There are several different approaches of mapping software into hardware:

- **Software Mapping of SDF**
  - Sequential (on a single CPU)
    - Using a Dynamic Schedule
      - *Single-thread executive*
      - *Multithreading*
  - Using a Static Schedule
    - *Single-thread executive*
    - *Inlined*
  - Parallel (on a multiple CPUs)
    - *Processor Networks*

We will first focus on implementing dataflow on single-processor systems.

This requires a sequential scheduling of dataflow actors.
Software Implementation of Data-Flow

• Using a **dynamic** schedule
  Here, the CPU determines the order in which actors should execute at runtime
  by testing firing rules to evaluate which actor can run

  Dynamic scheduling of an SDF system can be done using a **single-thread** executive or **multi-threading**

• Using a **static** schedule
  Here, we need to determine upfront the order of actor firing

  Allows for a single-threaded execution and an optimization in which the entire dataflow graph is 'inlined' into a single function

Recall the essential features of SDF graphs:
  SDF graphs represent concurrent systems, and use **actors** and **FIFO queues** to communicate
  Firing only depends on the availability of data (tokens) in the FIFO queues
  The amount of tokens produced/consumed per firing is given by labels
Software Implementation of Data-Flow

FIFO Queues:
In principle, SDF systems require **infinite** FIFO queues

In practice, queues have a limited # of positions, and need overflow detection

Another approach is to create a FIFO that grows dynamically each time the FIFO overflows

However, if we know a PASS, we know the *maximum number* of tokens on each queue and can set the queue size accordingly

Software object \( Q \)

\[
\text{void } Q\text{.put(element &)} \quad \text{void } Q\text{.get()}
\]

\[
\text{unsigned } Q\text{.getsize()}
\]
Software Implementation of Data-Flow

A typical software interface of a FIFO queue has *two parameters* and *three methods*

- The number of elements $N$ that can be stored by the queue
- The data type *element* of queue components
- A method to *put* elements into the queue
- A method to *get* elements from the queue
- A method to *test the number* of elements in the queue

A standard data structure such as a *circular queue* can be used

A *circular queue* consists of an array, a *write-pointer* and a *read-pointer*, and use *modulo* addressing, i.e., element $I$ at $(Rptr + I) \mod \text{array\_size}$.

![Diagram of circular queue operations](image)
Software Implementation of Data-Flow Model

Example fifo in C:

```c
#define MAXFIFO 1024

typedef struct fifo {
    int data[MAXFIFO]; // array
    unsigned wpwr;     // write pointer
    unsigned rptr;     // read pointer
} fifo_t;

void init_fifo(fifo_t *F);
void put_fifo(fifo_t *F, int d);
int get_fifo(fifo_t *F);
unsigned fifo_size(fifo_t *F);

int main()
{
    fifo_t F1;
    init_fifo(&F1); // resets wpwr, rptr
```
Software Implementation of Data-Flow Model

```c
put_fifo(&F1, 5);
put_fifo(&F1, 6);  // prints: 2 5
printf("%d %d\n", fifo_size(&F1), get_fifo(&F1));
printf("%d\n", fifo_size(&F1));  // prints: 1
```

**Actors:**

A dataflow actor can be represented as a function, with an interface to FIFOs.

The **firing** of an actor can be implemented as a simple C function.

![Diagram of dataflow actor implementation](image)

*Fig. 2.18* Software implementation of the dataflow actor
Software Implementation of Data-Flow Model

The function checks the firing rules and manipulates the input and output queue. Think of this as a *small controller* that controls its execution.

The *local controller* of an actor has **three** states:

- In the *read state*, it remains idle until a token arrives at the input queue.

- In the *work state*, the controller reads one token and runs the function, producing an output token which is put on the output queue in the *write state*.

We must make sure the firing rule is implemented correctly.

When an SDF actor fires, it reads all input queues and writes into all output queues according to the specified production and consumption rates.

A C implementation:

```c
typedef struct actorio {
    fifo_t *in1, *in2;
    fifo_t *out1, *out2;
} actorio_t;
```
Software Implementation of Data-Flow Model

```c
void sort_actor(actorio_t *g)
{
    int r1, r2;
    while((fifo_size(g->in1) > 0) && (fifo_size(g->in2) > 0))
    {
        r1 = get_fifo(g->in1);
        r2 = get_fifo(g->in2);
        put_fifo(g->out1, (r1 > r2) ? r1 : r2);
        put_fifo(g->out2, (r1 > r2) ? r2 : r1);
    }
}
```

Sequential Targets with a Dynamic Schedule:

In a dynamic system schedule, the firing rules of the actors will be tested at runtime.
Software Implementation: Single-Thread Dynamic Schedules

In a single-thread dynamic schedule, we implement the system schedule as a function that instantiates all actors and queues.

And then it calls the actors in a round-robin fashion.

```c
void main() {
    fifo_t F1, F2, F3, F4;
    actorio_t sort_io;
    ...
    sort_io.in1 = &F1;
    sort_io.in2 = &F2;
    sort_io.out1 = &F3;
    sort_io.out2 = &F4;
    while (1)
        {
            sort_actor(&sort_io);
            // .. call other actors
        }
}
```
Software Implementation: Single-Thread Dynamic Schedules

But what is the most appropriate call order of the actors in the system schedule? Remember that it is **impossible** to call the actors in the **wrong** order,

This is true b/c each of them still has a firing rule that prevents them from running when there is no data available.

Even though \textit{snk} will be called as often as \textit{src}, the firing rule of \textit{snk} will only allow that actor to run when there is sufficient data available.

In Figure 2.19a, this means that the \textit{snk} actor will fire **ONLY every other time** it is called.

\begin{itemize}
  \item Fig. 2.19 Two graph topologies and a dynamic system schedule
\end{itemize}
Software Implementation: Single-Thread Dynamic Schedules

While this type of dynamic scheduling prevents actors from running prematurely, some actors will produce tokens and cause queues to grow.

In Figure 2.19b, the \textit{src} actor produces two tokens per invocation while the \textit{snk} actor reads ONLY one per invocation.

The basic problem with the system schedule in Figure 2.19 is that the system schedule firing rate \textbf{differs} from the firing rate for a PASS.

For example, the PASS for this system would be \((\textit{src}, \textit{snk}, \textit{snk})\).

Two solutions:

- Solution 1: Adjust the system schedule to match the PASS

```c
void main() {
  ..
  while (1) {
    src_actor(&src_io);
    snk_actor(&snk_io);
    snk_actor(&snk_io);
  }
}
```
Software Implementation: Single-Thread Dynamic Schedules

Unfortunately, this solution is not elegant, because it voids the purpose of having a dynamic scheduler.

- Solution 2: Adjust the \textit{snk} actor code to continue execution as long as there are tokens present.

```c
void snk_actor(actorio_t *g) {
    int r1, r2;
    while ((fifo_size(g->in1) > 0)) {
        r1 = get_fifo(g->in1);
        ... // do processing
    }
}
```

Multi-Thread Dynamic Schedules

The actor functions as described are captured as real functions which \texttt{exit} in between invocations.

This prevents them from maintaining \textit{local} state, and forces global variables.
Software Implementation: Multi-Thread Dynamic Schedules

In multi-threaded programming, each actor lives in a separate thread.

For example, in a C program with two functions, each thread is executing one of the functions.

In a single CPU scenario, the threads are time-interleaved by a scheduler and the threads voluntarily release control back to the scheduler.

![Diagram showing cooperative multi-threading](image)

**Fig. 2.20** Example of cooperative multi-threading
Software Implementation: Multi-Thread Dynamic Schedules

Two functions are needed to build a threading system: `create()` and `yield()`

The scheduler can apply different strategies to schedule thread execution, with the simplest one shown above as a *round-robin* schedule

*Quickthreads* is a **cooperative multithreading library**

The quickthreads API (Application Programmers Interface) consists of 4 functions
- `spt_init()`: initializes the threading system
- `spt_create(stp_userf_t *F, void *G)` creates a thread that will start execution with user function F, and will be passed a single argument G
- `stp_yield()` releases control over the thread to the scheduler
- `stp_abort()` terminates a thread (prevents it from being scheduled)

Here’s an example

```c
#include "../qt/stp.h"
#include <stdio.h>
```
Software Implementation: Multi-Thread Dynamic Schedules

```c
void hello(void *null) {
    int n = 3;
    while (n-- > 0) {
        printf("hello\n");
        stp_yield();
    }
}

void world(void *null) {
    int n = 5;
    while (n-- > 0) {
        printf("world\n");
        stp_yield();
    }
}
```
Software Implementation: Multi-Thread Dynamic Schedules

```c
int main(int argc, char **argv)
{
    stp_init();
    stp_create(hello, 0);
    stp_create(world, 0);
    stp_start();
    return 0;
}
```

To compile and execute:
```
gcc -c ex1.c -o ex1 ../qt/libstp.a ../qt/libqt.a
./ex1
hello
world
hello
world
hello
world\nworld\nworld
```
Software Implementation: Multi-Thread Dynamic Schedules

A multi-threaded version of the SDF scheduler, using the sort_actor

```c
void sort_actor(actorio_t *g) {
    int r1, r2;
    while (1) {
        if ((fifo_size(g->in1) > 0) &&
            (fifo_size(g->in2) > 0)) {
            r1 = get_fifo(g->in1);
            r2 = get_fifo(g->in2);
            put_fifo(g->out1, (r1 > r2) ? r1 : r2);
            put_fifo(g->out2, (r1 > r2) ? r2 : r1);
        }
        stp_yield();
    }
}
```

```c
void main()
{
    fifo_t F1, F2, F3, F4;
    actorio_t sort_io;
    ...
}
Software Implementation: Multi-Thread Dynamic Schedules

```c
sort_io.in1 = &F1;     // connect queues to actor
sort_io.in2 = &F2;
sort_io.out1 = &F3;
sort_io.out2 = &F4;
stp_create(sort_actor, &sort_io); // create thread
stp_start();       // start system scheduler
```

Note, that as before, the execution rate of the actor code must be equal to the PASS firing rate in order to avoid unbounded growth of tokens.
Thus we use Solution 2 above, from the single-thread executive method described earlier.

Sequential Targets with Static Schedule

From the PASS analysis of an SDF graph, we know at least one solution for a feasible sequential schedule

This solution can be used to optimize the implementation in several ways.
Software Implementation: Sequential Targets with Static Schedule

- We can remove the firing rules since we know the exact sequential schedule. This will yield a small performance advantage (NOTE: it also prevents the use of dynamic scheduler).
- We can also determine an optimal interleaving of the actors to minimize the storage requirements for the queues.
- We can create a fully **inlined** version of the SDF graph which will allow us to get rid of the queues entirely.

Here, the relative firing rates of A, B, and C must be 4, 2, and 1 to yield a PASS.

```c
while(1) {
  // call A four times
  A(); A(); A(); A();
  // call B two times
  B(); B();
  // call C one time
  C();
}
```

**Fig. 2.21** System schedule for a multirate SDF graph
Software Implementation: Sequential Targets with Static Schedule

Given the interleaving schedule on the right, it can be seen that queue AB will carry a max of four tokens and queue BC will carry a max of two tokens in steady-state.

However, there is a BETTER interleaving schedule, i.e., by calling the actors in the sequence (A, A, B, A, A, B, C).

Here, the maximum # of tokens on any queue is reduced to two.

Therefore, the schedule determined using PASS is not necessarily the optimal (finding the optimal is an optimization problem).

As noted, implementing a truly static schedule means we do NOT need to check firing rules since the required tokens are guaranteed to be present.

Consider optimizing GCD using a single-thread SDF system with a static schedule.

Recall that a valid PASS requires firing each node once.
Software Implementation: Sequential Targets with Static Schedule

```c
void sort_actor(actorio_t *g) {
    int r1, r2;
    if ((fifo_size(g->in1) > 0) &&
        (fifo_size(g->in2) > 0)) {
        r1 = get_fifo(g->in1);
        r2 = get_fifo(g->in2);
        put_fifo(g->out1, (r1 > r2) ? r1 : r2);
        put_fifo(g->out2, (r1 > r2) ? r2 : r1);
    }
}

void diff_actor(actorio_t *g) {
    int r1, r2;
    if ((fifo_size(g->in1) > 0) &&
        (fifo_size(g->in2) > 0)) {
        r1 = get_fifo(g->in1);
        r2 = get_fifo(g->in2);
        put_fifo(g->out1, (r1 != r2) ? r1 - r2 : r1);
        put_fifo(g->out2, r2);
    }
}
```
Software Implementation: Sequential Targets with Static Schedule

```c
void main() {
  fifo_t F1, F2, F3, F4;
  actorio_t sort_io, diff_io;
  sort_io.in1 = &F1;
  sort_io.in2 = &F2;
  sort_io.out1 = &F3;
  sort_io.out2 = &F4;
  diff_io.in1 = &F3;
  diff_io.in2 = &F4;
  diff_io.out1 = &F1;
  diff_io.out2 = &F2;
  // initial tokens
  put_fifo(&F1, 16);
  put_fifo(&F1, 12);
  // system schedule
  while (1) {
    sort_actor(&sort_io);
    diff_actor(&diff_io);
  }
}
```
Software Implementation: Sequential Targets with Static Schedule

There are two simple optimizations that can be applied here:

- The firing schedule is static and fixed, and therefore the access order of queues is also fixed.

  This allows the queues to be optimized out and replaced with fixed variables.

For example, assume that we have determined that the access sequence on a particular FIFO queue will always be as follows:

```java
loop {
    ...
    F1.put(value1);
    F1.put(value2);
    ...
    .. = F1.get();
    .. = F1.get();
}
```
Software Implementation: Sequential Targets with Static Schedule

Given that only two positions of the FIFO F1 are occupied at a time, it can be replaced by two variables.

```plaintext
loop {
  ...
  r1 = value1;
  r2 = value2;
  ...
  .. = r1;
  .. = r2;
}
```

- A second optimization involves `inline`'ing actor code in the main program.
  In combination with the above optimization, this *eliminates* the firing rules and reduces the entire dataflow graph to a *single* function.

For the GCD example, each queue (F1, F2, F3, and F4) will contain no more than a single token, which means that each queue can be replaced by an *integer*. 
void main() {
    int f1, f2, f3, f4;
    // initial token
    f1 = 16;
    f2 = 12;
    // system schedule
    while (1) {
        // code for actor 1
        f3 = (f1 > f2) ? f1 : f2;
        f4 = (f1 > f2) ? f2 : f1;
        // code for actor 2
        f1 = (f3 != f4) ? f3 - f4 : f3;
        f2 = f4;
    }
}
Software Implementation: Sequential Targets with Static Schedule

These optimizations reduce the runtime of the program significantly. For example, we have dropped testing of the *firing rules* and manipulating the FIFOs.

This is possible here because we have determined a *valid PASS* for the initial data-flow system, and determined a *fixed schedule* to implement that PASS.

Note that we have traded some of the *runtime flexibility* for *improved efficiency*.