1

INTRODUCTION

The focus of this thesis is the exploration of architectures and design methodologies for application-specific parallel systems for embedded applications in digital signal processing (DSP). The hardware model we consider consists of multiple programmable processors (possibly heterogeneous) and multiple application-specific hardware elements. Such a heterogeneous architecture is found in a number of embedded applications today: cellular radios, image processing boards, music/sound cards, robot control applications, etc. In this thesis we develop systematic techniques aimed at reducing inter-processor communication and synchronization costs in such multiprocessors that are designed to be application-specific. The techniques presented in this thesis apply to DSP algorithms that involve simple control structure; the precise domain of applicability of these techniques will be formally stated shortly.

Applications in signal processing and image processing require large computing power and have real-time performance requirements. The computing engines in such applications tend to be embedded as opposed to general-purpose. Custom VLSI implementations are usually preferred in such high throughput applications. However, custom approaches have the well known problems of long design cycles (the advances in high-level VLSI synthesis notwithstanding) and low flexibility in the final implementation. Programmable solutions are attractive

in both these respects: the programmable core needs to be verified for correctness only once, and design changes can be made late in the design cycle by modifying the software program. Although verifying the embedded software to be run on a programmable part is also a hard problem, in most situations changes late in the design cycle (and indeed even after the system design is completed) are much easier and cheaper to make in the case of software than in the case of hardware.

Special processors are available today that employ an architecture and an instruction set tailored towards signal processing. Such software programmable integrated circuits are called “Digital Signal Processors” (DSP chips or DSPs for short). The special features that these processors employ are discussed by Lee in [Lee88a]. However, a single processor — even DSPs — often cannot deliver the performance requirement of some applications. In these cases, use of multiple processors is an attractive solution, where both the hardware and the software make use of the application-specific nature of the task to be performed.

Over the past few years several companies have been offering boards consisting of multiple DSP chips. More recently, semiconductor companies are offering chips that integrate multiple CPUs on a single die: Texas Instruments (the TMS320C80 multi-DSP), Star Semiconductors (SPROC chip), Adaptive Solutions (CNAPS processor), etc. Multiple processor DSPs are becoming popular because of variety of reasons. First, VLSI technology today enables one to “stamp” 4-5 standard DSPs onto a single die; this trend is only going to continue in the coming years. Such an approach is expected to become increasingly attractive because it reduces the testing time for the increasingly complex VLSI systems of the future. Second, since such a device is programmable, tooling and testing costs of building an ASIC (application-specific integrated circuit) for each different application are saved by using such a device for many different applications, a situation that is going to be increasingly important in the future with up to a tenfold improvement in integration. Third, although there has been reluctance in adopting automatic compilers for embedded DSP processors, such parallel DSP products make the use of automated tools feasible; with a large number of processors per chip, one can

afford to give up some processing power to the inefficiencies in the automatic tools. In addition new techniques are being researched to make the process of automatically mapping a design onto multiple processors more efficient — this thesis is also an attempt in that direction. This situation is analogous to how logic designers have embraced automatic logic synthesis tools in recent years — logic synthesis tools and VLSI technology have improved to the point that the chip area saved by manual design over automated design is not worth the extra design time involved: one can afford to “waste” a few gates, just as one can afford to waste processor cycles to compilation inefficiencies in a multiprocessor DSP.

Finally, there are embedded applications that are becoming increasingly important for which programmability is in fact indispensable; set-top boxes capable of recognizing a variety of audio/video formats and compression standards, multimedia workstations that are required to run a variety of different multimedia software products, programmable audio/video codecs, etc.

The generalization of such a multiprocessor chip is one that has a collection of programmable processors as well as custom hardware on a single chip. Mapping applications onto such an architecture is then a hardware/software codesign problem. The problems of inter-processor communication and synchronization are identical to the homogeneous multiprocessor case. In this thesis when we refer to a “multiprocessor” we will imply a heterogeneous architecture that may be comprised of different types of programmable processors and may include custom hardware elements too. All the techniques we present here apply to such a general system architecture.

Why study application-specific parallel processing in the first place instead of applying the ideas in general purpose parallel systems to the specific application? The reason is that general purpose parallel computation deals with a user-programmable computing device. Computation in embedded applications, however, is usually one-time programmed by the designer of that embedded system (a digital cellular radio handset for example) and is not meant to be programmable by the end user. The computation in embedded systems is specialized (the computa-

tion in a cellular radio handset involves specific DSP functions such as speech compression, channel equalization, modulation, etc.). Furthermore, embedded applications face very different constraints compared to general purpose computation: non-recurring design costs, power consumption, and real-time performance requirements are a few examples. Thus it is important to study techniques that are application-specific, and that make use of the special characteristics of the applications they target, in order to optimize for the particular metrics that are important for that specific application. These techniques adopt a design methodology that tailors the hardware and software implementation to the particular application. Some examples of such embedded computing systems are in robot controllers [Sriv92] and real-time speech recognition systems [Stolz91]; in consumer electronics such as future high-definition televisions sets, compact disk players, electronic music synthesizers and digital audio systems; and in communication systems such as digital cellular phones and base stations, compression systems for video-phones and video-conferencing, etc.

The idea of using multiple processing units to execute one program has been present from the time of the very first electronic computer in the nineteen forties. Parallel computation has since been the topic of active research in computer science. Whereas parallelism within a single processor has been successfully exploited (instruction-level parallelism), the problem of partitioning a single user program onto multiple such processors is yet to be satisfactorily solved. Instruction-level parallelism includes techniques such as pipelining (employed in traditional RISC processors), vectorization, VLIW (very large instruction word), superscalar — these techniques are discussed in detail by Patterson and Hennessy in [Patt90]. Architectures that employ multiple CPUs to achieve task-level parallelism fall into the shared memory, message passing, or dataflow paradigms. The Stanford DASH multiprocessor [Len92] is a shared memory machine whereas the Thinking Machines CM-5 falls into the message passing category. The MIT Monsoon machine [Pap90] is an example of a dataflow architecture.

Although the hardware for the design of such multiple processor machines

— the memory, interconnect network, IO, etc. — has received much attention, software for such machines has not been able to keep up with the hardware development. Efficient partitioning of a general program (written in C say) across a given set of processors arranged in a particular configuration is still an open problem. Detecting parallelism, the overspecified sequencing in popular imperative languages like C, managing overhead due to communication and synchronization between processors, and the requirement of dynamic load balancing for some programs (an added source of overhead) makes the partitioning problem for a general program hard.

If we turn away from general purpose computation to application-specific domains, however, parallelism is easier to identify and exploit. For example, one of the more extensively studied family of such application-specific parallel processors is the systolic array architecture [Kung88][Quin84][Rao85]; this architecture consists of regularly arranged arrays of processors that communicate locally, onto which a certain class of applications, specified in a mathematical form, can be systematically mapped. We discuss systolic arrays further in section 1.3.2.

The necessary elements in the study of application-specific computer architectures are: 1) a clearly defined set of problems that can be solved using the particular application-specific approach, 2) a formal mechanism for specification of these applications, and 3) a systematic approach for designing hardware from such a specification.

In this thesis, the applications we focus on are those that can be described by **Synchronous Dataflow Graphs** (**SDF**) [Lee87] and its extensions; we will discuss this model in detail shortly. SDF in its pure form can only represent applications that have no decision making at the task level. Extensions of SDF (such as the *Boolean dataflow* (BDF) model [Lee91][Buck93]) allow control constructs, so that data-dependent control flow can be expressed in such models. These models are significantly more powerful in terms of expressivity, but they give up some of the useful analytical properties that the SDF model has. For instance, Buck shows that it is possible to simulate any Turing machine in the BDF model [Buck93]. The

BDF model can therefore compute all Turing computable functions, whereas this is not possible in the case of the SDF model. We discuss the Boolean dataflow model further in Chapter 6.

In exchange for the limited expressivity of an SDF representation, we can efficiently check conditions such as whether a given SDF graph deadlocks, and whether it can be implemented using a finite amount of memory. No such general procedures can be devised for checking the corresponding conditions (deadlock behaviour and bounded memory usage) for a computation model that can simulate any given Turing machine. This is because the problems of determining if any given Turing machine halts (the halting problem), and determining whether it will use less than a given amount of memory (or tape) are *undecidable* [Lew81]; that is, no general algorithm exists to solve these problems in finite time.

In this thesis we will first focus on techniques that apply to SDF applications, and we will propose extensions to these techniques for applications that can be specified essentially as SDF, but augmented with a limited number of control constructs (and hence fall into the BDF model). SDF has proven to be a useful model for representing a significant class of DSP algorithms; several DSP tools have been designed based on the SDF and closely related models. Examples of commercial tools based on SDF are the Signal Processing Worksystem (SPW), developed by Comdisco Systems (now the Alta group of Cadence Design Systems) [Pow92][Barr91]; and COSSAP, developed by Cadis in collaboration with Meyr's group at Aachen University [Ritz92]. Tools developed at various universities that use SDF and related models include Ptolemy [Pin95a], the Warp compiler [Prin92], DESCARTES [Ritz92], GRAPE [Lauw90], and the Graph Compiler [Veig90].

The SDF model is popular because it has certain analytical properties that are useful in practice; we will discuss these properties and how they arise in the following section. The property most relevant for this thesis is that it is possible to effectively exploit parallelism in an algorithm specified in SDF by scheduling computations in the SDF graph onto multiple processors at compile or design time

rather than at run time. Given such a schedule that is determined at compile time, we can extract information from it with a view towards optimizing the final implementation. The main contribution of this thesis is to present techniques for minimizing synchronization and inter-processor communication overhead in statically (i.e. compile time) scheduled multiprocessors where the program is derived from a dataflow graph specification. The strategy is to model run time execution of such a multiprocessor to determine how processors communicate and synchronize, and then to use this information to optimize the final implementation.

1.1 The Synchronous Dataflow model

1.1.1 Background

Dataflow is a well-known programming model in which a program is represented as a directed graph, where the vertices (or **actors**) represent computation and edges (or **arcs**) represent FIFO (first-in-first-out) queues that direct data values from the output of one computation to the input of another. Edges thus represent data precedences between computations. Actors consume data (or **tokens**) from their inputs, perform computation on them (**fire**), and produce certain number of tokens on their outputs.

Programs written in high-level functional languages such as pure LISP, and in dataflow languages such as Id and Lucid can be directly converted into dataflow graph representations; such a conversion is possible because these languages are designed to be *free of side-effects*, i.e. programs in these languages are not allowed to contain global variables or data structures, and functions in these languages cannot modify their arguments [Ack82]. Also, since it is possible to simulate any Turing machine in one of these languages, questions such as deadlock (or, equivalently, terminating behaviour) and determining maximum buffer sizes required to implement edges in the dataflow graph become undecidable. Several models based on dataflow with restricted semantics have been proposed; these models give up the descriptive power of general dataflow in exchange for properties that facilitate formal reasoning about programs specified in these models, and are useful in practise, leading to simpler implementation of the specified computation in hardware or software.

One such restricted model (and in fact one of the earliest graph based computation models) is the **computation graph** of Karp and Miller [Karp66]. In their seminal paper Karp and Miller establish that their computation graph model is *determinate*, i.e. the sequence of tokens produced on the edges of a given computation graph are unique, and do not depend on the order that the actors in the graph fire, as long as all data dependencies are respected by the firing order. The authors also provide an algorithm that, based on topological and algebraic properties of the graph, determines whether the computation specified by a given computation graph will eventually terminate. Because of the latter property, computation graphs clearly cannot simulate all Turing machines, and hence are not as expressive as a general dataflow language like Lucid or pure LISP. Computation graphs provide some of the theoretical foundations for the SDF model.

Another model of computation relevant to dataflow is the **Petri net model** [Peter81][Mur89]. A Petri net consists of a set of *transitions*, which are analogous to actors in dataflow, and a set of *places* that are analogous to arcs. Each transition has a certain number of input places and output places connected to it. Places may contain one or more *tokens*. A Petri net has the following semantics: a transition *fires* when all its input places have one or more tokens and, upon firing, it produces a certain number of tokens on each of its output places.

A large number of different kinds of Petri net models have been proposed in the literature for modeling different types of systems. Some of these Petri net models have the same expressive power as Turing machines: for example if transitions are allowed to posses "inhibit" inputs (if a place corresponding to such an input to a transition contains a token, then that transition is not allowed to fire) then a Petri net can simulate any Turing machine (pp. 201 in [Peter81]). Others (depending on topological restrictions imposed on how places and transitions can be interconnected) are equivalent to finite state machines, and yet others are similar to SDF graphs. Some extended Petri net models allow a notion of time, to model execution times of computations. There is also a body of work on stochastic extensions of timed Petri nets that are useful for modeling uncertainties in computation times. We will touch upon some of these Petri net models again in Chapter 4. Finally, there are Petri nets that distinguish between different classes of tokens in the specification (*colored* Petrinets), so that tokens can have information associated with them. We refer to [Peter81] [Mur89] for details on the extensive variety of Petri nets that have been proposed over the years.

The particular restricted dataflow model we are mainly concerned with in this thesis is the SDF — Synchronous Data Flow — model proposed by Lee and Messerschmitt [Lee87]. The SDF model poses restrictions on the firing of actors: the number of tokens produced (consumed) by an actor on each output (input) edge is a fixed number that is known at compile time. The arcs in an SDF graph may contain *initial tokens,* which we also refer to as **delays**. Arcs with delays can be interpreted as data dependencies across iterations of the graph; this concept will be formalized in the following chapter. In an actual implementation, arcs represent buffers in physical memory.

DSP applications typically represent computations on an indefinitely long data sequence; therefore the SDF graphs we are interested in for the purpose of signal processing must execute in a nonterminating fashion. Consequently, we must be able to obtain periodic schedules for SDF representations, which can then be run as infinite loops using a finite amount of physical memory. Unbounded buffers imply a sample rate inconsistency, and deadlock implies that all actors in the graph cannot be iterated indefinitely. Thus for our purposes, correctly constructed SDF graphs are those that can be scheduled periodically using a finite amount of memory. The main advantage of imposing restrictions on the SDF model (over a general dataflow model) lies precisely in the ability to determine whether or not an arbitrary SDF graph has a periodic schedule that neither deadlocks nor requires unbounded buffer sizes [Lee87]. The buffer sizes required to implement arcs in SDF graphs can be determined at compile time (recall that this

is not possible for a general dataflow model); consequently, buffers can be allocated statically, and run time overhead associated with dynamic memory allocation is avoided. The existence of a periodic schedule that can be inferred at compile time implies that a correctly constructed SDF graph entails no run time scheduling overhead.

An SDF graph in which every actor consumes and produces only one token from each of its inputs and outputs is called a **homogeneous SDF** graph (**HSDFG**). An HSDF graph actor fires when it has one or more tokens on all its input edges; it consumes one token from each input edge when it fires, and produces one token on all its output edges when it completes execution. A general (*multirate*) SDF graph can always be converted into an HSDF graph [Lee86]; this transformation may result in an exponential increase in the number of actors in the final HSDF graph (see [Pin95b] for an example of an SDF graph in which this blowup occurs). Such a transformation, however, appears to be necessary when constructing periodic multiprocessor schedules from multirate SDF graphs. There is some recent work on reducing the complexity of the HSDFG that results from transforming a given SDF graph by applying graph clustering techniques to that SDF graph [Pin95b]. Since we are concerned with multiprocessor schedules in this thesis, we assume we start with an application represented as a homogeneous SDF graph henceforth, unless we state otherwise. This of course results in no loss of generality because a multirate graph is converted into a homogeneous graph for the purposes of multiprocessor scheduling anyway. In Chapter 6 we discuss how the ideas that apply to HSDF graphs can be extended to graphs containing actors that display data-dependent behaviour (i.e. *dynamic* actors).

We note that an HSDFG is very similar to a **marked graph** in the context of Petri nets [Peter81]; transitions in the marked graph correspond to actors in the HSDFG, places correspond to edges, and initial tokens (or initial marking) of the marked graph correspond to initial tokens (or delays) in HSDFGs. We will represent delays using bullets (•) on the edges of the HSDFG; we indicate more than one delay on an edge by a number alongside the bullet, as in Fig. 1.1(a).

SDF should not be confused with **synchronous languages** [Hal93][Ben91] (e.g. LUSTRE, SIGNAL, and ESTEREL), which have very different semantics from SDF. Synchronous languages have been proposed for formally specifying and modeling reactive systems, i.e. systems that constantly react to stimuli from a given physical environment. Signal processing systems fall into the reactive category, and so do control and monitoring systems, communication protocols, man-machine interfaces, etc. In these languages variables are possibly infinite sequences of data of a certain type. Associated with each such sequence is a conceptual (and sometimes explicit) notion of a *clock signal*. In LUSTRE, each variable is explicitly associated with a clock, which determines the instants at which the value of that variable is defined. SIGNAL and ESTEREL do not have an explicit notion of a clock. The clock signal in LUSTRE is a sequence of Boolean values, and a variable in a LUSTRE program assumes its n th value when its corresponding clock takes its n th TRUE value. Thus we may relate one variable with another by means of their clocks. In ESTEREL, on the other hand, clock ticks are implicitly defined in terms of instants when the reactive system corresponding to an ESTEREL program receives (and reacts to) external events. All computations in synchronous language are defined with respect to these clocks.

In contrast, the term “synchronous” in the SDF context refers to the fact that SDF actors produce and consume fixed number of tokens, and these numbers are known at compile time. This allows us to obtain periodic schedules for SDF graphs such that the average rates of firing of actors are fixed relative to one another. We will not be concerned with synchronous languages in this thesis, although these languages have a close and interesting relationship with dataflow models used for specification of signal processing algorithms [Lee95].

1.1.2 Utility of dataflow for DSP

As mentioned before, dataflow models such as SDF (and other closely related models) have proven to be useful for specifying applications in signal processing and communications, with the goal of both simulation of the algorithm at

the functional or behavioural level, and for synthesis from such a high level specification to a software description (e.g. a C program) or a hardware description (e.g. VHDL) or a combination thereof. The descriptions thus generated can then be compiled down to the final implementation, e.g. an embedded processor, or an ASIC.

One of the reasons for the popularity of such dataflow based models is that they provide a formalism for block-diagram based visual programming, which is a very intuitive specification mechanism for DSP; the expressivity of the SDF model sufficiently encompasses a significant class of DSP applications, including multirate applications that involve upsampling and downsampling operations. An equally important reason for employing dataflow is that such a specification exposes parallelism in the program. It is well known that imperative programming styles such as C and FORTRAN tend to over-specify the control structure of a given computation, and compilation of such specifications onto parallel architectures is known to be a hard problem. Dataflow on the other hand imposes minimal data-dependency constraints in the specification, potentially enabling a compiler to detect parallelism. The same argument holds for hardware synthesis, where it is important to be able to exploit concurrency.

The SDF model has also proven useful for compiling DSP applications on single processors. Programmable digital signal processing chips tend to have special instructions such as a single cycle multiply-accumulate (for filtering functions), modulo addressing (for managing delay lines), bit-reversed addressing (for FFT computation); DSP chips also contain built in parallel functional units that are controlled from fields in the instruction (such as parallel moves from memory to registers combined with an ALU operation). It is difficult for automatic compilers to optimally exploit these features; executable code generated by commercially available compilers today utilizes one and a half to two times the program memory that a corresponding hand optimized program requires, and results in two to three times higher execution time compared to hand-optimized code [Zivo95]. There has been some recent work on compilation techniques for embedded software targetted towards DSP processors and microcontrollers [Liao95]; it is still too early to determine the impact of these techniques on automatic compilation for large-scale DSP/control applications, however.

Block diagram languages based on models such as SDF have proven to be a bridge between automatic compilation and hand coding approaches; a library of reusable blocks in a particular programming language is hand coded, this library then constitutes the set of atomic SDF actors. Since the library blocks are reusable, one can afford to carefully optimize and fine tune them. The atomic blocks are fine to medium grain in size; an atomic actor in the SDF graph may implement anything from a filtering function to a two input addition operation. The final program is then automatically generated by concatenating code corresponding to the blocks in the program according to the sequence prescribed by a schedule. This approach is mature enough that there are commercial tools available today, for example the SPW and COSSAP tools mentioned earlier, that employ this technique. Powerful optimization techniques have been developed for generating sequential programs from SDF graphs that optimize for metrics such as memory usage [Bhat94].

Scheduling is a fundamental operation that must be performed in order to implement SDF graphs on both uniprocessor as well as multiprocessors. Uniprocessor scheduling simply refers to determining the sequence of execution of actors such that all precedence constraints are met and all the buffers between actors (corresponding to arcs) return to their initial states. We discuss the issues involved in multiprocessor scheduling next.

1.2 Parallel scheduling

We recall that in the execution of a dataflow graph, actors fire when sufficient number of tokens are present at their inputs. The task of scheduling such a graph onto multiple processing units therefore involves assigning actors in the HSDFG to processors (the **processor assignment** step), ordering execution of these actors on each processor (the **actor ordering** step), and determining when each

actor fires such that all data precedence constraints are met. Each of these three tasks may be performed either at run time (a dynamic strategy) or at compile time (static strategy). We restrict ourselves to **non-preemptive** schedules, i.e. schedules where an actor executing on a processor can not be interrupted in the middle of its execution to allow another task to be executed. This is because preemption entails a significant implementation overhead and is therefore of limited use in embedded, time-critical applications.

Lee and Ha [Lee89] propose a scheduling taxonomy based on which of the scheduling tasks are performed at compile time and which at run time; we use the same terminology in this thesis. To reduce run time computation costs it is advantageous to perform as many of the three scheduling tasks as possible at compile time, especially in the context of algorithms that have hard real-time constraints. Which of these can be effectively performed at compile time depends on the information available about the execution time of each actor in the HSDFG.

For example, dataflow computers first pioneered by Dennis [Denn80] perform the assignment step at compile time, but employ special hardware (the *token-match* unit) to determine, at runtime, when actors assigned to a particular processor are ready to fire. The runtime overhead of token-matching and dynamic scheduling (within each processor) is fairly severe, so much so that dataflow architectures have not been commercially viable; even with expensive hardware support for dynamic scheduling, performance of such computers has been unimpressive.

The performance metric of interest for evaluating schedules is the **average iteration period** T: the average time it takes for all the actors in the graph to be executed once. Equivalently, we could use the throughput T^{-1} (i.e. the number of iterations of the graph executed per unit time) as a performance metric. Thus an optimal schedule is one that minimizes T.

In this thesis we focus on scheduling strategies that perform both processor assignment and actor ordering at compile time, because these strategies appear to be most useful for a significant class of real time DSP algorithms. Although

assignment and ordering performed at run time would in general lead to a more flexible implementation (because a dynamic strategy allows for run time variations in computation load and for operations that display data dependencies) the overhead involved in such a strategy is usually prohibitive and real-time performance guarantees are difficult to achieve. Lee and Ha [Lee89] define two scheduling strategies that perform the assignment and ordering steps at compile time: **fully-static** and **self-timed**. We use the same terminology in this thesis.

1.2.1 Fully-static schedules

In the fully-static (**FS**) strategy, the exact firing time of each actor is also determined at compile time. Such a scheduling style is used in the design of systolic array architectures [Kung88], for scheduling VLIW processors [Lam88], and in high-level VLSI synthesis of applications that consist only of operations with guaranteed worst-case execution times [DeMich94]. Under a fully static schedule, all processors run in lock step; the operation each processor performs on each clock cycle is predetermined at compile time and is enforced at run time either implicitly (by the program each processor executes, perhaps augmented with "nop"s or idle cycles for correct timing) or explicitly (by means of a program sequencer for example).

A fully-static schedule of a simple HSDFG G is illustrated in Fig. 1.1. The FS schedule is schematically represented as a Gantt chart that indicates the processors along the vertical axis, and time along the horizontal axis. The actors are represented as rectangles with horizontal length equal to the execution time of the actor. The left side of each actor in the Gantt chart corresponds to its starting time. The Gantt chart can be viewed as a processor-time plane; scheduling can then be viewed as a mechanism to tile this plane while minimizing total schedule length and idle time ("empty spaces" in the tiling process). Clearly, the FS strategy is viable only if actor execution time estimates are accurate and data-independent or if tight worst-case estimates are available for these execution times.

As shown in Fig. 1.1, two different types of FS schedules arise, depending

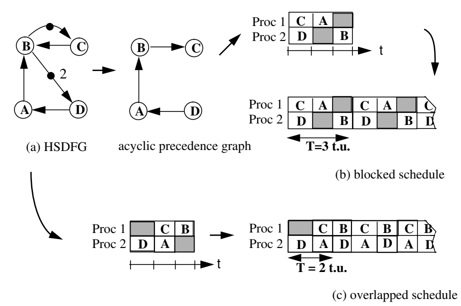
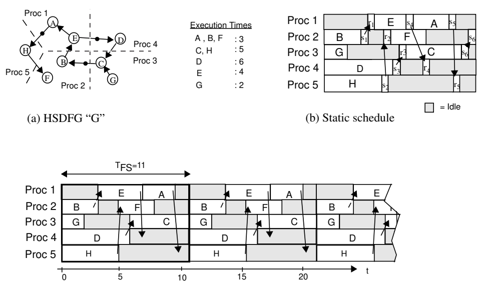


Figure 1.1. Fully static schedule

on how successive iterations of the HSDFG are treated. Execution times of all actors are assumed to be one time unit (t.u.) in this example. The FS schedule in Fig. 1.1(b) represents a **blocked schedule**: successive iterations of the HSDFG in a blocked schedule are treated separately so that each iteration is completed before the next one begins. A more elaborate blocked schedule on five processors is shown in Fig. 1.2. The HSDFG is scheduled as if it executes for only one iteration, i.e. inter-iteration dependencies are ignored; this schedule is then repeated to get an infinite periodic schedule for the HSDFG. The length of the blocked schedule determines the average iteration period T. The scheduling problem is then to obtain a schedule that minimizes T (which is also called the **makespan** of the schedule). A lower bound on T for a blocked schedule is simply the length of the **critical path** of the graph, which is the longest delay-free path in the graph.

Ignoring the inter-iteration dependencies when scheduling an HSDFG is equivalent to the classical multiprocessor scheduling problem for an Acyclic Precedence Graph (APG): the acyclic precedence graph is obtained from the given



(a) HSDFG "G" (b) Static schedule

(c) Fully-static execution

Figure 1.2. Fully-static schedule on five processors

HSDFG by eliminating all edges with delays on them (edges with delays represent dependencies across iterations) and replacing multiple edges that are directed between the same two vertices in the same direction with a single edge. This replacement is done because such multiple edges represent identical precedence constraints; these edges are taken into account individually during buffer assignment, however. Optimal multiprocessor scheduling of an acyclic graph is known to be NP-Hard [Garey79], and a number of heuristics have been proposed for this problem. One of the earliest, and still popular, solutions to this problem is *list scheduling*, first proposed by Hu [Hu61]. List scheduling is a greedy approach: whenever a task is ready to run, it is scheduled as soon as a processor is available to run it. Tasks are assigned priorities, and among the tasks that are ready to run at any instant, the task with the highest priority is executed first. Various researchers have proposed different priority mechanisms for list scheduling [Adam74], some of which use critical path based (CPM) methods [Ram72][Koh75][Blaz87] ([Blaz87] summarizes a large number of CPM based heuristics for scheduling).

The heuristics mentioned above ignore communication costs between processors, which is often inappropriate in actual multiprocessor implementations. An edge of the HSDFG that crosses processor boundaries after the processor assignment step represents interprocessor communication (**IPC**) (illustrated in Fig. 1.3(a)). These communication points are usually implemented using *send* and *receive* primitives that make use of the processor interconnect hardware. These primitives then have an execution cost associated with them that depends on the multiprocessor architecture and hardware being employed. Fully-static scheduling heuristics that take communication costs into account include [Sark89][Sih91][Prin91].

Computations in the HSDFG, however, are iterated essentially infinitely. The blocked scheduling strategies discussed thus far ignore this fact, and thus pay a penalty in the quality of the schedule they obtain. Two techniques that enable blocked schedules to exploit inter-iteration parallelism are **unfolding** and **retiming**. The unfolding strategy schedules J iterations of the HSDFG together, where J is called the **blocking factor**. Thus the schedule in Fig. 1.1(b) has $J = 1$. Unfolding often leads to improved blocked schedules (pp. 78-100 [Lee86], [Parhi91]), but it also implies a factor of J increase in program memory size and also in the size of the scheduling problem, which makes unfolding somewhat impractical.

Retiming involves manipulating delays in the HSDFG to reduce the critical path in the graph. This technique has been explored in the context of maximizing clock rates in synchronous digital circuits [Lei83], and has been proposed for improving blocked schedules for HSDFGs ("cutset transformations" in [Lee86], and [Hoang93]).

Fig. 1.1(c) illustrates an example of an **overlapped schedule**. Such a schedule is explicitly designed such that successive iterations in the HSDFG overlap. Obviously, overlapped schedules often achieve a lower iteration period than blocked schedules. In Fig. 1.1, for example, the iteration period for the blocked schedule is 3 units whereas it is 2 units for the overlapped schedule. One might

wonder whether overlapped schedules are fundamentally superior to blocked schedules with the unfolding and retiming operations allowed. This question is settled in the affirmative by Parhi and Messerschmitt [Parhi91]; the authors provide an example of an HSDFG for which no blocked schedule can be found, even allowing unfolding and retiming, that has a lower or equal iteration period than the overlapped schedule they propose.

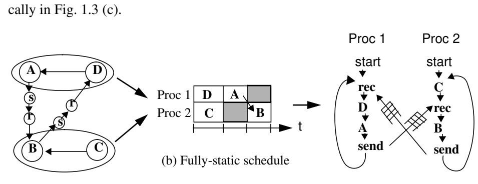
Optimal resource constrained overlapped scheduling is of course NP-Hard, although a periodic overlapped schedule in the absence of processor constraints can be computed efficiently and optimally [Parhi91][Gasp92].

Overlapped scheduling heuristics have not been as extensively studied as blocked schedules. The main work in this area is by Lam [Lam88], and deGroot [deGroot92], who propose a modified list scheduling heuristic that explicitly constructs an overlapped schedule. Another work related to overlapped scheduling is the "cyclo-static scheduling" approach proposed by Schwartz. This approach attempts to optimally tile the processor-time plane to obtain the best possible schedule. The search involved in this process has a worst case complexity exponential in the size of the input graph, although it appears that the complexity is manageable in practice, at least for small examples [Schw85].

1.2.2 Self-timed schedules

The fully-static approach introduced in the previous section cannot be used when actors have variable execution times; the FS approach requires precise knowledge of actor execution times to guarantee sender-receiver synchronization. It is possible to use worst case execution times and still employ an FS strategy, but this requires tight worst case execution time estimates that may not be available to us. An obvious strategy for solving this problem is to introduce explicit synchronization whenever processors communicate. This leads to the self-timed scheduling (**ST**) strategy in the scheduling taxonomy of Lee and Ha [Lee89]. In this strategy we first obtain an FS schedule using the techniques discussed in section 1.2, making use of the execution time estimates. After computing the FS schedule (Fig. 1.3

(b)), we simply discard the timing information that is not required, and only retain the processor assignment and the ordering of actors on each processor as specified by the FS schedule (Fig. 1.3(c)). Each processor is assigned a sequential list of actors, some of which are *send* and *receive* actors, that it executes in an infinite loop. When a processor executes a communication actor, it synchronizes with the processor(s) it communicates with. Exactly when a processor executes each actor depends on when, at run time, all input data for that actor are available, unlike the fully-static case where no such run time check is needed. Conceptually, the processor sending data writes data into a FIFO buffer, and blocks when that buffer is full; the receiver on the other hand blocks when the buffer it reads from is empty. Thus flow control is performed at run time. The buffers may be implemented using shared memory, or using hardware FIFOs between processors. In a self-timed strategy, processors run sequential programs and communicate when they execute the communication primitives embedded in their programs, as shown schematically in Fig. 1.3 (c).



(a) HSDFG (b) Fully-static schedule (c) Self-timed implementation (schematic)

Figure 1.3. Steps in a self-timed scheduling strategy

An ST strategy is robust with respect to changes in execution times of actors, because sender-receiver synchronization is performed at run time. Such a strategy, however, implies higher IPC costs compared to the fully-static strategy because of the need for synchronization (e.g. using semaphore management). In addition the ST strategy faces arbitration costs: the FS schedule guarantees mutually exclusive access of shared communication resources, whereas shared resources need to be arbitrated at run time in the ST schedule. Consequently, whereas IPC in the FS schedule simply involves reading and writing from shared memory (no synchronization or arbitration needed), implying a cost of a few processor cycles for IPC, the ST strategy requires of the order of tens of processor cycles, unless special hardware is employed for run time flow control. We discuss in detail how this overhead arises in a shared bus multiprocessor configuration in Chapter 3.

Run time flow control allows variations in execution times of tasks; in addition, it also simplifies the compiler software, since the compiler no longer needs to perform detailed timing analysis and does not need to adjust the execution of processors relative to one another in order to ensure correct sender-receiver synchronization. Multiprocessor designs, such as the Warp array [Ann87][Lam88] and the 2-D MIMD (Multiple Instruction Multiple Data) array of [Ziss87], that could potentially use fully-static scheduling, still choose to implement such run time flow control (at the expense of additional hardware) for the resulting software simplicity. Lam presents an interesting discussion on the trade-off involved between hardware complexity and ease of compilation that ensues when we consider dynamic flow control implemented in hardware versus static flow control enforced by a compiler (pp. 50-68 of [Lam89]).

1.2.3 Execution time estimates and static schedules

We assume we have reasonably good estimates of actor execution times available to us at compile time to enable us to exploit static scheduling techniques; however, these estimates need not be exact, and execution times of actors may even be data-dependent. Thus we allow actors that have different execution times from one iteration of the HSDFG to the next, as long as these variations are small or rare. This is typically the case when estimates are available for the task execution times, and actual execution times are close to the corresponding estimates with high probability, but deviations from the estimates of (effectively) arbitrary

magnitude occasionally occur due to phenomena such as cache misses, interrupts, user inputs or error handling. Consequently, tight worst-case execution time bounds cannot generally be determined for such operations; however, reasonably good execution time estimates can in fact be obtained for these operations, so that static assignment and ordering techniques are viable. For such applications self-timed scheduling is ideal, because the performance penalty due to lack of dynamic load balancing is overcome by the much smaller run time scheduling overhead involved when static assignment and ordering is employed.

The estimates for execution times of actors can be obtained by several different mechanisms. The most straightforward method is for the programmer to provide these estimates when he writes the library of primitive blocks. This strategy is used in the Ptolemy system, and is very effective for the assembly code libraries, in which the primitives are written in the assembly language of the target processor (Ptolemy currently supports the Motorola 56000 and 96000 processors). The programmer can provide a good estimate for blocks written in such a library by counting the number of processor cycles each instruction consumes, or by profiling the block on an instruction-set simulator.

It is more difficult to estimate execution times for blocks that contain control constructs such as data-dependent iterations and conditionals within their body, and when the target processor employs pipelining and caching. Also, it is difficult, if not impossible, for the programmer to provide reasonably accurate estimates of execution times for blocks written in a high-level language (as in the C code generation library in Ptolemy). The solution adopted in the GRAPE system [Lauw90] is to automatically estimate these execution times by compiling the block (if necessary) and running it by itself in a loop on an instruction-set simulator for the target processor. To take into account data-dependent execution behaviour, different input data sets can be provided for the block during simulation. Either the worst case or the average case execution time is used as the final estimate.

The estimation procedure employed by GRAPE is obviously time consum-

ing; in fact estimation turns out to be the most time consuming step in the GRAPE design flow. Analytical techniques can be used instead to reduce this estimation time; for example, Li and Malik [Li95] have proposed algorithms for estimating the execution time of embedded software. Their estimation technique, which forms a part of a tool called *cinderella*, consists of two components: 1) determining the sequence of instructions in the program that results in maximum execution time (program path analysis) and 2) modeling the target processor to determine how much time the worst case sequence determined in step 1 takes to execute (micro-architecture modeling). The target processor model also takes the effect of instruction pipelines and cache activity into account. The input to the tool is a generic C program with annotations that specify the loop bounds (i.e. the maximum number of iterations that a loop runs for). Although the problem is formulated as an integer linear program (ILP), the claim is that practical inputs to the tool can be efficiently analyzed using a standard ILP solver. The advantage of this approach, therefore, is the efficient manner in which estimates are obtained as compared to simulation.

It should be noted that the program path analysis component of the Li and Malik technique is in general an undecidable problem; therefore for these techniques to function, the programmer must ensure that his or her program does not contain pointer references, dynamic data structures, recursion, etc. and must provide bounds on all loops. Li and Malik's technique also depends on the accuracy of the processor model, although one can expect good models to eventually evolve for DSP chips and microcontrollers that are popular in the market.

The problem of estimating execution times of blocks is central for us to be able to effectively employ compile time design techniques. This problem is an important area of research in itself, and the strategies employed in Ptolemy and GRAPE, and those proposed by Li and Malik are useful techniques, and we expect better estimation techniques to be developed in the future.

1.3 Application-specific parallel architectures

There has been significant amount of research on general purpose high-performance parallel computers. These employ expensive and elaborate interconnect topologies, memory and Input/Output (I/O) structures. Such strategies are unsuitable for embedded DSP applications as we discussed earlier. In this section we discuss some application-specific parallel architectures that have been employed for signal processing, and contrast them to our approach.

1.3.1 Dataflow DSP architectures

There have been a few multiprocessors geared towards signal processing that are based on the dataflow architecture principles of Dennis [Denn80]. Notable among these are Hughes Data Flow Multiprocessor [Gau85], the Texas Instruments Data Flow Signal Processor [Grim84], and the AT&T Enhanced Modular Signal Processor [Bloch86]. The first two perform the processor assignment step at compile time (i.e. tasks are assigned to processors at compile time) and tasks assigned to a processor are scheduled on it dynamically; the AT&T EMPS performs even the assignment of tasks to processors at runtime.

Each one of these machines employs elaborate hardware to implement dynamic scheduling within processors, and employs expensive communication networks to route tokens generated by actors assigned to one processor to tasks on other processors that require these tokens. In most DSP applications, however, such dynamic scheduling is unnecessary since compile time predictability makes static scheduling techniques viable. Eliminating dynamic scheduling results in much simpler hardware without an undue performance penalty.

Another example of an application-specific dataflow architecture is the NEC μPD7281 [Chase84], which is a single chip processor geared towards image processing. Each chip contains one functional unit; multiple such chips can be connected together to execute programs in a pipelined fashion. The actors are statically assigned to each processor, and actors assigned to a given processor are scheduled on it dynamically. The primitives that this chip supports, convolution, bit manipulations, accumulation, etc., are specifically designed for image processing applications.

1.3.2 Systolic and wavefront arrays

Systolic arrays consist of processors that are locally connected and may be arranged in different topologies: mesh, ring, torus, etc. The term "systolic" arises because all processors in such a machine run in lock-step, alternating between a computation step and a communication step. The model followed is usually SIMD (Single Instruction Multiple Data). Systolic arrays can execute a certain class of problems that can be specified as "Regular Iterative Algorithms (RIA)" [Rao85]; systematic techniques exist for mapping an algorithm specified in a RIA form onto dedicated processor arrays in an optimal fashion. Optimality includes metrics such as processor and communication link utilization, scalability with the problem size, achieving best possible speedup for a given number of processors, etc. Several numerical computation problems were found to fall into the RIA category: linear algebra, matrix operations, singular value decomposition, etc. (see [Kung88][Leigh92] for interesting systolic array implementations of a variety of different numerical problems). Only fairly regular computations can be specified in the RIA form; this makes the applicability of systolic arrays somewhat restrictive.

Wavefront arrays are similar to systolic arrays except that processors are not under the control of a global clock. Communication between processors is asynchronous or self-timed; handshake between processors ensures run time synchronization. Thus processors in a wavefront array can be complex and the arrays themselves can consist of a large number of processors without incurring the associated problems of clock skew and global synchronization. Again, similar to FS versus ST scheduling, the flexibility of wavefront arrays over systolic arrays comes at the cost of extra handshaking hardware.

The Warp project at Carnegie Mellon University [Anna87] is an example

of a programmable systolic array, as opposed to a dedicated array designed for one specific application. Processors are arranged in a linear array and communicate with their neighbors through FIFO queues. Programs are written for this computer in a language called W2 [Lam88]. The Warp project also led to the iWarp design [Bork88], which has a more elaborate inter-processor communication mechanism than the Warp machine. An iWarp node is a single VLSI component, composed of a computation engine and a communication engine; the latter consists of a crossbar and data routing mechanisms. The iWarp nodes can be connected in various single and two dimensional topologies, and point to point message-passing type communication is supported.

1.3.3 Multiprocessor DSP architectures

In this section we discuss multiprocessors that make use of multiple off the shelf programmable DSP chips.

The SMART architecture [Koh90] is a reconfigurable bus design comprised of AT&T DSP32C processors, and custom VLSI components for routing data between processors. Clusters of processors may be connected onto a common bus, or may form a linear array with neighbor to neighbor communication. This allows the multiprocessor to be reconfigured depending on the communication requirement of the particular application being mapped onto it. Scheduling and code generation for this machine is done by the McDAS compiler [Hoang93].

The DSP3 multiprocessor [Shive92] was built at AT&T, and is comprised of DSP32C processors connected in a mesh configuration. The mesh interconnect is implemented using custom VLSI components for data routing. Each processor communicates with four of its adjacent neighbors through this router, which consists of input and output queues, and a crossbar that is configurable under program control. Data packets contain headers that indicate the ID of the destination processor.

The Ring Array Processor (RAP) system [Morg92] uses TI DSP320C30 processors connected in a ring topology. This system is designed specifically for

speech recognition applications based on artificial neural networks. The RAP system consists of several boards that are attached to a host workstation, and acts as a coprocessor for the host. The unidirectional pipelined ring topology employed for interprocessor communication was found to be ideal for the particular algorithms that were to be mapped to this machine. The ring structure is similar to the SMART array, except that no processor ID is included with the data, and processor reads and writes into the ring are scheduled in a fully-static fashion. The ring is used to broadcast data from one processor to all the others during one phase of the neural net algorithm, and is used to shift data from processor to processor in a pipelined fashion in the second phase.

The MUSIC system [Gunz92] uses Motorola DSP96000 processors, and has been designed for neural network simulations and scientific simulations. An "intelligent" communication network, implemented on Xilinx gate arrays, broadcasts data generated by any processing element (PE) to all the other PEs. The PEs are arranged in a ring topology. This kind of broadcast mechanism is suited to the applications the MUSIC system targets: the outputs from one layer of a multi-layer perceptron (a kind of neural net) is needed by all the "neurons" in the next layer, making broadcasting an ideal strategy when the different net layers are executed in a pipelined fashion. The molecular dynamics example the authors provide also benefits from this broadcast mechanism.

1.4 Thesis overview: our approach and contributions

We argued that the self-timed scheduling strategy is suited towards parallel implementation for DSP. The multiprocessor architectures we discussed in the previous section support this argument: the dataflow architectures in section 1.3.1 use dynamic scheduling, but pay a high hardware cost, which makes them unsuited for embedded applications. In the case of the NEC dataflow chips, parallelism is mainly derived through pipelined execution. The dataflow model of execution that is implemented in hardware in this chip, although elegant, is of limited use for the

image processing applications that this part has been designed for; the order in which each processor executes instructions assigned to it can potentially be fixed at compile time without loss in parallelism.

Systolic arrays normally employ a fully-static strategy. As we discussed before, RIA specifications and the primarily SIMD approach used in systolic array mapping techniques restrict their domain of applicability. The approach taken by these techniques is to use a large number of very simple processors to perform the computation, whereas the approach we follow in this thesis is to use a small number of powerful processors. This enables us to handle algorithms specified as dataflow graphs where the actors are tasks with a potentially large granularity. The parallelism we employ is therefore at the task level (functional parallelism). Such a strategy gives up some of the optimality properties that systolic array mapping techniques guarantee in exchange for a larger application domain. Again, utilizing a number of pretested processor cores is economically more attractive than building a systolic array implementation from scratch.

Ideally we would like to exploit the strategy of partitioning data among different processors (data parallelism) that systolic techniques employ, along with task level parallelism. There has not been much work in this direction, although the work of Printz [Prin91], and the Multidimensional SDF model proposed by Lee in [Lee93], are two promising approaches for combining data and functional parallelism.

The multiple DSP machines we discussed in the last section all employ some form of self-timed scheduling. Clearly, general purpose parallel machines like the Thinking Machines CM-5 and Stanford Dash multiprocessor can also be programmed using the self-timed scheduling style, since these machines provide mechanisms for run time synchronization and flow control. These machines, however, do not attempt to make use of the fact that the interprocessor communication pattern in a self-timed implementation is fairly predictable. In this thesis we explore techniques that optimize the parallel implementation of a self-timed schedule by performing compile time analysis of the schedule to determine the pattern of processor communication and synchronization. We assume we are given a self-timed schedule, i.e. we are given a processor assignment and actor ordering. This could be obtained using any of the techniques discussed in section 1.2. For the actual programs examples that we discuss in this thesis, we use one of three scheduling methods that are currently available within the Ptolemy system: Sih's scheduling heuristics [Sih92], Hu's list scheduling heuristic, and manual scheduling.

The first technique we present is the **Ordered Transaction** (**OT**) strategy. In the OT strategy, we impose additional constraints on the self-timed execution. In particular we force an order in which processors communicate. Doing this allows us to eliminate arbitration and synchronization costs, like the fully-static strategy, but enables us to retain some of the flexibility of the self-timed schedule in that execution times of actors need only be *estimates* rather than values accurate down to the clock cycle.

In Chapter 3 we discuss the OT strategy in detail and present the hardware design and implementation of a four processor machine — the Ordered Memory Access Architecture (OMA) — that employs the OT idea to achieve low cost interprocessor communication. The OMA is a shared bus machine that uses shared memory for IPC. The order in which processors access shared memory for the purpose of communication is predetermined at compile time and enforced by a bus controller on the board, resulting in a low cost IPC mechanism without the need for explicit synchronization.

Reducing IPC costs is particularly important for parallel implementation of embedded real-time applications, because there is usually a premium on processor cycles in these situations. For example, if we process audio at a sample rate of 44 KHz on a multiprocessor that consists of processors with a 60 nanosecond cycle time, we have about 380 instructions on average per processor to complete all operations on each audio sample. IPC can potentially waste precious processor cycles, negating some of the benefits of using multiple processors.

In Chapter 4 we present a graph theoretic scheme for modeling self-timed schedules using a structure we call the **IPC graph** that takes into account the pro-

cessor assignment and ordering constraints that a self-timed schedule specifies. In the context of the OT scheme, we use the IPC graph construction to determine a transaction order that makes optimal use of actor execution time information available to us during the compilation or design phase. The idea is to predict the pattern of processor communications using the IPC graph, and then to use this pattern to determine the transaction order. We outline this procedure in Chapter 4. We also discuss the effect of run time variations in execution times of actors on the performance of a multiprocessor executing a self-timed schedule.

Determining the pattern of processor communications is relatively straightforward in SIMD implementations. Techniques applied to systolic arrays in fact use the regular communication pattern to determine an optimal interconnect topology for a given algorithm. An interesting architecture in this context is the GF11 machine built at IBM [Beet85]. The GF11 is an SIMD machine in which processors are interconnected using a Benes network, which allows the GF11 to support a variety of different interprocessor communication topologies rather than a fixed topology, e.g. a mesh or a ring, that most systolic arrays support.

Benes networks are non-blocking, i.e. they can provide connections from all the network inputs to the network outputs simultaneously according to any specified permutation. These networks achieve the functional capability of a full crossbar switch with much simpler hardware. The drawback, however, is that in a Benes network, computing switch settings needed to achieve a particular permutation involves a somewhat complex algorithm [Leigh92]. In the GF11, this problem is solved by precomputing the switch settings based on the program to be executed on the array, and reconfiguring the Benes network at runtime based on these predetermined switch settings.

The OT strategy is along similar lines as the static network configuration strategy employed in the GF11. The GF11 strategy, however, applies to a FS scheduling scenario: the communication network and the processors run synchronously and the compiler must determine what the network and the processors do at each clock cycle. As we discussed earlier, this restriction narrows down the appli-

cation domain for such a scheme. Indeed, the GF11 is employed mainly for scientific computation applications that have a very regular communication and computation structure (linear algebra, finite element methods, FFTs, etc.). Again, the techniques we explore in this thesis apply to an MIMD style of computation where we do not assume precise knowledge of execution times of tasks, or even the availability of guaranteed worst case execution times for operations.

The second technique we present in this thesis consists of efficient algorithms that minimize the overall synchronization activity in the implementation of a given self-timed schedule. A straightforward implementation of a self-timed schedule often includes **redundant synchronization** points, i.e. the objective of a certain set of synchronizations is guaranteed as a side effect of other synchronization points in the system. In Chapter 5 we present efficient (polynomial time) algorithms for detecting and eliminating such redundant synchronization operations. We relate our techniques to the algorithm proposed by Shaffer [Sha89] for minimizing *directed synchronizations* in multiprocessors that are essentially executing self-timed schedules, and to the work described by Dietz *et. al.* [Dietz92] for minimizing the number of barrier synchronizations in a multiprocessor that contains barrier synchronization hardware. It is also possible to reduce the overall synchronization cost of a self-timed implementation by adding synchronization points between processors that were not present in the schedule specified originally. In Chapter 5, we present a technique, called **resynchronization**, for systematically manipulating synchronization points in this manner. The work described in this chapter is a part of an ongoing research effort in collaboration with Dr. Shuvra Bhattacharyya, a research scientist at Hitachi America Ltd.

The ordered transaction strategy can be viewed as a hardware approach that optimizes for IPC and synchronization overhead in statically scheduled multiprocessor implementations. The synchronization optimization techniques on the other hand operate at the level of a scheduled parallel program by altering the synchronization structure of a given schedule to minimize the synchronization overhead in the final implementation.

In Chapter 6 we discuss ideas for extending the OT strategy to models more powerful than SDF, e.g. the Boolean dataflow model. The strategy here is to assume we have only a small number of control constructs in the SDF graph and explore techniques for this case. This allows us to expand the domain of applicability of our techniques without having to deal with the complexity of tackling a general BDF model. Finally, we present our conclusions in Chapter 7 and point out future directions for this research.

In the following chapter (Chapter 2) we lay down some of the notation and terminology we follow in the remainder of the thesis.