Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

In multi-threaded programming, each actor (implemented as a function) lives in a separate thread.

The threads are time-interleaved by a scheduler in single processor environments.

Systems in which threads voluntarily relinquish control back to the scheduler is referred to as cooperative multithreading.

Such a system can be implemented using two functions create() and yield() as shown.
Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

The scheduler can apply different strategies to schedule threads, 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>
```
Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

```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();
    }
}
```
Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

```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
```
Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

A multi-threaded version of the SDF scheduler, using the fft2 actor

```c
void fft2(actorio_t *g) {
    int a, b;
    while (1)
    {
        while (fifo_size(g->in[0]) >= 2)
        {
            a = get_fifo(g->in[0]);
            b = get_fifo(g->in[0]);
            put_fifo(g->out[0], a+b);
            put_fifo(g->out[0], a-b);
        }
        stp_yield();
    }
}
```
Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

```c
void main()
{
    fifo_t q1, q2, q3, q4;
    actorio_t fft2_io = {{&q2}, {&q3}};
    ...
    stp_create(fft2, &fft2_io); // create thread
    ...
    stp_start(); // start system scheduler
}
```

Note, as before, the `actor` code must enable convergence to the PASS firing rate (through while loops) in order to avoid queue overflow.
Mapping DFGs to Single Processors: 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:

- We can remove the firing rules since we know the exact sequential schedule. This yields only a small performance benefit.
- We can also determine an optimal interleaving of the actors to minimize the storage requirements for the queues.
- Finally, we can create a fully inlined version of the SDF graph which eliminates the queues altogether.

Here, the relative firing rates of A, B, and C must be 4, 2, and 1 to yield a PASS.
Mapping DFGs to Single Processors: Static Schedule

Given the interleaving schedule on the right, *queue* AB will need to store at most 4 *tokens* and *queue* BC at most 2 *tokens* in steady-state.

However, the interleaving schedule (A,A,B,A,A,B,C) is better because the maximum # of tokens on any *queue* is now 2.

Therefore, the schedule determined using PASS is **not** necessarily the optimal (in fact, finding the best schedule 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 the four-point FFT with a **single-thread** SDF system and a static schedule.

The 3 *actors*, *reorder*, *fft2* and *fft4mag*, have firing rates 1, 2 and 1, which yields a static, cyclic schedule [*reorder*, *fft2*, *fft2*, *fft4mag*]
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**

The *queue* access can be replaced as shown in the comments

```java
  loop {
    ...
    q1.put(value1); // replace with r1 = value1;
    q1.put(value2); // replace with r2 = value2;
    ...
    .. = q1.get(); // replace with .. = r1;
    .. = q1.get(); // replace with .. = r2;
  }
```
Software Implementation: Sequential Targets with Static Schedule

• A second optimization involves `inline`’ing the `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

```c
void dftsystem(int in0, in1, in2, in3,
               *out0, *out1, *out2, *out3) {
    int reorder_out0, reorder_out1;
    int reorder_out2, reorder_out3;
    int fft2_0_out0, fft2_0_out1;
    int fft2_0_out2, fft2_0_out3;
    int fft2_1_out0, fft2_1_out1;
    int fft2_1_out2, fft2_1_out3;
    int fft4mag_0_out0, fft4mag_0_out1;
    int fft4mag_0_out2, fft4mag_0_out3;

    // Reorder operation
    reorder_out0 = in0; reorder_out1 = in2;
    reorder_out2 = in1; reorder_out3 = in3;
```
Software Implementation: Sequential Targets with Static Schedule

// Two fft2 implementations
fft2_0_out0 = reorder_out0 + reorder_out1;
fft2_0_out1 = reorder_out0 - reorder_out1;
fft2_1_out0 = reorder_out2 + reorder_out3;
fft2_1_out1 = reorder_out2 - reorder_out3;

// fft4 implementation
fft4mag_out0 = (fft2_0_out0 + fft2_1_out0) *
                  (fft2_0_out0 + fft2_1_out0);
fft4mag_out1 = (fft2_0_out1 * fft2_0_out1) -
                  (fft2_1_out1 * fft2_1_out1);
fft4mag_out2 = (fft2_0_out0 - fft2_1_out0) *
                  (fft2_0_out0 - fft2_1_out0);
fft4mag_out3 = (fft2_0_out1 * fft2_0_out1) -
                  (fft2_1_out1 * fft2_1_out1);
Software Implementation: Sequential Targets with Static Schedule

These optimizations reduce the runtime of the program significantly.

For example, we have eliminated testing of the firing rules and calls to the queue and actor functions.

This is possible here because a valid PASS could be determined from the DFG, as well as fixed schedule to implement the PASS.

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