$$O_{a^{\alpha+1}} > O_{p^\rho}$$

Figure 5.4 depicts an example of a chain assignment in the RTDG. For practical reasons, we only use the term *chain assignment* for the sequences that comply with the Lemma 5.2.

DEFINITION 5.7. *Landing instruction: The instruction in which the destination variable of a chain assignment is consumed is called the landing instruction of the chain.*

In the chain assignment shown in Figure 5.4, we can replace d with a in the landing instruction "$e := f(d, x)$" in the code which consequently creates the opportunity for the other intermediate assignments to become candidates for elimination if their contribution

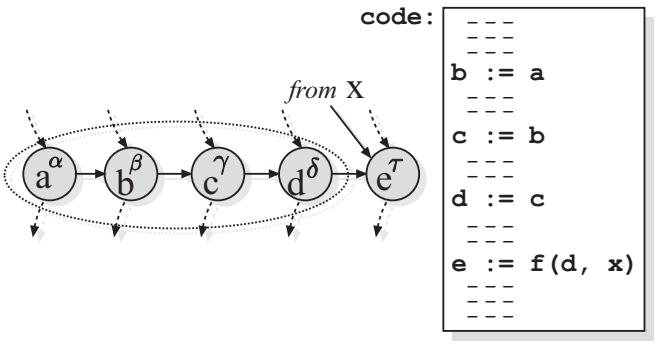


FIGURE 5.4. Example of a Chain Assignment created form the given code. The part within the enclosed dashed area represent a chain in accordance to Lemma 5.2, and the node e represents the Landing Instruction of the chain.

to the code ends here. Note that replacing the variables in the landing instruction doesn't necessarily permit us to remove the intermediate assignments. In fact, we need to make sure that every instruction influenced by an intermediate assignment is being considered, and only if the intermediate assignment becomes completely redundant then it can be removed from the code. In Section 5.4 we discuss how to determine if an assignment is no longer relevant in the code as part of the RACE algorithm.

DEFINITION 5.8. *Depth: The number of intermediate nodes in between the source and the destination of a chain is called depth of the chain.*

The depth of a chain is an upper bound on the number of assignments that can be removed from the code for this chain. In Figure 5.4, the depth of the given chain is four.

5.3.2. Array Variables. In the array form of the problem, each element of an array is represented in the RTDG as if it is a scalar variable. However, another dimension is added to the graph to maintain the stretch of the array along its index dimensions. The discussion in this work focuses only on one dimensional arrays. However, arrays with more dimensions can be easily handled by either transforming the multi dimensional array into a one dimensional array or by adding more dimensions to the RTDG. Figure 5.5 illustrates the index dimension in the graph.

The chain assignments are recognized on an element by element basis. In other words, each element of an array is followed individually throughout the code regardless of its association with other elements of the array. Each chain assignment connects the source of the chain to the destination. The collection of all chain assignments starting with one array

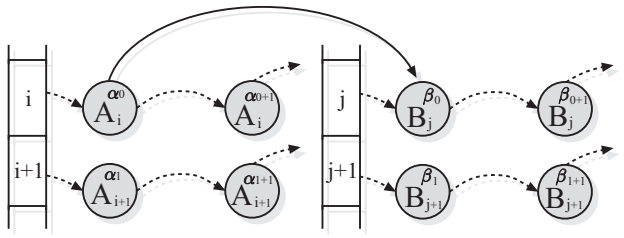


FIGURE 5.5. Example of the RTDG in the array form.

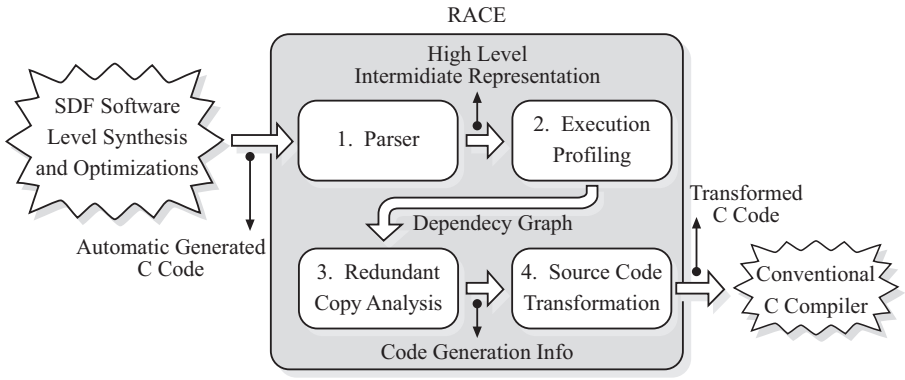


FIGURE 5.6. RACE block diagram.

with all its different elements (indexes) reveal the mapping between the elements of the source array and the destination array. Figure 5.2.e shows an example of such mappings. This mapping helps us to replace the destination array with the source array in the landing instruction after the intermediate assignments are eliminated.

More discussion on finding the mappings and replacing the final arrays in the landing assignments is given in the next Section (RACE Algorithm) where we propose our solution to this problem.

5.4. RACE Algorithm

RACE is a source-to-source optimization algorithm which means the input to the algorithm is a high level source code and the output is the optimized version of the same code. In this Section every time we mention code or program the source code to the RACE algorithm is meant, and the word algorithm is reserved for the RACE algorithm.

RACE consists of four main modules depicted in Figure 5.6. The main input and output of each module are also shown in the Figure. In the remainder of this Section we describe each module in more detail.

5.4.1. Parser. The function of the parser used in RACE is to simply collect enough information from the input code for the execution profiling unit to be able to determine the index values of the arrays during the execution time, differentiate between alternate types of assignments, and finally disclose the relationship between the indexing of an array and the loop structures of the code. A high level intermediate representation of the code is generated in the parser and is sent to the execution profiling module where this information can be used to evaluate the code for further analysis.

5.4.2. Execution Profiling. In this module, the entire code is evaluated without having access to the input data (dynamic information). The type and number of input and output data, however, is known as they are pre-defined in SDF models and are translated into the type and size of the data arrays in the code. In each step of the execution, the high level state of the program is stored. This information consists of the current Instruction Order (analogous to program counter), the value of the index variables if they are known at the time, and also the next instruction to run.

Definition 5.9. *The Known and Unknown instructions: As instruction is known if the values of the variables involved in the instruction are known and can be statically resolved at the time of execution profiling. An instruction is unknown otherwise.*

The known assignments are usually associated with the index variables in which a known numeric value is assigned to a variable. The unknown assignments are generally affiliated with the input data, hence, their values are unknown to the algorithm when the code is statically analyzed. When such an assignment is encountered, a node in the RTDG is generated and connected to the proper node(s) in the graph based on the other variable(s) on the right hand side of the assignment. Please note that all index variables in the current assignment should have known values, otherwise that particular application is out of the scope of the RACE optimization.

Using the information extracted in this module, the RTDG is generated for the entire variable set of the program and sent to the next module.

Definition 5.10. *Node Table: For practical reasons, we also create a set for each assignment consisting of every node of the RTDG associated with that assignment. This set is then kept in a hash table called Node Table using the related assignment as the key.*

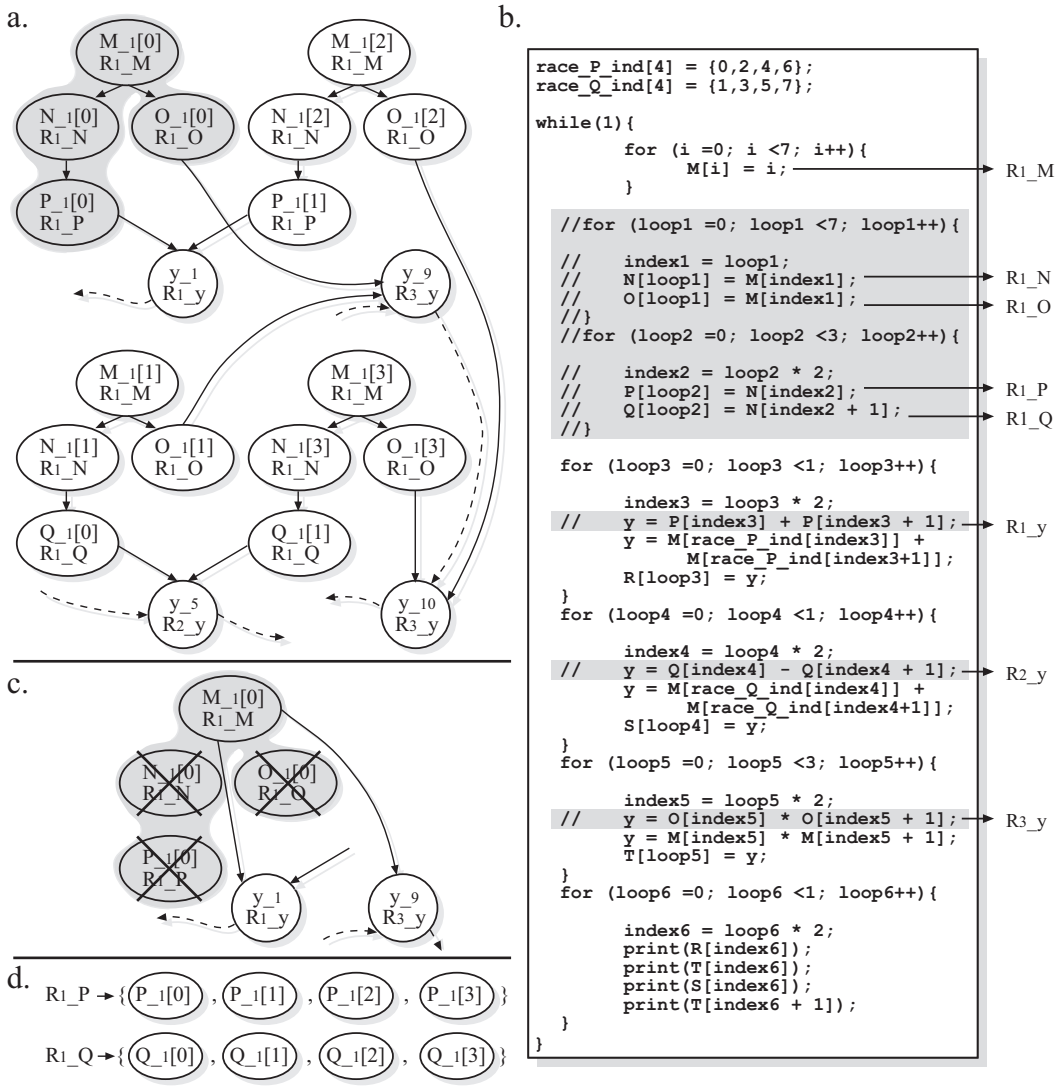


FIGURE 5.7. a. A partial RTDG corresponding to the code illustrated in Figure 5.2.c. The instruction responsible for changing the value of the variable associated with a node is also shown in the node. The part in gray represents an example of a chain assignment. b. The optimized version of the code shown in Figure 5.2.c after applying the RACE optimization. Some of the instructions are named to add clarity between the RTDG and the code. The parts in gray are the eliminated assignments. c. The process of eliminating intermediate nodes from the RTDG in a chain assignment. d. Two entries of the Node Table associated with the given RTDG.

The Node Table is also sent to the next module along with the RTDG.

Figure 5.7.a demonstrates a small portion of the RTDG associated with the C code shown in Figure 5.2.c (Section 5.1). The code was generated from the toy SDF graph shown in Figure 5.2.a under the given schedule targeting a single uniprocessor. The solid edges represent the variable dependencies and the dotted edges denote the alterations of

each variable. The numbers next to each variable in each node of the RTDG represents the change count of that variable. None of the array variables change more than once in this application, however, the variable y changes frequently throughout the execution of the application, thus the alteration edges are only shown for this variable.

In each node, the instruction responsible for changing the value of the variable associated with that node is also given. as it was discussed in Section 5.3, the nodes of the RTDG not only represent the variables but the assignments in which the values of the variables change as well. Two entries of the Node Table associated with this RTDG are also given in Figure 5.7.d.

5.4.3. Redundant Memory Access Analysis. In this module, the redundant memory accesses are recognized. The first step towards this recognition is to identify the chain assignments explained in Section 5.3. Starting from any occurrence of a variable, the dependency graph can be recursively traversed forward from parents to children to get to the last node(s) of the chain. As it is shown in Figure 5.4, the last node in a chain is the one whose children do not represent a simple assignment. Note that since an array can be copied over multiple arrays a chain can be in the form of a tree with multiple destinations, each of which can be subject to optimization. In the RTDG given in Figure 5.7.a, the gray area represents a chain tree with multiple branches. The node at the end of each branch has multiple dependencies, thus representing a complex assignment. Algorithm 3 shows how the RTDG is recursively traversed to extract chains that start from a given variable.

Moreover, as it was discussed in Section 5.3 the source and the destination of the chain must satisfy the condition in Lemma 5.2. The test is checking if the Instruction Order of the source is smaller than that of all destinations of the chain.

The recursive procedure allows us to start from one element of an array and discover the settling destination of the value of this element before it is engaged in any calculation in the program (consumed in the landing instruction). The number of intermediate assignments that this value goes through before getting to the destination (the depth of the chain) is, in fact, the number of redundant copies we can potentially eliminate from the code. If this procedure is run on all elements of an array, we will have the complete mapping between the source and the destination arrays.

Algorithm 3 chainIdentification

Input: $DG, chain, parent$
{/* DG : dependency graph , s: the source of the chain, $chain$: the data structure to keep the results, $parent$: the start node for the traverse forward */}
if $parent.childrenSet = \phi$ **then**
return
end if
$chain.add(parent)${/* the current node is added to the chain */}
for $child$ in $parent.childrenSet$ **do**
if $child.type = simple$ **then**
$chainIdentification(DG, s, chain, child)$
else
if $parent.order < s.nextAlter.order$ **then**
$parent \leftarrow end\ point${/* mark parent as end point */}
{/* $order$ is the instruction order and $s.nextAlter$ is the next instance when the value of s changes following the alteration edges */}
end if
end if
end for

Note that different elements of an array can potentially cross different paths in the code and can be part of different chains with different depths. We define the depth of a mapping to be the maximum depth of all chains involved in the mapping. All array variables used in the code can be sorted by the depth of their mappings. Intuitively, we would want to start the redundant copy elimination process from the array with the maximum depth. It is because the chains with the depth larger than two contain chains with smaller depths depending on which node is selected as the source. Therefore, when the larger chain is considered first the partial chains existing within the large chain would be processed only once.

Depending on the nesting loop structure of the code, this mapping can have different patterns. In principle, there are only two major patterns that can occur between the source and the destination arrays. Figure 5.8 illustrates these two patterns. In Figure 5.8.a, the source array is read and copied, element by element, onto another array with the same size (or more than one array) or multiple times onto different elements of another array(s) with a larger size.

In Figure 5.8.b, the size of the source array is larger than the size of the destination array. Therefore, in each iteration one part of the source array is copied onto the destination array (or partially copied if the destination array has another source), and in the next iteration

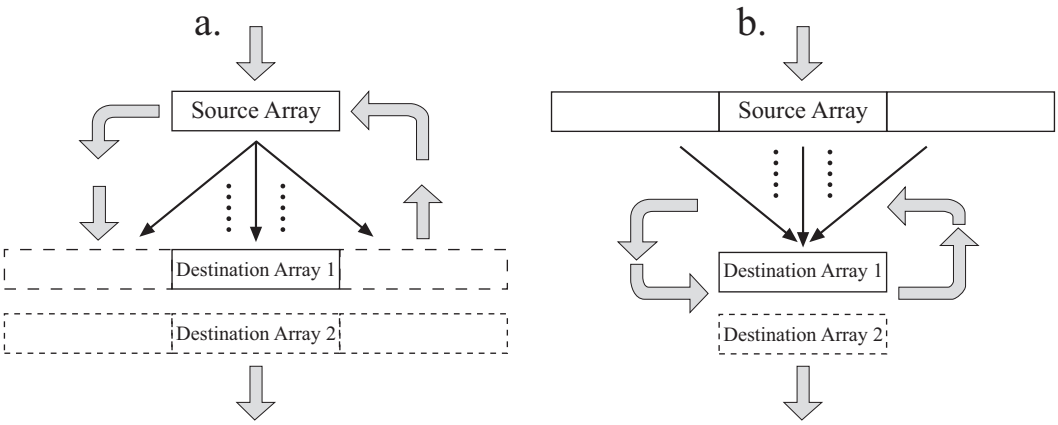


FIGURE 5.8. Two major patterns between the source and the destination arrays. a. The elements of the source array are rearranged in one or more destination arrays. One element can be copied in different locations in the destination arrays, and the entire process can be repeated in a loop. b. The elements of the source array are partially copied and rearranged in one or more destination arrays. In each iteration of the inner nesting loop, different groups of elements are copied in the destination arrays with the same arraignment as that in the first iteration.

another part of the source is copied over the previous values of the destination. In this pattern, the mapping between the first part in the source array and the destination array is repeated in other iterations as well. The only difference is, in each iteration the indexes of the elements read from the source array is increased by the size of the destination array.

Note that in both of these mapping patterns, there is only one source array. It is not because there will never be a situation in which two arrays are copied in one destination. However, in this methodology we only consider one source at a time. When there are two or more sources for one destination, each part of this transaction is considered separately.

After uncovering the mapping of a source array and also the type of its pattern, the next step would be to start the elimination process by updating the RTDG. To this end, for every chain with the current source array, all intermediate nodes in the chain are removed from the graph, and the source and the destination become directly connected. These nodes are also removed from the Node Table described in the previous Subsection. Whenever an entry (a set of nodes associated with an assignment) in this table becomes empty, the related assignment can be removed from the code. In Figure 5.7.c, an example for the node elimination process is demonstrated. Any node removed from the RTDG will be removed from its associated entry in the Node Table as well.

5.4.4. Code Transformation. It is still necessary to insert the proper assignments connecting the source and the destination arrays to complete the code. The code transformation module is responsible for removing/generating irrelevant/required code lines. The code generation info shown in Figure 5.6 consists of any pair of source and destination arrays that form chain assignments, the mapping between the pairs, and irrelevant code lines which needs to be removed from the code. The irrelevant code lines are the ones whose entry in the Node Table have become empty during the node elimination process meaning that the assignment will not have an observable effect on program behavior. For example in Figure 5.7.c, the assignments $R1\_N$, $R1\_O$, $R1\_P$, AND $R1\_Q$ become irrelevant after the node elimination process.

The irrelevant code lines are simply deleted (or commented out) from the original code. When these code lines are removed, there might remain other code lines which become irrelevant as well, such as the instructions in charge of the indexing of the deleted array assignments or some of the loop structures. We leave this part to the conventional compilers (e.g. gcc) as they are equipped with algorithms for dead code elimination [**DEMDS00**].

To generate the necessary code lines, we first discuss mappings of pattern type *a.* in Figure 5.8. Assume that the mapping between the source and the destination arrays is recognized as follows: $B[0] = A[i_0]$, $B[1] = A[i_1]$, ..., $B[n] = A[i_n]$ where A and B are the source and the destination arrays, respectively. In general, we can always replace the the destination array with the source array in the landing instruction as it is shown in Figure 5.9.a.

DEFINITION 5.11. *Mapping Array: A constant array in the size of the destination array, which discloses the relationship between the source and the destination arrays. In other words, mapping array is essentially a lookup table for the indices of the two arrays.*

indA in Figure 5.9 is an example of a mapping array. The preferred location for the mapping array would be outside of the main loop of the program. The C code shown in Figure 5.7.b is the optimized version of the the original code given in Figure 5.2.c after performing the RACE optimization. The mapping patterns in this example are all of the type a. in Figure 5.8. The first two lines of the code instantiate the mapping arrays between

a.

```
1:    indA = {i0, i1, i2, ..., in};
      \* C[j] = f(B[k + const]); -->    the old instruction *\
2:    C[j] = f(A[indA[k + const]]);    \* the new auto generated instruction *\
```

b.

```
1:    indA = {i0, i1, i2, ..., in};
2:    jump = -offset;

...

3:    loop{
4:        jump = jump + offset;
          \* C[j] = f(B[k + const]); --> the old instruction *\
5:        C[j] = f(A[indA[k + const + jump]]); \* the new auto generated instruction *\
      }
```

FIGURE 5.9. Examples of the optimized codes associated with the patterns a. and b. shown in Figure 5.8, respectively.

the array M and the arrays P and Q in the code. The landing instructions in which the arrays P and Q are used ($R1\_y$ and $R2\_y$) are changed accordingly.

In the case of mappings of pattern type *b.* the same principle is applied. However, since the source array is larger than the destination array and periodically copied onto the destination with some offset in each iteration, the generated code needs to be adjusted as it is shown in Figure 5.9.b.

As it was discussed in Section 5.3, *offset* is usually equal to the size of the destination array. The first two lines of the code shown in Figure 5.9.b should be located outside of the main loop of the program.

5.5. Extensions to RACE

In this Section, we explore two more optimizations which can be performed in addition to the main RACE algorithm to further improve the quality of the generated final code. These optimizations use the same information produced in the parser and execution profiling modules but apply them differently in the copy elimination and code transformation modules.

5.5.1. Reverse Application. Previously discussed in Section 5.3, the source and the destination of a chain must satisfy the condition in Lemma 5.2, which guarantees that the source is not changed before it is used in the landing instruction. Algorithm 3 allows us to start from one element of an array and discover the settling destination of the value of this element by traversing the RTDG in the forward direction. This direction also complies with

the actual flow code. However, after applying RACE on the code in this direction, there remains a possible case in which traversing the graph in the reverse direction can uncover additional copy elimination opportunities.

Figure 5.10.b shows a snipped of code copied (and changed to suit as an example) from the FFT application, which is a benchmark application presented in Section 5.6. Figure 5.10.d illustrates a small portion of the RTDG associated with the given code. As it is shown in the Figure, the value of $results[0]$ is copied to $N[0]$ but changes before it can be used in the landing instruction. It violates the condition in Lemma 5.2, therefore this copy will not form a chain in the RTDG.

However, if the RTDG is traversed in the reverse direction, and the the condition in Lemma 5.2 is applied in the reverse order, more chains can be revealed in the RTDG. An example chain formed in this order is shown in Figure 5.10.e. If the RTDG is traversed in reverse for all the elements of array N, the mappings between the source and the destination arrays are revealed.

In the code transformation module, the same principle described in 5.4.4 applies, only this time the destination array replaces the source array in the initiation instruction (the instruction in which the source array is last changed in a complex assignment). Figure 5.10.c demonstrates the optimized code after applying the reverse RACE optimization.

Please note that if the forward RACE optimization is applied first on the code, the reverse RACE optimization will only find chains with the depth of one as the intermediate copies are captured in the forward RACE. Figure 5.10.a depicts the general case in which the reverse RACE optimization is applicable. In this scenario, the source array is generated within a loop, and then copied over to a larger array. This process continues until the larger array receives the generated data in the right order which will be used later in the code.

5.5.2. Mapping Patterns. Inserting the mapping array ($indA$) in the code as it is shown in the example codes in Figure 5.9 is not always necessary. There might be some mappings with known patterns between the indices of the source and the destination arrays that can be reflected on the landing instruction without using the mapping array. Some of the simple possibilities would be if the source array is copied onto the destination array with the exact same indexing order, or if only the odd or even indices are copied. For

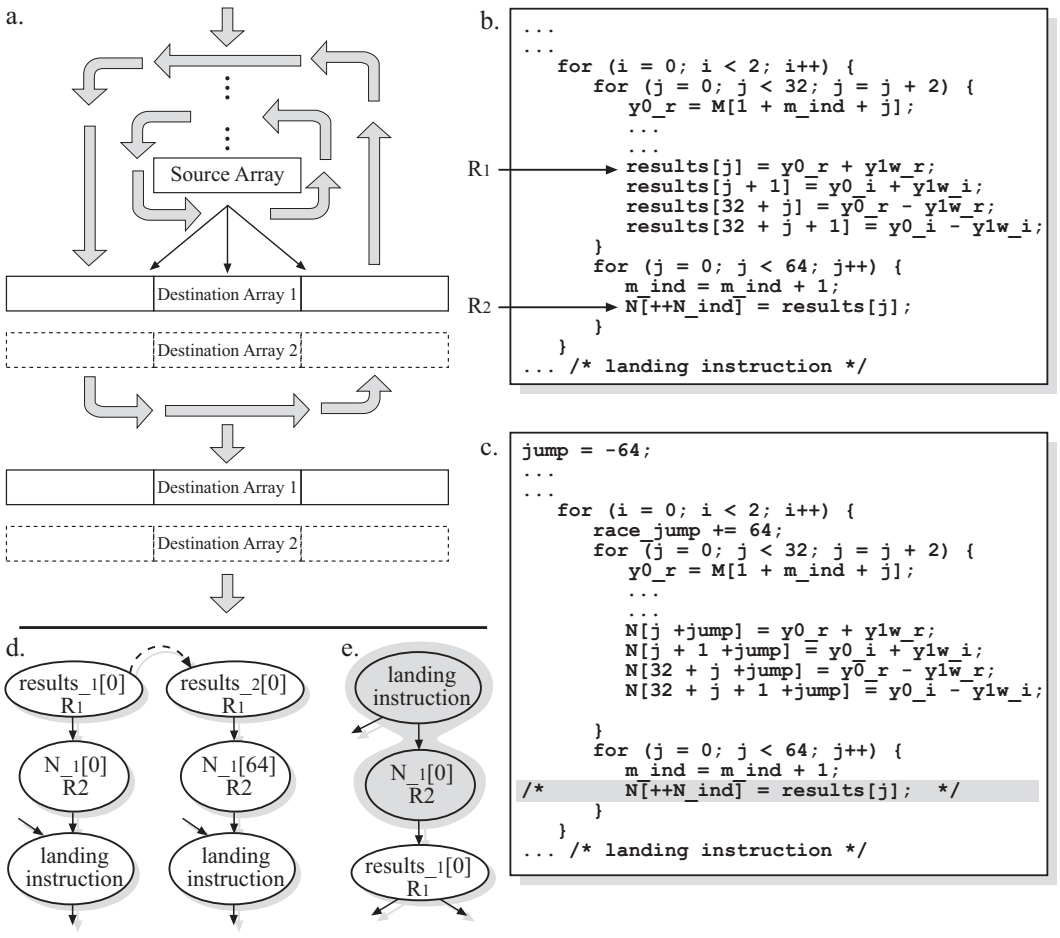


FIGURE 5.10. a. the general case in which the reverse RACE optimization is applicable. b. a simplified sippet code from the FFT2 application, which is a benchmark application presented in Section 5.6. The landing instruction is assumed to appear later in the code and is not shown here. c. The optimized version of the code after applying the reverse RACE optimization. d. A partial RTDG associated with the given code. e. An example chain formed in reverse order.

example in Figure 5.7.b, the arrays M and O have exactly the same indexing order, thus the reconstruction of the landing instruction ($R3\_y$) is simply done by replacing array O with array M.

The mapping arrays will also cause the overhead of increasing the code size of the application especially when the associated source and destination arrays are large. If the mapping between the source and destination arrays can be formulated in a function, we could use this function to regenerate the proper indices required to reconstruct the landing instruction. Assume *mapping_addr* (*int addr*) is a mapping function which accepts an index of the destination array B and calculates its mapping to the source array A. In this case,

a. Main

```
...
    \* C[j] = f(B[k + const]); --> the old instruction *\
    C[j] = f(A[mapping_addr(k + const)]);\* the new auto generated instruction *\
...
```

b. Matrix Multiplier

```
1:    int mapping_addr( int in_index) {
2:        int out_index;
3:        int j;
4:        out_index = in_index & 1;
5:        for (j = 1; j < 7; j++) if (in_index & (1 << j))
              out_index = (out_index + (1 << (7 - j)));
6:        return out_index
7:    }
```

c. FFT

```
1:    int mapping_addr( int in_index) {
2:        return ((in_index % 2 == 0) ? (10*(in_index/200) + (in_index % 20)/2)
           : (100 + (in_index % 200)/20 + 10 * (in_index % 20)/2));
3:    }
```

FIGURE 5.11. a. An example of the optimized code when a mapping function is used. b. The manually implemented mapping function for the Matrix Multiplier application. c. The manually implemented mapping function for the FFT2 application.

the landing instruction $f(B[k+const])$ can be reconstructed as it is shown in the main code given in Figure 5.11.a.

In general, finding a pattern between two sets of numbers that are regenerated inside one or a series of nested for-loops is a complex problem and is out of the scope of this dissertation. However, to show the level of difficulty arising from this problem when dealing with real life applications, we manually created the mapping functions for two of the benchmark applications used in our evaluation in Section 5.6. The mapping functions for "Matrix Multiplier" and "FFT" applications are shown in Figures 5.11.b and 5.11.c.

In Section 5.6 we further discuss the potential performance gain/loss of applying this approach using the manually calculated functions given in Figures 5.11 regardless of the origin of these functions (manually or automatically obtained). This discussion will determine if future work on the automatic realization of the mapping functions is justified.

5.6. Experimental Evaluation

In this Section, we present our experimental results to demonstrate the effectiveness of RACE in further optimizing the auto-generated executable codes outputted from MIT

StreamIt compiler [**GTK$^+$02**]. Although we use StreamIt as an example of existing academic software synthesis tools for evaluation purposes, our proposed technique is not conceptually specific to StreamIt implementation.

5.6.1. Setup. The benchmark applications presented in this Section are implemented in StreamIt language [**GTK$^+$02**] which conforms to the SDF semantics, by modeling an application as a graph of interconnected but independent "filters" with statically-defined input and output rates. The StreamIt compiler translates stream programs to C, which can be passed to any standard C compiler to generate executable binaries. In this work, before the generated C codes are sent to a standard compiler, they are processed once again in the RACE algorithm for further optimizations.

RACE is a source-to-source optimization algorithm which transforms the given generated C code into the optimized version in the same native programming language (C code).This version of RACE only supports the basic syntax of the C programming language such as basic data types, arrays, loops, variable assignments, prints, etc. This decision was made primarily because the more complex syntax of the language do not appear in automatically generated codes, such as in the StreamIt benchmark applications we use for evaluation.

We report the results from Nios II processor running the original auto-generated benchmark codes and the ones after RACE optimization. Nios II is a 32-bit RISC architecture designed specifically for the Altera family of FPGAs [**Alt15**].

The original and the optimized codes were also compiled and executed on a Unix machine to ensure that functional correctness is preserved after our transformation.

We create six different hardware settings by generating three Nios II processors shown in Table 5.1, each of which is accompanied by a separate Floating Point Unit (FPU) hardware that is enabled in only half of the settings. These settings are selected to comply with commonly used general purpose embedded processors.

Figure 5.12 depicts the flow of execution and the tool chain used in our experiments on the benchmark applications from programming in StreamIt language to running the final executable binary codes on the FPGA platform. RACE is our only contribution to this flow.

TABLE 5.1. **The specifications of the Nios II processors**

	name	branch prediction	HW mult/dev	instruction cache	data cache
Proc. I	Nios II/e	-	-	-	-
Proc. II	Nios II/f	X	X	16KB	-
Proc. III	Nios II/f	X	X	16KB	16KB

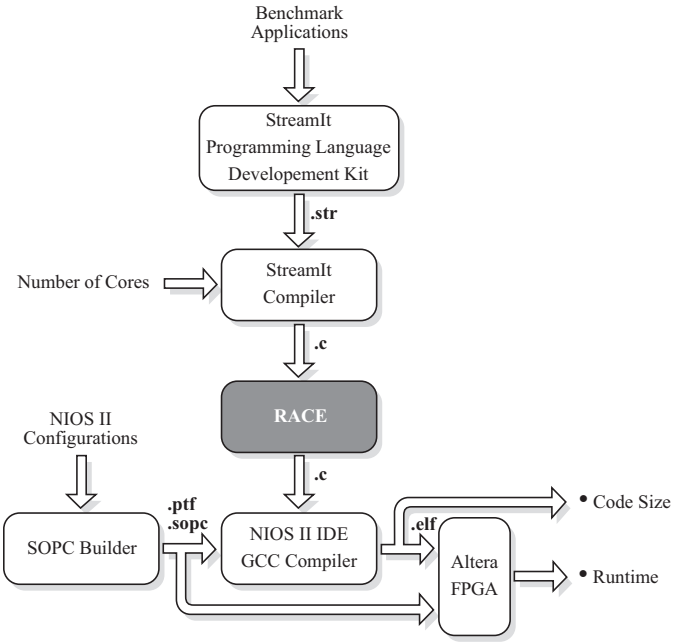


FIGURE 5.12. The flow of execution and the tool chain used to run the experiments in this Section.

5.6.1.1. *Benchmark Applications.* To evaluate the proposed technique we selected four different streaming kernels as our benchmarks. They include matrix multiplication, the fast Fourier transform (FFT), time delay estimation (TDE), and a ten-stage lattice filter. These kernels frequently appear in many higher-level applications that are used in portable and handheld embedded systems. Table 5.2 shows the benchmark applications along with some of the specifications of the original C code and executable binaries outputted form StreamIt compiler.

5.6.2. Results. Table 5.2 also reports the results on the benchmark applications after the RACE optimization. The runtime numbers given in this Table are reported from running the original codes and the codes after the RACE optimization is applied on the given hardware settings. The original and after RACE codes are compiled using GCC version

TABLE 5.2. **The specifications of the benchmark applications before and after RACE optimization**

			Matrix Multiplier		FFT2		TDE_PP		Lattice	
			original	after RACE	original	after RACE	original	after RACE	original	after RACE
total inst. order			17063	8863	10924	9644	1033216	892096	6418	6414
total num. of array elements			2700	200	3828	1788	85112	61848	2044	2042
code size (kB)	without FPU	proc. I	679	689	712	709	833	829	686	686
		proc. II	676	686	709	706	830	826	682	682
		proc. III	676	686	710	707	830	826	683	683
	with FPU	proc. I	678	688	708	705	827	823	683	683
		proc. II	675	685	705	702	824	820	680	680
		proc. III	676	685	706	703	825	821	681	681
run time cycles (k)	without FPU	proc. I	22451	16667	14423	13531	3411060	3232712	8760	8757
		proc. II	3143	1438	2448	2198	298999	263913	1286	1286
		proc. III	878	512	689	642	122049	107569	377	371
	with FPU	proc. I	7449	1662	3540	2692	643871	463251	1681	1681
		proc. II	2158	434	1065	812	117786	85341	504	504
		proc. III	503	132	254	207	44697	33814	123	123
maximum chain depth			3		5		9		1	
num. of src.-dst. pairs			1		2		4		1	
largest src. array			200		256		6480		2	
largest dst. array			2000		128		256		2	
RACE runtime (sec)			6		4		193		2	

4.5.3 included in the Altera Quarts 14.0 tool set. The reported runtime numbers are for one iteration of the applications averaged from 10 iteration runs.

Three of the applications in the benchmark set show substantial improvements after RACE optimization is applied. The last application (Lattice) does not provide us with any significant improvement. In fact, it is worth mentioning that horizontally paralleled applications in which the data is copied and spread among multiple actors are better suited

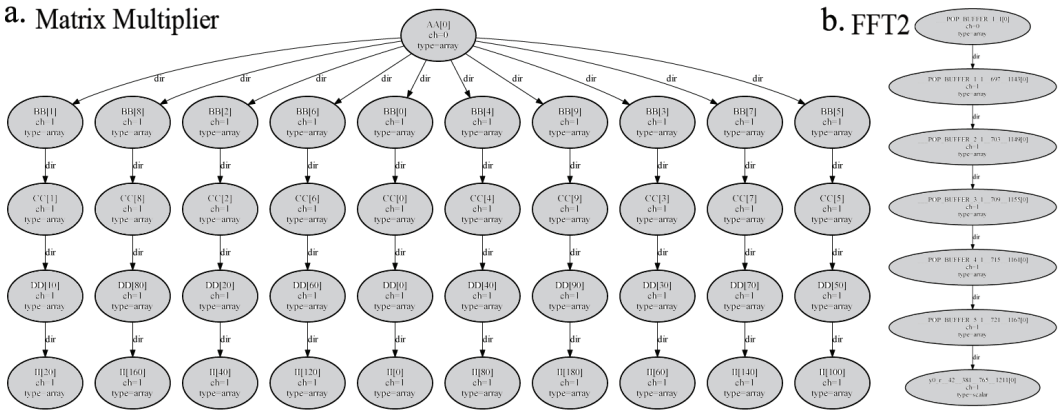


FIGURE 5.13. Sample chains for two of the applications.

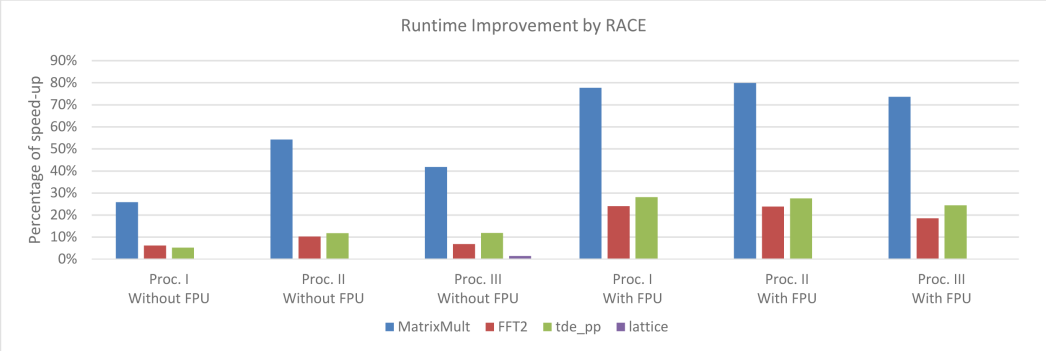
for RACE optimization in contrast to vertically paralleled applications where a series of actors work in form of a pipeline.

In the bottom part of the Table 5.2 we also report some of the parameters from the RACE algorithm in order to compare different factors which contribute to the effectiveness of RACE optimization. For example, although in Matrix Multiplier there is only one instance of a source-destination pair with the maximum chain depth of three, a large number of chain assignments are involved in this instance. It is because the source array is copied multiple times into the destination array using the intermediate assignments. Therefore, a large number of assignments are eliminated in the elimination process. Figure 5.13 depicts a sample chain for one of the elements in the source array for Matrix Multiplier application along with a sample chain for FFT2 application.

Figure 5.14.a demonstrates the speed-up made by RACE on the runtime of running the application on the given hardware settings over the original benchmark codes for the benchmark applications. As shown in this Figure, Matrix Multiplier achieves maximum of 80% improvement on runtime after applying RACE optimization.

The effectiveness of RACE is expressed differently in different hardware settings. Figure 5.14.a shows a major increase in the speed-up made by RACE between the processors when the FPU accelerator is enabled as opposed to when it is disabled. It is because in the absence of the FPU accelerator, the floating point computations are done in software, hence the ratio of memory operations and non-memory operations is changed. Since the RACE algorithm decreases the number of memory operations by eliminating redundant memory accesses, less improvement is observed on the systems without a FPU.

a.



b.

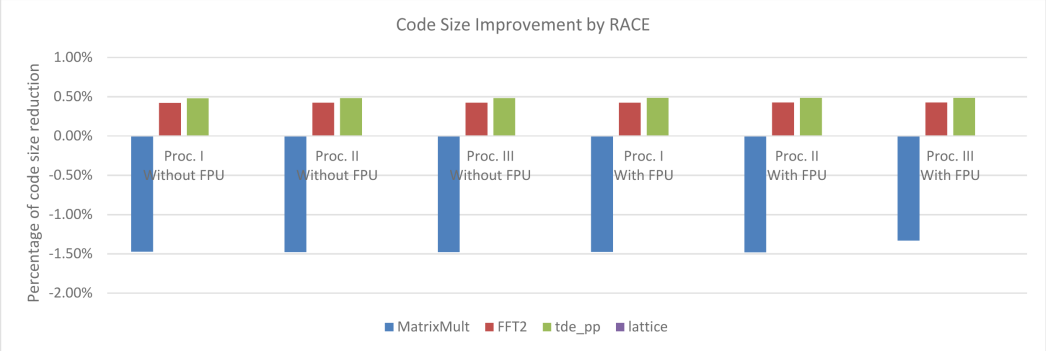


FIGURE 5.14. The percentage of improvements made by RACE over the original codes (before RACE) in both runtime and code size reduction.

The same effect is noted when cache memory is introduced to the system. The difference between processor II and processor III is that the former only has instruction cache and the latter has both instruction and data caches. In the presence of data cache memory operations impose less delay, and consequently RACE becomes less effective compared to the system without data cache.

Figure 5.14.b demonstrates the effect of the RACE optimization on the size of the final compiled code. Introducing mapping arrays to the code (discussed in Section 5.4.4) increases the code size. On the other hand, eliminating the irrelevant array assignments results in code size reduction. Therefore, the combination of these additions and eliminations determines the final code size of the application. For the Matrix Multiplier application, the RACE optimization results in an increase in the size of the code memory. The RACE optimization has a negligible impact on the code size of the other two applications.

TABLE 5.3. **Reverse Application**: The results of adapting Reverse Application (RA) in RACE for the FFT2 and TDE_PP applications. The results for the RACE main approach are also given for comparison.

			FFT2		TDE_PP	
			RACE/main	RACE/RA	RACE/main	RACE/RA
run time cycles (k)	without FPU	proc. I	13531	12694	3232712	3105552
		proc. II	2198	2006	263913	255781
		proc. III	642	598	107569	100047
	with FPU	proc. I	2692	1945	463251	350584
		proc. II	812	621	85341	66151
		proc. III	207	167	33814	27062

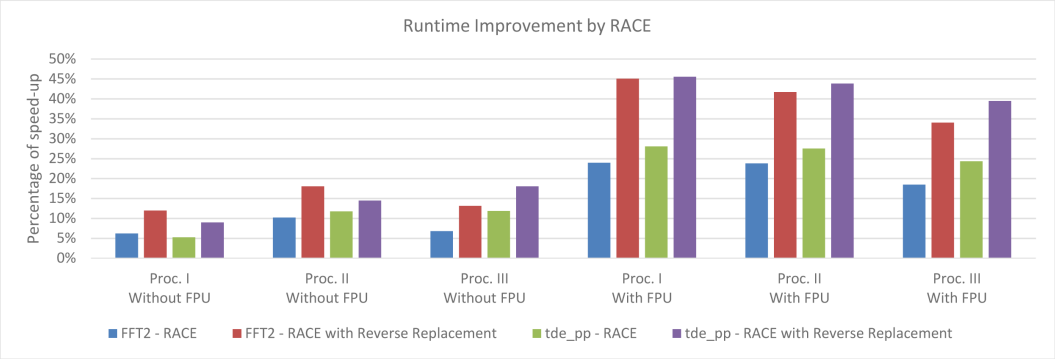


FIGURE 5.15. The percentage of improvements made by reverse replacement optimization over the original codes (before RACE) in runtime reduction. The same results on the RACE main approach is also given for comparison.

5.6.3. Results on the Extensions. In Section 5.5 tow more optimizations in addition to the main RACE algorithm are explored. In this part of this Section we show the impact of these additional optimizations on the same benchmark applications.

5.6.3.1. *Results on Reverse Application.* The opportunity to apply the reverse application optimization (described in Section 5.5.1) appears only in FFT2 and TDE_PP applications. Table 5.3 shows the results of applying this optimization in addition to the RACE main approach (forward) for the two applications on the given hardware settings. The results for the RACE main approach without the additional optimization are also given for comparison.

Figure 5.15 demonstrates the improvements made by the reverse application optimization compared to the original codes. For comparison, the improvements made by the RACE main approach are also depicted in the Figure. The impact of reverse application on the code size is negligible, and thus, not shown in the results.

TABLE 5.4. **Mapping Formulations**: The results of adapting mapping formulations in RACE in both function and macro implementations for the Matrix Multiplier and FFT2 applications. The mapping formulations are manually generated and are shown in Figure 5.11.

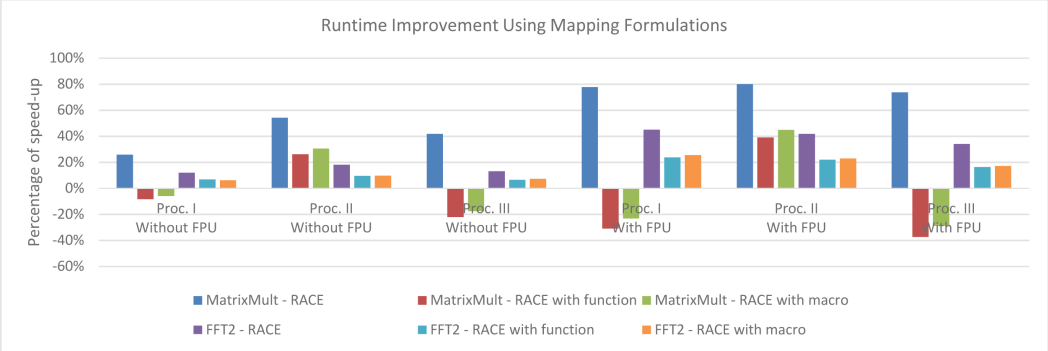
			Matrix Multiplier		FFT2	
			RACE/function	RACE/macro	RACE/function	RACE/macro
code size (kB)	without FPU	proc. I	689	692	708	709
		proc. II	686	689	705	706
		proc. III	686	689	706	706
	with FPU	proc. I	688	691	704	705
		proc. II	685	688	701	702
		proc. III	685	688	702	702
run time cycles (k)	without FPU	proc. I	24315	23794	13452	13524
		proc. II	2321	2187	2216	2211
		proc. III	1072	1032	689	644
	with FPU	proc. I	9751	9177	2700	2639
		proc. II	1314	1190	831	821
		proc. III	691	648	212	210

5.6.3.2. *Results on Mapping Patterns.* As discussed in Section 5.5.2, it is possible to formulate the mapping between the source and destination arrays in a function. Although automatic realization of such functions is a hard problem on its own, we show the effect of this formulation on the runtime and the code size of the applications by manually creating the mapping functions for two of the benchmark applications used in this Section. The mapping functions for the Matrix Multiplier and FFT2 applications are shown in Figures 5.11.b and 5.11.c, respectively. In order to reduce the overhead of function calls in the code, we also implement the mapping functions as inline macros.

Table 5.4 shows the results of implementing the mapping formulations in the form of both functions and macros for the Matrix Multiplier and FFT2 applications on the given hardware settings.

Because the mapping between the source and the destination arrays is usually the result of complex reordering, duplicating, or dropping the input data in streaming applications, mapping formulations tend to be complex as well. Consequently, employing mapping formulations predominantly increases the amount of computations in the code. On the other hand, removing the mapping arrays from the code can possibly reduce the size of the final

a.



b.

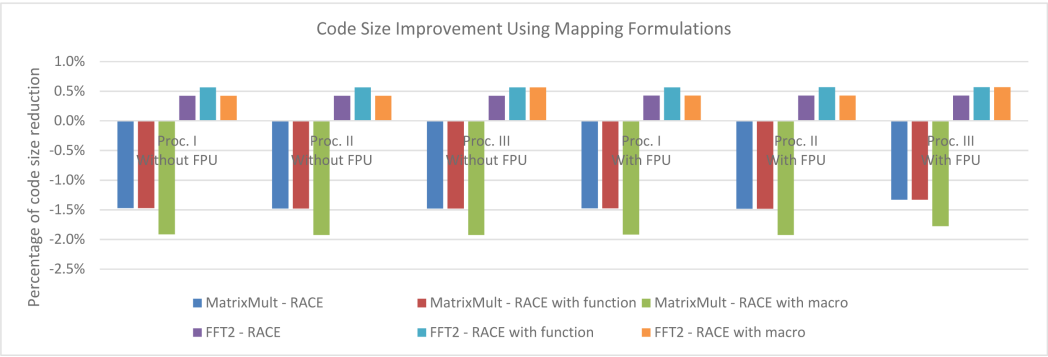


FIGURE 5.16. The effect of employing mapping formulations instead of mapping arrays on the runtime and code size of the applications in both function and macro implementations. The original RACE implementation (using mapping arrays) is also given for comparison.

compiled code if the inserted functions are smaller than the required mapping arrays in code size.

Figure 5.16.a illustrates the effect of employing mapping formulations instead of mapping arrays on the runtime of the applications in both function and macro implementations. Figure 5.16.b demonstrate the same effect on the code size of the applications.

As it is depicted in Figures 5.16.a and 5.16.b, employing mapping formulations reduces or in some cases reverses the effectiveness of the RACE optimization with almost no improvement gained on the size of the final compiled code.

CHAPTER 6

Related Work

Suitability of synchronous dataflow (SDF) graphs for modeling and analysis of streaming applications has been long recognized. There has been ample research on modeling, optimization and synthesis of digital signal processing systems using SDF graphs. Lee and Messerschmitt presented one of the earliest results in this area, by providing necessary conditions for static schedulability of SDFs [**LM87b, LM87a**].

There are a number of frameworks that support modeling, analysis and experimentation with SDF-based specifications. Simulink [**sim**] and LabVIEW [**lab**] are prime examples of commercial tools that support development of such designs. In the academic world, the Ptolemy project is perhaps one of the better known frameworks for modeling and experimentation with various models of computations, including SDF [**EJL+03**]. StreamIt [**GTK+02**] is an academically-developed SDF-based language and compiler that supports development of streaming software.

Efficient management of system resources is especially important in the embedded application space, where execution platforms are often resource-constrained. The required memory for storage of application data and instructions is an important resource, due to its impact on system cost and power dissipation, which is especially important for embedded platforms. As a result, many researchers have focused on reducing the memory requirement of streaming software applications.

Compiler-level approaches to streaming buffer optimization can be broadly divided into scheduling-oriented and allocation-based techniques. Scheduling-oriented techniques consider the impact of task scheduling on instruction memory [**BLM96**], total buffer size [**MBL97, ODH05**], and overall code size [**KTA03**]; while allocation-based schemes are typically applied after scheduling [**MB01, MB04, FHHG10**].

The vast majority of SDF-based code generation schemes utilize function inlining, which eliminates the function call overhead and the need for stack maintenance. In such settings,

the application instruction size can be greatly reduced by resorting to "single-appearance (SA) schedule" [**BLM96**], which requires tasks to appear exactly once in the periodic schedule. In [**MBL97**], the authors present a dynamic programming-based algorithm for generating the single-appearance schedule that minimizes total buffer memory for a chain SDF. Furthermore, this work is heuristically extended to handle non-chain SDF graphs [**BLM96**]. In [**ZTB00**] the trade-off between buffer memory usage and compile time is explored. The work combines the algorithm reported in [**MBL97**] with an evolutionary optimization algorithm to further explore the search space. Authors of [**KTA03**] show that single-appearance schedules are not necessarily optimal, when code size and buffer memory are collectively taken into consideration, and presents a technique called "phase scheduling" to address the holistic problem. An approach with similar objective is presented in [**KMB07**], where optimized non-single appearance schedules are systematically constructed to balance the memory cost of buffers and code size. The authors utilize function calls instead of function inlining, and offer techniques to control the number of function calls in the synthesized code.

Allocation-based schemes attempt to reduce the required memory size by smart allocation of buffers in the memory space. For example, Murthy and Bhattacharyya exploit buffers' lifetime to allow safe sharing of memory space among buffers that are not alive at the same time [**MB01**]. This work is later extended to merge chains of buffers, when additional information on the number of input and output tokens that are simultaneously alive, is available [**MB04**]. The authors also introduced the notion of "consume before produce" to formalize the aforementioned information [**BM04**].

In Chapter 3, we proposed an allocation-based scheme and demonstrated the benefits of analyzing buffers' spatiotemporal behavior at the finest possible analysis granularity for both single-appearance and non-single appearance schedules. We also introduce a technique to explore tradeoffs between optimization complexity and total memory footprint.

The aforementioned memory optimization techniques only aim at minimizing the memory footprint of the system and do not discuss the number of memory accesses in the final executable codes. In fact, Prior work from embedded systems community deals with analysis at the level of SDF graphs, and stops after software implementation is synthesized. Optimization of the produced code is conventionally considered to be within the purview of

standard compilers, and out of the scope of SDF synthesis. However, the subject problem of Chapter 5 and the proposed solution are specific to software that is synthesized from SDFs, as the ability to statically analyze the behavior is central to both detection and optimization of redundant array accesses. Since SDF models have not been of particular interest in conventional programming languages and compilers communinities, the problem has not received any attention from compiler researchers. As it was demonstrate in Section 5.6, state of the art compilers (e.g., gcc) are unable to handle array optimization of the type developed in Chapter 5.

On the other hand, the rise of System on Chip (SoC) architectures in recent years has identified interconnection methodologies as an important aspect of system design and optimization. Various forms of Network on Chip (NoC) design techniques are introduced to improve communication on SoC platforms [**BJM+05, DRGR03, MNTJ04**]. The mapping between virtual and physical resources, which affects application throughput, energy consumption, and quality of service, is one of the challenges of implementing an application on a NoC-based platform.

Marcon et al. [**MCM+05**] introduced CDCM (Communication Dependence and Computation Model) to capture the characteristics of application computation. They proposed a simulated annealing algorithm to solve the mapping problem for mesh based NoC architectures, targeting both throughput and power consumption of the system. Ascia et al. [**ACP04**] proposed a multi-objective task mapping approach using genetic algorithms with similar optimization criteria as [**MCM+05**] to explore the search space.

Instead of direct performance or energy metrics, some researchers have focused on maximum bandwidth required by the system as the optimization objective. For example, Murali et al. employ traffic splitting as a technique to reduce the required bandwidth on links of the network [**MDM04**]. They propose a three phase algorithm called NMAP. In the first phase, a new mapping is initialized. In the second phase, minimum path computations are performed, and finally the initial solution is iteratively improved by repeating the second phase for pair-wise swapping of vertices. They revisit the problem in [**MBdM05**] with an emphasis on Quality of Service (QoS) in the final mapping solution.

Hu et al. [**HM05**] propose a runtime-aware technique using a branch and bound algorithm, which constructs a mapping solution with a deadlock-free deterministic routing

function such that the total communication energy is minimized. Srinivasan et al. [**SC05**] proposed a technique called MOCA, which utilizes the principles used in [**MDM04**] with a focus on the energy consumption of the system. Tosun et al. [**TOO09**] formulate the mapping problem using Integer Linear Programming (ILP), and leverage the best solutions found within tolerable solver time to obtain the optimal or high quality mapping solutions. Tosun [**Tos11**] later proposed another technique called CastNet, which takes advantage of the symmetry in mesh architecture to improve both energy consumption and algorithm runtime compared to NMAP [**MDM04**] and MOCA [**SC05**] algorithms.

The aforementioned mapping approaches target similar packet-based NoC as the underlying interconnect architecture. Thus, they do not readily address circuit-switched interconnection of processors. To underscore difrences between these two platforms, we implement one of the recent algorithms (CastNet [**Tos11**]) discussed in this Section and report its results on the benchmark applications given the constraints of circuit-switched architectures. The reported results (Section 4.5.2) highlight the need for a new approach to solve the mapping problem for GALS platforms. Circuit-switched GALS architectures have shown promising results in improving the performance and power consumption of SoC platforms [**MVK$^+$99, OMCM07**]. The comparison between the two approaches to interconnect network design is out of the scope of this dissertation, however, one can find such a comparison in [**CSC06**].

In Chapter 4 of this dissertation, we introduce a constructive mapping algorithm for circuit-switched GALS NoC architectures called BAMSE (BAlanced Mapping Space Exploration). The goal of this algorithm is to minimize the "maximum communication distance" in the mapped application. As it is discussed in Section 4.1, this optimization improves the overall performance of the mapped application. Minimizing the total communication distances is our secondary objective, as it translates to reduced system energy dissipation.

In Chapter 4, we also bring practical considerations, such as usecase scenarios, core failures and fixed functions, into the mapping context. We explore the trade-off between solution quality and the tool run time via parameter configuration. Furthermore, we statistically analyze the problem of parameter configuration, and outline development of a configuration layer on top of the basic algorithm for arriving at a solution with acceptable quality. BAMSE offers a unique way of exploring the search space in different directions,

and allows the designers to adjust the optimization time budget depending on the situation. This makes the mapping tool suitable for both online mappings where the run time of the tool is the limiting factor, and offline mappings where the quality of produced mapping has higher priority.

CHAPTER 7

Conclusion and Future Work

In this dissertation, we study different optimization steps in automated software synthesis for streaming applications on embedded manycore platforms. Automated software synthesis significantly reduces the development time. The vision is to enable seamless and efficient transformation from a higher-order specification of the stream application (e.g., dataflow graph) to parallel software code (e.g., multiple `.C` files) for a given target manycore platform. This automated process involves many steps which are being actively researched, including task assignment, task scheduling, buffer allocation, processor mapping, and finally code generation. Chapters /refch:BufferManagement to /refch:race presented our contributions to efficient optimization techniques required for automated software synthesis, a summary of which is the following.

- Streaming kernels and applications are abundant in the embedded systems domain, where underlying hardware platforms have to deal with strict resource constraints. Therefore, it is critical to understand the tradeoffs involved in resource requirement of streaming applications. We contribute to this important goal by developing a framework that captures the tradeoff between buffer footprint and optimization complexity during synthesis of streaming applications from synchronous dataflow specifications. We demonstrate that analysis of buffers' spatiotemporal patterns can be performed at different resolutions. Varying the analysis resolution compromises complexity (runtime) with the quality (buffer size) of the process. Consequently, we transform the buffer allocation problem into packing of complex polygons in the two-dimensional space, and present an evolutionary algorithm that is applicable to different resolution levels. Experimental results demonstrate both the effectiveness of our approach, and the superiority of our technique compared to existing competitors in terms of the memory footprint of the synthesized applications.

- We studied the unique characteristics of processor mapping problem in GALS based CMP platforms. We discussed the unique requirements of GALS platforms, and their implications for task mapping problem. We presented an algorithm called BAMSE, which generates high quality mappings of application task graphs for such platforms. Experiments show that the BAMSE mapping algorithm outperforms the time consuming manual mappings of real life existing applications up to 65% for the longest inter-processor communication link, and up to 19% for total length of the links, when the two criteria are used as primary and secondary optimization objectives, respectively. Furthermore, the reported results from employing one of the previously published mapping techniques specific to packet-switched NoC architectures (CastNet) on the presented benchmark set given the constraints of a packet-switched GALS architecture demonstrate the effectiveness of our approach in solving the mapping problem for GALS platforms. Additionally, BAMSE generates the mappings very fast, and it matches or beats solutions generated by solving ILP instances after 10 days of solver runtime.
- we also studied the automatic generated codes for streaming applications outputted from software synthesis tools. We show that the inherited properties in the code from the abstraction layer allow us to statically analyze the code for further optimizations. We introduce a technique called Redundant Array Copy Elimination (RACE) that potentially minimizes the number of memory instructions in the final code. Experimental results show up to 80% improvement in the runtime of the optimized code in our benchmark application set.

In the light of lesson learned from our previous works, the future direction of this work is the following.

As it is shown in Figure 5.1, automated software synthesis process consists of several key algorithmic steps to generate high quality software for the target hardware platform. Due to the complexity of the problem, the optimization in each step is traditionally made separately. The idea is to generate a high performance code at the end by sufficiently optimizing the problem in each step. However these separate optimizations in different stages of software synthesis do not guarantee an optimized code at the end. In fact, the effect of each step on the others can dramatically change the final performance of the system. There have been

efforts in the literature to make note of these effects and to integrate some of these steps to obtain better results. For example the effect of task scheduling and memory allocation in backend optimization step has been discussed in [**BLM96, MBL97, ODH05, KTA03**].

One of the critical and yet missing steps in this direction is the relationship between task assignment and processor binding. To the best of our knowledge no research has been conducted on this topic. Figures 7.1 and 7.2 show the effect of task assignment on the final mapping for the given merge sort application. In Figure 7.1 the resulting graph from task assignment has the maximum workload of 70 cycles among all cores compare to 100 in Figure 7.2. However the processor binding step shows different mapping results for each task assignment. Both LC (longest Connection) and TC (Total number of Connections) attributes (discussed in Chapter 4) of the final mapping in Figure 7.2 are smaller than those in Figure 7.1. As we discussed in Chapter 4 these differences may translate into different criteria in different architectures. For example the mapping in Figure 7.1 is practically infeasible on AsAP2 processor [**TCM$^+$09**] since the required number of links in each direction between cores exceeds two, which is the limit in this processor. Moreover, longer communications tend to slow down the system in some hardware platforms (ex. AsAP2). Therefore the resulting throughput in one mapping solution may become less than that in another mapping even if the workload estimation in task assignment step predicts differently. A mapping-aware task assignment technique can effectively improve the quality of the final mapping while considering other optimization objectives such as throughput and/or memory requirement of the system.

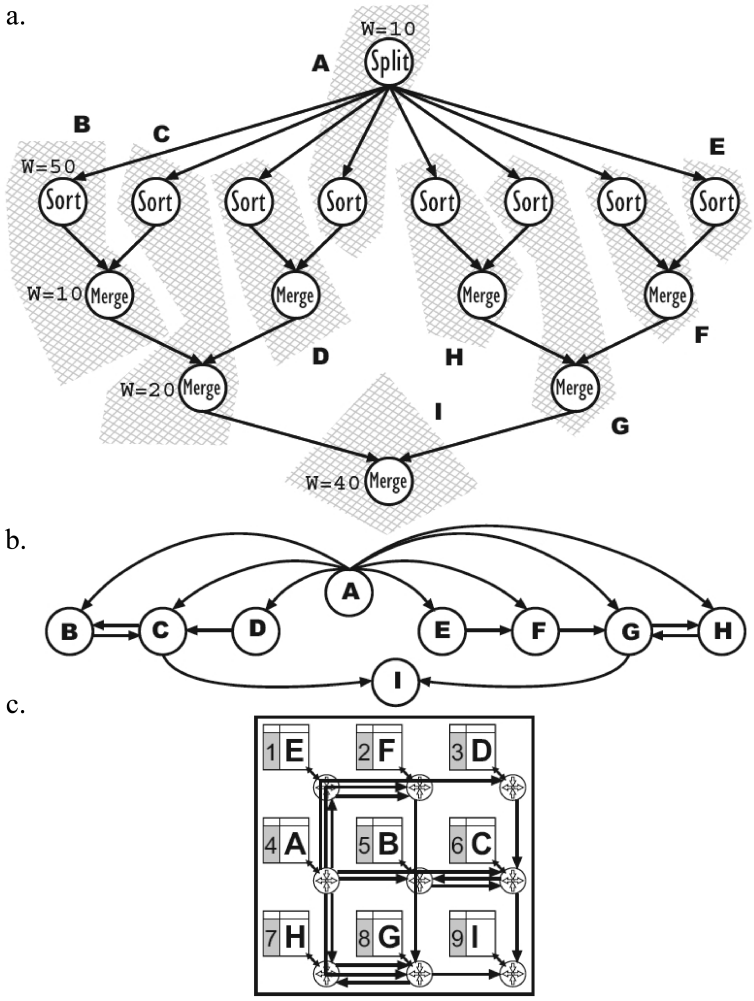


FIGURE 7.1. a. Data flow graph of a merge sort application. The Ws are the estimated workload of each task. The tasks bounded in each hashed area will be assigned to a virtual processor as the result of the task assignment step. The name of each virtual processor is also given next to hashed areas. b. The final graph resulting from the task assignment step. c. The final mapping of the given merge sort application based on the indicated task assignment. The target hardware platform is a 3X3 mesh architecture. The maximum workload among cores is 70 cycles. Processor binding optimally results in the following attributes: Longest connection = 3, Total number of connections = 23, and Maximum required links between processors in each direction = 3.

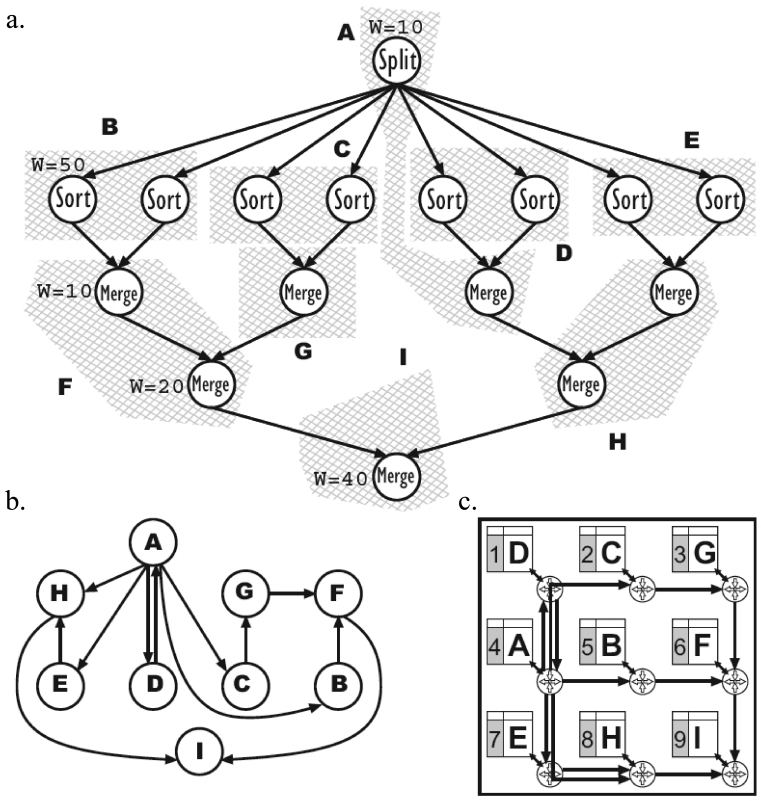


FIGURE 7.2. a. Data flow graph of a merge sort application. The Ws are the estimated workload of each task. The tasks bounded in each hashed area will be assigned to a virtual processor as the result of the task assignment step. The name of each virtual processor is also given next to hashed areas. b. The final graph resulting from the task assignment step. c. The final mapping of the given merge sort application (the same as Figure 7.1 based on the indicated task assignment. The target hardware platform is a 3X3 mesh architecture. The maximum workload among cores is 100 cycles. Processor binding optimally results in the following attributes: Longest connection = 2, Total number of connections = 13, and Maximum required links between processors in each direction = 2.

Bibliography

[ABC+06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al., *The landscape of parallel computing research: A view from berkeley*, Tech. report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ABD+09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, *A view of the parallel computing landscape*, Commun. ACM **52** (2009), 56–67.

[ACP04] G. Ascia, V. Catania, and M. Palesi, *Multi-objective mapping for mesh-based NoC architectures*, Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on, sept. 2004, pp. 182 – 187.

[Alt15] Altera, *Nios ii processor: The world's most versatile embedded processor*, 2015, available online at https://www.altera.com/products/processors/overview.tablet.html.

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, *Cyclo-static data flow*, Acoustics, Speech, and Signal Processing, IEEE International Conference on **5** (1995), 3255–3258.

[BJM+05] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, *NoC synthesis flow for customized domain specific multiprocessor systems-on-chip*, **16** (2005), no. 2, 113 – 129.

[BLM96] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software synthesis from dataflow graphs*, Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[BM04] S. S. Bhattacharyya and P. K. Murthy, *The cbp parameter: A module characterization approach for DSP software optimization*, The Journal of VLSI Signal Processing **38** (2004), 131–146.

[Bor99] S. Borkar, *Design challenges of technology scaling*, Micro, IEEE **19** (1999), no. 4, 23–29.

[Cha84] D. M. Chapiro, *Globally-asynchronous locally-synchronous systems.*, Tech. report, DTIC Document, 1984.

[CHJ07] J. Cong, G. Han, and W. Jiang, *Synthesis of an application-specific soft multiprocessor system*, International Symposium on Field Programmable Gate Arrays, 2007, pp. 99–107.

[CM69] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, Proceedings of the 1969 24th national conference, ACM '69, 1969, pp. 157–172.

[CSB92] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, *Low-power CMOS digital design*, IEICE Transactions on Electronics **75** (1992), no. 4, 371–382.

[CSC06] K.-C. Chang, J.-S. Shen, and T.-F. Chen, *Evaluation and design trade-offs between circuit-switched and packet-switched NoCs for application-specific SoCs*, Proceedings of the 43rd annual Design Automation Conference, DAC '06, 2006, pp. 143–148.

[DEMDS00] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, *Compiler techniques for code compaction*, ACM Trans. Program. Lang. Syst. **22** (2000), no. 2, 378–415.

[DRGR03] J. Dielissen, A. Rdulescu, K. Goossens, and E. Rijpkema, *Concepts and implementation of the philips network-on-chip*, IP-based SoC Design (2003).

[EJL+03] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, *Taming heterogeneity - the ptolemy approach*, Proceedings of the IEEE **91** (2003), no. 1, 127–144.

[FHHG10] M. H. Foroozannejad, M. Hashemi, T. L. Hodges, and S. Ghiasi, *Look into details: the benefits of fine-grain streaming buffer analysis*, Conference on Languages, Compilers and Tools for Embedded Systems, 2010, pp. 27–36.

[FHHG12] M. H. Foroozannejad, T. Hodges, M. Hashemi, and S. Ghiasi, *Postscheduling buffer management trade-offs in streaming software synthesis*, ACM Trans. Des. Autom. Electron. Syst. **17** (2012), no. 3, 1 – 31.

[FHM+14] M. H. Foroozannejad, M. Hashemi, A. Mahini, B. M. Baas, and S. Ghiasi, *Time-scalable mapping for circuit-switched gals chip multiprocessor platforms*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **33** (2014), no. 5, 752–762.

[GB04] M. Geilen and T. Basten, *Reactive process networks*, International Conference on Embedded Software, 2004, pp. 137–146.

[GGS+06] A.-H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. Bekooij, B. Theelen, and M. Mousavi, *Throughput analysis of synchronous data flow graphs*, Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on, 2006, pp. 25–36.

[GTK+02] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. P. Amarasinghe, *A stream compiler for communication-exposed architectures*, International Conference on Architectural Support for Programming Languages and Operating Systems, 2002, pp. 291–303.

[Has11] M. Hashemi, *Automated software synthesis for streaming applications on embedded manycore processors*, Ph.D. thesis, UNIVERSITY OF CALIFORNIA DAVIS, 2011.

[HFGE12] M. Hashemi, M. H. Foroozannejad, S. Ghiasi, and C. Etzel, *Formless: scalable utilization of embedded manycores in streaming applications*, SIGPLAN LCTES **47** (2012), no. 5, 71–78.

[HG10a] M. Hashemi and S. Ghiasi, *Versatile task assignment for heterogeneous soft dual-processor platforms*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **29** (2010), no. 3, 414 –425.

[HG10b] M. Hashemi and S. Ghiasi, *Versatile task assignment for heterogeneous soft dual-processor platforms*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) (2010).

[HM05] J. Hu and R. Marculescu, *Energy- and performance-aware mapping for regular NoC architectures*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **24** (2005), no. 4, 551 – 562.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel, *Generating efficient code from data-flow programs*, Programming Language Implementation and Logic Programming (J. Maluszyski and M. Wirsing, eds.), Lecture Notes in Computer Science, vol. 528, Springer Berlin Heidelberg, 1991, pp. 207–218 (English).

[Jak96] S. Jakobs, *On the genetic algorithms for the packing of polygons*, European Journal of Operational Research **88** (1996), 165–181.

[KMB07] M.-Y. Ko, P. K. Murthy, and S. S. Bhattacharyya, *Beyond single-appearance schedules: Efficient DSP software synthesis using nested procedure calls*, ACM Trans. Embed. Comput. Syst. **6** (2007).

[KTA03] M. Karczmarek, W. Thies, and S. Amarasinghe, *Phased scheduling of stream programs*, Conference on Languages, Compilers and Tools for Embedded Systems, 2003, pp. 103–112.

[lab] *National instruments' labview*, available online at http://www.ni.com/labview/.

[LB11] B. Liu and B. M. Baas, *A high-performance area-efficient AES cipher on a many-core platform*, Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on, IEEE, 2011, pp. 2058–2062.

[LM87a] E. A. Lee and D. G. Messerschmitt, *Static scheduling of synchronous data flow programs for digital signal processing*, IEEE Transactions on Computers **36** (1987), no. 1.

[LM87b] ———, *Synchronous data flow*, Proceedings of the IEEE **75** (1987), no. 9, 1235–1245.

[LMM03] A. Lodi, S. Martello, and M. Monaci, *Two-dimensional packing problems: a survey*, European Journal of Operational Research **141** (2003), no. 2, 241–252.

[MB01] P. K. Murthy and S. S. Bhattacharyya, *Shared buffer implementations of signal processing systems using lifetime analysis techniques*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **20** (2001), no. 2.

[MB04] P. K. Murthy and S. S. Bhattacharyya, *Buffer merging–– powerful technique for reducing memory requirements of synchronous dataflow specifications*, ACM Trans. Des. Autom. Electron. Syst. **9** (2004), 212–237.

[MBdM05] S. Murali, L. Benini, and G. de Micheli, *Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees*, Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific, vol. 1, jan. 2005, pp. 27 – 32 Vol. 1.

[MBL97] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, *Joint minimization of code and data for synchronous dataflow programs*, Formal Methods in System Design **11** (1997), 41–70.

[MCM+05] C. Marcon, N. Calazans, F. Moraes, A. Susin, I. Reis, and F. Hessel, *Exploring NoC mapping strategies: An energy and timing aware technique*, Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, DATE '05, 2005, pp. 502–507.

[MDM04] S. Murali and G. De Micheli, *Bandwidth-constrained mapping of cores onto NoC architectures*, Proceedings of the conference on Design, automation and test in Europe - Volume 2, DATE '04, 2004, pp. 20896–.

[MNTJ04] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, *Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip*, Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, vol. 2, feb. 2004, pp. 890 – 895 Vol.2.

[MVK+99] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner, *Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems*, ASIC/SoC Conference, 1999. Proceedings. Twelfth Annual IEEE International, 1999, pp. 317 –321.

[ODH05] H. Oh, N. D. Dutt, and S. Ha, *Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs*, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 2005, pp. 157–165.

[OMCM07] U. Ogras, R. Marculescu, P. Choudhary, and D. Marculescu, *Voltage-frequency island partitioning for gals-based networks-on-chip*, Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE, june 2007, pp. 110 –115.

[PHLB95] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, *Software synthesis for DSP using ptolemy*, IEEE Transactions on Very Large Scale Integration Systems **9** (1995), no. 1-2, 7–21.

[SC05] K. Srinivasan and K. S. Chatha, *A technique for low energy mapping and routing in network-on-chip architectures*, Proceedings of the 2005 international symposium on Low power electronics and design, ISLPED '05, 2005, pp. 387–392.

[sim] *Mathworks simulink - simulation and model-based design*, available online at http://www.mathworks.com/products/simulink/.

[SK03] R. Szymanek and K. Krzysztof, *Partial task assignment of task graphs under heterogeneous resource constraints*, Proceedings of the 40th Annual Design Automation Conference (New York, NY, USA), DAC '03, ACM, 2003, pp. 244–249.

[Str] *StreamIt website*, available online at http://groups.csail.mit.edu/cag/streamit/.

[TCM+09] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas, *A 167-processor computational platform in 65 nm CMOS*, Solid-State Circuits, IEEE Journal of **44** (2009), no. 4, 1130 –1144.

[TLA03] W. Thies, J. Lin, and S. Amarasinghe, *Partitioning a structured stream graph using dynamic programming*, 5th Workshop Media Streaming Processors, December 2003.

[TOO09] S. Tosun, O. Ozturk, and M. Ozen, *An ilp formulation for application mapping onto network-on-chips*, Application of Information and Communication Technologies, 2009. AICT 2009. International Conference on, oct. 2009, pp. 1 –5.

[Tos11] S. Tosun, *New heuristic algorithms for energy aware application mapping and routing on mesh-based NoCs*, Journal of Systems Architecture **57** (2011), no. 1, 69 – 78.

[TTB08] A. Tran, D. Truong, and B. Baas, *A complete real-time 802.11a baseband receiver implemented on an array of programmable processors*, Signals, Systems and Computers, 2008 42nd Asilomar Conference on, oct. 2008, pp. 165 –170.

[TTB10] A. T. Tran, D. N. Truong, and B. Baas, *A reconfigurable source-synchronous on-chip network for gals many-core platforms*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **29** (2010), no. 6, 897 –910.

[V+07] S. Vangal et al., *An 80-tile 1.28tflops network-on-chip in 65nm CMOS*, ISSCC, 2007, pp. 98–99.

[Wor07] E. W. Work, *Algorithms and software tools for mapping arbitrarily connected tasks onto an asychronous array of simple processors*, M.S. thesis, Office Graduate Studies, University of California, Davis (2007).

[XLB11] Z. Xiao, S. Le, and B. Baas, *A fine-grained parallel implementation of a h. 264/avc encoder on a 167-processor computational platform*, Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on, IEEE, 2011, pp. 2067–2071.

[ZTB00] E. Zitzler, J. Teich, and S. S. Bhattclcharyya, *Evolutionary algorithms for the synthesis of embedded software*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **8** (2000), no. 4, 452–455.