Internal Report 2011–02 May 2011

Universiteit Leiden
Opleiding Informatica

Code migration support
in Espam

Joris Ivo Huizer

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Introduction **3**
1.1 Code migration in general . . . 3
1.2 Embedded systems (MPSoCs) . . . 4
1.3 Uses for code migration in embedded systems . . . 4
1.4 Related Work . . . 5
1.5 Report organization . . . 5

2 Embedded system design using KPNgen and Espam **6**
2.1 The KPN Model . . . 6
2.2 KPNgen . . . 7
2.3 Espam . . . 8
2.4 Hardware used in our design . . . 9
2.4.1 Local memory . . . 9
2.4.2 Communication between processors . . . 10
2.5 Code structure in our design . . . 11
2.5.1 Thread management . . . 11

3 High-level discussion of code migration implementation **12**
3.1 Our design choices for code migration . . . 12
3.2 General steps taken in code migration . . . 14
3.3 The migration manager . . . 16

4 Low-level hardware implementation of communication for code migration **17**
4.1 The possibility of using the crossbar . . . 17
4.2 Better memory usage with shared memory blocks . . . 17
4.3 Sharing program memory between processors . . . 18
4.3.1 Prototyping requirements . . . 19

5 Low-level software implementation for code migration **20**
5.1 Stopping a thread . . . 20
5.2 Restarting a thread . . . 21
5.3 Corrections to the crossbar read primitive . . . 22
5.3.1 Threading-related changes to the crossbar read primitive 22
5.3.2 Fifo switching support in the crossbar primitives . . . 23
5.3.3 Remaining problems using the crossbar component . . . 23
5.4 Migration manager implementation . . . 25
5.4.1 Tracking hardware fifos . . . 25
5.4.2 Tracking thread-related fifo views . . . 25
5.4.3 Fifo management functions . . . 26
5.4.4 The migration algorithm . . . 28
5.4.5 Initialization work . . . 30

6 Performance 31
6.1 System wide timings 31
6.2 Migrating performance 32

7 Challenges and future work 34
7.1 Hanging situations with the crossbar 34
7.2 Improving on the time to migrate 34

8 Conclusion 35

9 References 35

1 Introduction

This chapter first explains code migration in 1.1 and embedded systems as background for the report in 1.2.
Afterwards related work will be presented in 1.4, and the organization of the rest of the report is presented in 1.5.

1.1 Code migration in general

Code migration has two terms: “code” and “migration”.

- “code” stands for a program, or a single function or procedure, which is executing.
- “migration” means, making the code available on another processor, and continuing running the code at the second processor.

The key part is that, before the migration, the code is executing at one location, and after the migration it is executing on another location, *as if nothing changed.*

When migrating code from location A to location B, these are requirements to make the migration work:

1. The function (program code) needs to be made available at location B.
2. The state of the function before migrating needs to be copied.
   Any resources the function depends on, need to be made available at location B.
   Examples of resources are:
   - local memory: (variables on a stack, in static memory, or in dynamic memory
   - libraries the code uses
   - various I/O resources, like files

   In general, making the required resources available at B may not be trivial, and may be a time-consuming task.

The goal behind code migration is usually to enhance performance. There are a number of different reasons why migration can improve performance, including;

- Migrating code from a processor which is under heavy load, to one which is lightly loaded.
- Migrating to a processor that has faster access to resources.
- Making unused resources (such as memory) available to a program by migrating it to the processor that has access to it.

An example might be moving the execution of code from a server to its client, thus allowing the server to move on to the next client faster.

Much more background on the topic of code migration in general can be found in [1].

1.2 Embedded systems (MPSoCs)

An embedded system is designed specifically for one (type of) task, for instance image or sound decoding or encoding.

The class of embedded systems of interest in this report is that of the multiprocessor system-on-chip (MPSoC for short).

An MPSoc is often intended for high-speed data handling such as multimedia. Such a problem has strict timing goals.
For example, with a video decoder as MPSoC, the system has to be fast enough to generate the images and sound to be displayed - but not generating so fast that big amounts of memory are needed to buffer images to be displayed later on.

This is in contrast with "high-performance computing", which is more focused on solving large amounts of computations, where there are no strict timing goals.

There is also a stronger focus on power usages and cost.
A practical example of this is that an mp3 player should be able to play for several hours – and it shouldn't be too expensive.

Compared to PCs, including multicore processors, an MPSoc is more finetuned in it's design for performance, focused on one (kind of) algorithm.
Instead of general (but less optimal) solutions, which are fine for a PC, more optimal solutions are chosen, specifically based on the (kind of) algorithm the system is targeted for.

1.3 Uses for code migration in embedded systems

There are a number of reasons to make code migration possible in embedded systems:

- The amount of work to be done on a different processor may vary, when running a number of different algorithms at the same time. By using code migration the load can be spread better over more processors.

- More than one dynamic programs can be implemented on MPSoCs. This means that a given function in an algorithm may take a different load at different times, or that the load may change because of a user request.
  Using code migration the load can be spread more evenly.

- In the most dynamic case, a user can install a new program and start it. To deal with the load changes on the different processors, code migration may be needed.

1.4 Related Work

The work was based on the existing Espam tool. This will be described in section 2.

Code migration in general has been discussed in detail in [1].

Migration in embedded systems has been documented in a number of reports. A feasibility study was done in [2]. It presents a possible design. Our design is similar in software, but the hardware implementation is different.

A research was done on code migration, to improve the power usage in an embedded system in [3]. In our work there has not been specific attention on power usage. Instead, our goal is using code migration for improved performance.

A research to implement efficient code migration in [4].

In these works code migration is also referred to as “task migration” or “process migration”.

1.5 Report organization

The structure of this report is as follows:

Section 2 describes the tools used to generate solutions for embedded systems, such as *KPNgen* and *Espam*.
Section 3 describes a high-level description of what is needed for code migration in an embedded systems environment.
Section 4 focuses on the hardware implementation details.
Section 5 focuses on the software implementation details, based on the descriptions in section 3.
Section 6 describes the test results of the possible influences of code migration.
Section 7 indicates challenges for future work.

2 Embedded system design using KPNgen and Espam

For embedded systems, the hardware requirements are much more restrictive than for a ordinary PC. The amount of memory available is limited, and also the processor used is slower and simpler. This is because the power usage must be kept sufficiently low.
In order to implement programs efficiently in embedded systems, parallelism is used. That is, independent steps in a program can run simultaneously on different processors.

The type of programs that we are investigating are streaming applications. This means that a continuous stream of input data is received by the program, which translates input into a corresponding output.
A typical example is translating input image data into a different format, or determining certain characteristics of it.

Designing an embedded system, that uses a number of processors, memory components, etc, is a complex task. Doing so by hand is a lengthy procedure and is error prone.

KPNgen and Espam are designed to help the programmer to generate an embedded system design from sequential code (C or C++ code).

2.1 The KPN Model

To describe flow of data in a program, a Kahn Process Network (`KPN`) model is generated. In a KPN, data dependencies in the program are made explicit.

The functions to execute are shown as nodes in a graph. Their inputs and outputs are the edges between the nodes.

In a KPN model, functions have these properties:

- A function cannot execute unless all input is available.
- Functions can execute as long as they have inputs.
  The current function can execute and send its outputs, even when a different function, that takes the generated output, also has to wait for other inputs.

The second point implies that a (theoretically unbounded) buffer is placed between all the functions to capture data generated, as long as a function is waiting for other data to execute.

A schematic example of a KPN can be seen in figure 1. As one can see, there are “source” nodes (that do not depend on other functions), there are “sink” nodes (that do not generate data for other functions), and there are other nodes in between.

While it is not the case in this example, a node can generate or read data for more than one node (in fact, any number that’s required for a given model).

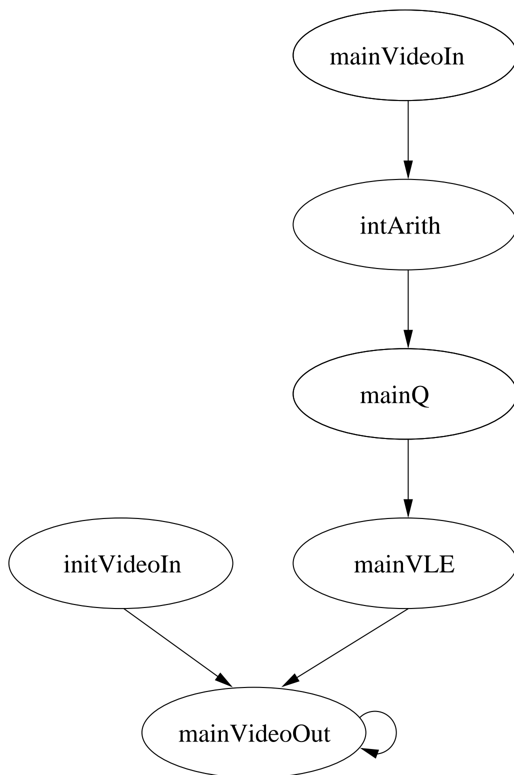

Figure 1: KPN example for MJPEG

2.2 KPNgen

KPNgen is a tool to generate a Kahn Process Network (KPN) model from sequential code, making explicit the data dependencies in the code.

For KPNgen to be able to generate a Kahn Process Network, the programmer is only required to rewrite the `main` function such that it is a series of loops, in which

1. input data is received (in array elements), calling an input function.
2. the data is translated using one or more function calls.
3. the generated output is handled, calling an output function.

There are no specific restrictions on the functions mentioned above, apart from the fact that no state (global variables) should be shared between different function being called.

Typically only the input and output functions have to be adapted in order to run on the embedded system.

Note that this is a Kahn Process Network, which is chosen such that no deadlocks will occur, as long as sufficient buffer sizes are used.

2.3 Espam

Espam takes three inputs:

- Platform specification.
  This is a high level description of the hardware to be used. It specifies the different processors used (`PPC` or `Microblaze`) including the amount of local memory, external memories connected, etc.

- Mapping specification.
  This describes on which processor each function in the KPN is mapped.

- KPN specification.
  The KPN model generated by KPNgen. See section 2.2.

Espam takes each of the inputs and generates both a hardware description for the embedded system design, and the software implementation of the specified KPN for each processor.

The generated files can be processed by a synthesizer. During testing, the Xilinx XST tool was used to generate a hardware description and to run it on an ADM-XRC-II FPGA.

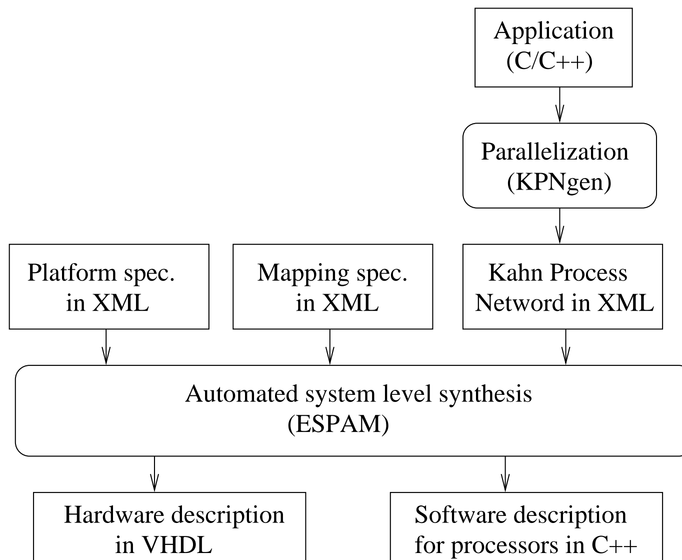

Figure 2: Schematic espam flow

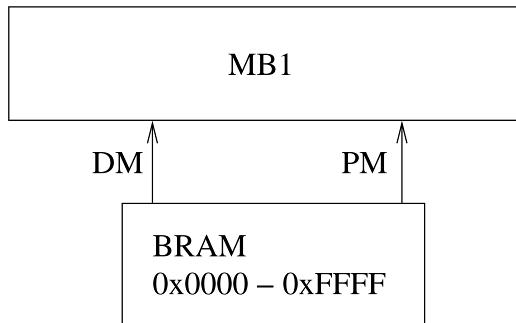

Figure 3: The common memory layout for local memory

2.4 Hardware used in our design

In an embedded system design, there are a number of important parameters to be decided on:

- How many processors to be used, and the types of them.
  We only use a Microblaze for the processor, as this is fully supported by the Xilinx synthesis tools.
- What kind of memory is used.
  This will be discussed below.

In embedded systems local memories are often used. This means that only one processor can access a given block of memory.
The reason to prefer this is that the best performance is possible, and that the power usage can be reduced in this design.

In a system with multiple processors, communication is needed to send data from one processor to the other.

2.4.1 Local memory

The memory blocks in our designs are designed for local memory per processor. Every memory block can have two controllers to refer to the memory block.

One of the memory controllers is used to referenced data (data memory, DM for short), the other is used to reference code (program memory, PM for short); This is schematically depicted in figure 3.

This is the common way to use the memory controllers but it is possible to assign the two controllers to different processors.
We use this in the implementation for migration.

2.4.2 Communication between processors

There are a number of possibilities to make communication between processors possible. Common ones are:

- Using shared memory with a bus
  This is the most 'general' and flexible solution. It isn't as efficient as other options.

- Using point-to-point connections and local memories.
  A point-to-point connection allows a processor to send data to a fifo of one processor.
  This is the most efficient solution, but also the most inflexible.

- Using a crossbar.
  With a crossbar, every processor can read from fifo's from every other processor. In both efficiency and flexibility this is in between the first two possibilities.

In our design we use a crossbar. We need to have the flexibility to choose to read from a fifo of any processor, because after a migration a thread will be generating it's data on a different processor than before.

In figure 4 a schematic picture of a crossbar is shown.
In it, each processor has access to a number of fifos to write to. This processor "owns" the fifo.
The crossbar component doesn't play any role in writing to fifos.

To be able to read from any fifo, the address to read needs to be indicated.
At times access to a given fifo is blocked.
If a fifo is owned by processor A, a read request from processor B will only be granted when no other processor is reading *any* fifo from A.
This happens not only when two processors try to read the same fifo.

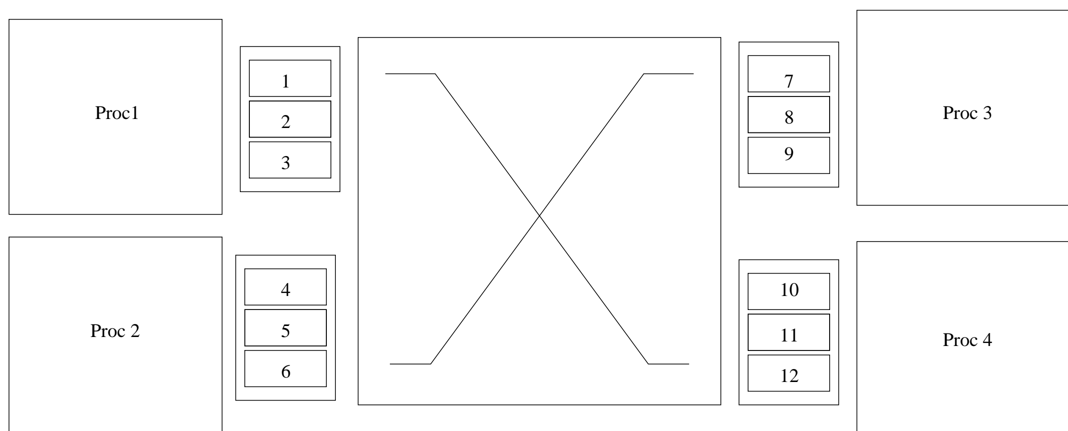

Figure 4: A schematic picture of a crossbar

This limitation is a result from the design from the crossbar.

2.5 Code structure in our design

Espam can be instructed to generated one thread per KPN node. Either a static schedule can be requested or a dynamic schedule using threading support.
In this discussion, threading support is chosen for every processor.

To understand what is needed to make code migration work, the structure of the code is described here.
The function executed in a thread, corresponds to one KPN node.
Figure 5 shows an example of such a function.

The important details are:

- The function only contains only one nested loop
- The general format looks like this:
  A number of reads, followed by one function (`_intArith` in the example) to be executed, and finally a number of writes.
- Iterators for the loop are only used for counting. They are not part of the calculation in the function.
- The embedded function is assumed not to have any state.
  This means that executing the function with the same inputs always generates the same output values.

2.5.1 Thread management

When a dynamic schedule is requested, it is implemented using threads. In the `main` thread, the required threads are started and afterwards the `main` thread will wait for each thread to finish running.

With migration support, we want the migration manager to be in charge over starting and exiting threads, so this design will be changed.

```
void *thread3(void *arg) {
    // Input Arguments
    tCH_2 in_0ND_2;

    // Output Arguments
    tCH_3 out_1ND_2;

      for( int c0 =  ceil1(0); c0 <=  floor1(7); c0 += 1 ) {
        for( int c1 =  ceil1(0); c1 <=  floor1(15); c1 += 1 ) {
          for( int c2 =  ceil1(0); c2 <=  floor1(7); c2 += 1 ) {

              readDynMF(ND_2_IG_1_CH_2, &in_0ND_2, sizeof(tCH_2)/4);

              _intArith(in_0ND_2, out_1ND_2);

              writeDynMF(ND_2_OG_3_CH_3, &out_1ND_2, sizeof(tCH_3)/4);

          } // for c2
        } // for c1
      } // for c0

  } // thread3
```

Figure 5: Example of (simplified) code for thread3

3 High-level discussion of code migration implementation

In this section an overview of the design is given, without discussing implementation details.

In sections 4 and 5 more details are discussed of the implementation itself.

3.1 Our design choices for code migration

As mentioned in section 1.1, when migrating code from location A to location B, these are requirements to make the migration work:

1. The function (program code) needs to be made available at location B.
2. The state of the function before migrating needs to be copied.
   Any resources the function depends on, need to be made available at location B.

For an embedded system environment, both steps need to be kept as simple, thus as efficient, as possible.

When migrating code from location A to location B, the following design choices are effects of this:

1. Having code available for B, is taken care of in the simplest possible way: All code that may be migrated from A to B, *is already available for B*. This means no copying of code is required at all.

   The situation is depicted in figure 6. The function *function2* is to be migrated to *Proc B*. The code is already available, but the state is not available.

   Having the code available on both A and B may seem like a waste of memory. Code may not actually be migrated in the end, but memory will still occupied to enable it. This is problem is reduced by sharing physical memory. This will be discussed later on

   Physically copying memory may be implemented as a separate step. Resource management as described here Could be reused without adaption in that scenario.

2. The only resource copied is a set of iterator values - just a few integers. In figure 5, these are the iterators `c0`, `c1` and `c2`.

   The migrated functions are written by Espam, so their inputs can be chosen to ease migration.
   Each function takes the set of iterators to indicate the initial state from which to start. Passing the current state of the function from A thus allows to continue at the right iteration.

   Migrating happens only at the start of an iteration, not in the middle of a loop.

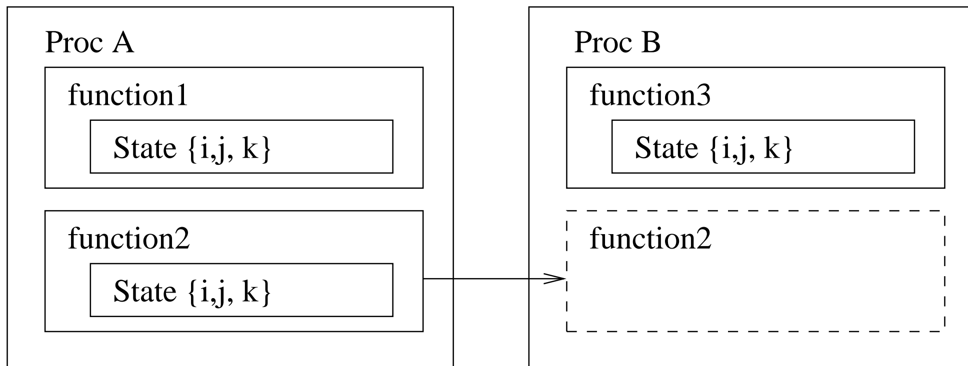

Figure 6: Situation (simplified) before migration: no state available

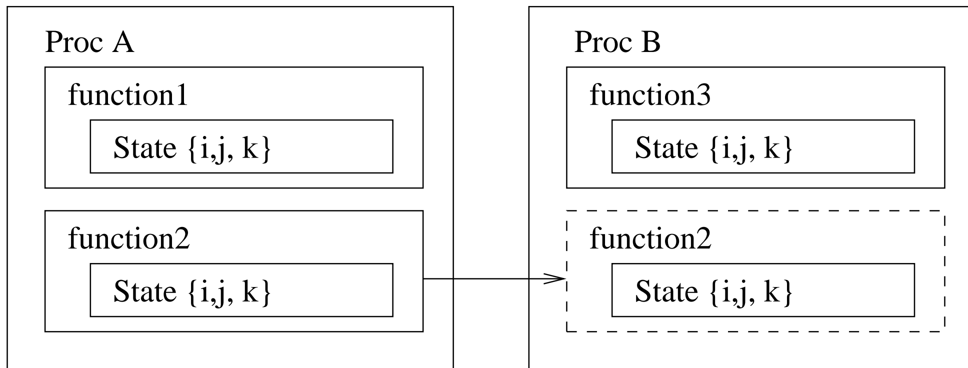

Figure 7: Situation (simplified) after migration: state has been made available

3.2 General steps taken in code migration

As described in section 3.1, no actual copying of code happens in the current implementation of code migration.

During a migration two changes are needed.
When migrating from processor A to processor B,

- A thread is stopped at A and restarted at B.
- Communication fifos are assigned on the B to the new thread.

Before the migration, the thread writes to a fifo to communicate its output to another KPN node.
This is shown in a schematic way in figure 8. Here, Thread 2 is depending on thread 1 to provide some data, which it will use in it's execution.

After the migration, the thread (thread 1) starts writing to another fifo to communicate.
However, any remaining tokens left in the previous fifo must be read before thread 2 switches to the new fifo. This situation is depicted in figure 9.

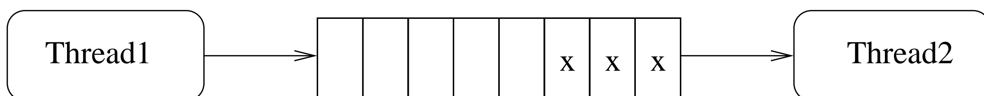

Figure 8: Fifo before migration

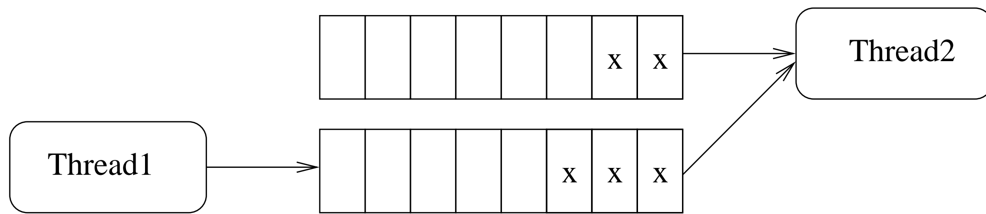

Figure 9: Fifo after migration

The steps to perform a code migration are the following:

1. New fifos are allocated on the destination processor.
   In case this fails, the migration is impossible so it is canceled.

2. The thread is stopped on the source.

3. The iterator state of the thread is stored.

4. The iterator state is copied to the destination processor.

5. The thread is started on the destination.

6. Successor threads keep reading on the old fifos until it is empty.

7. Successor threads start reading on the new fifos when the old one gets empty.

3.3 The migration manager

Assuming one processor is heavy loaded, and another is not, execution of a thread should be moved from the first processor to the other.
Normally, having multiple threads running on one processor, helps hiding the waiting delay for waiting on input data to arrive to a function. However, when a processor is heavy loaded, threads end up waiting a lot for the running thread to finish, even though their input data has arrived. If, on another processor, threads tend to be waiting on their input data, it would make sense to move one or more threads to that processor.

Figures 6 and 7 show indicate moving from `MB1` to `MB2`. This is actually a simplification of how the movement is really done. The real implementation of memory movement will be described in section 4

In the setup we use, there is a separate processor which serves as *migration manager*. The idea is that, when a processor A is too heavy-loaded, the migration manager will:

1. Choose a processor B to continue the thread
2. Select a thread T on A to stop
3. Select fifos for thread T on B.
   In case this fails migration is canceled.
4. Send a request for thread T to be stopped
5. Copy iterators of thread T
6. Make the iterators available on B
7. Send a request to B to continue execution of the thread.

Steps 1 and 2 still need to be implemented. Only ad-hoc choices have been used to test the other steps. The other steps have been implemented.

The problem of deciding to reduce the load on a processor, and selecting the right thread for this, is not solved in this work. The goal is simply to provide a means to migrate code (threads), assuming it has been decide when to migrate and where.

4 Low-level hardware implementation of communication for code migration

In the setup we have chosen the migration manager to dictate when a migration happens. It needs to communicate to the other processors in order to do this.

4.1 The possibility of using the crossbar

The simplest solution to make this communication possible would be to use the crossbar component for this. Extra fifos would be added to sent requests and data between the processors.

Using a crossbar, two fifos would be needed per client processor - one to write data to the manager processor, one to send data back.
Because of the crossbar design, a minimum of two memory blocks would be needed (Each 512 bytes or larger).

4.2 Better memory usage with shared memory blocks

As mentioned before in section 2.4.1, normally two memory controllers refer to the same memory. One is used to referenced data (data memory, DM for short), the other is used to reference code (program memory, PM for short); see figure 10.

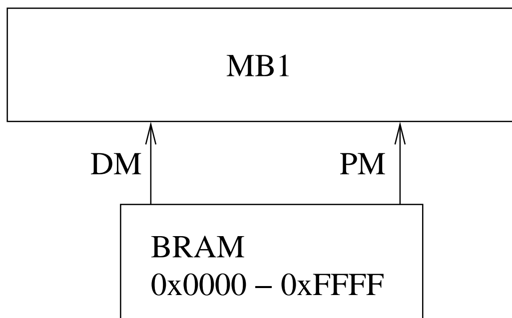

Figure 10: The usual memory layout

It is possible to connect one of the memory controllers to one processor, and the other to another one. This is shown in figure 11.

Every processor in the system is sharing a small amount of memory with the migration manager in this manner.

For this block of memory, both the migration manager and one other processor can refer to the same address, such that, when the first writes to an address (a variable), the other will see the updated value.

This way of sharing memory does not mean any overhead. The only requirement is that the software will not read data while it is being overwritten by the other processor; This has been guaranteed in our code.

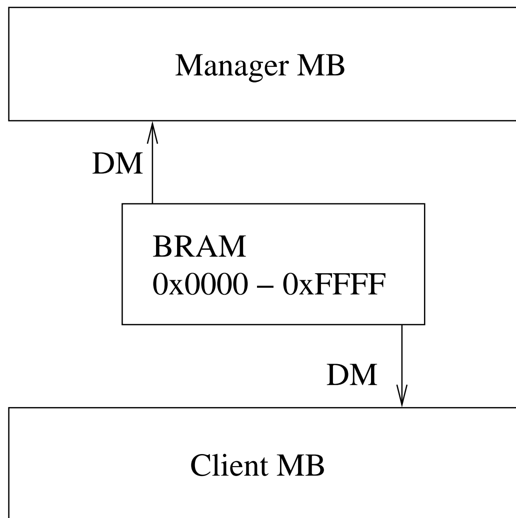

Figure 11: Specialized memory layout for communication between two processors

Note that, though this is simply using the different memory controllers in an unusual way. It is important to realize, there is no need to design new hardware for this.

In comparison to the crossbar solution, only half of the memory is required, because one block of 512 bytes of memory is used instead of two.

We use the crossbar component for communication between the threads that are executing the KPN functionality.

For this communication, we make use of the first in, first out behavior and we need the synchronization of reads. Doing this using shared blocks of memory would complicate the code.

4.3 Sharing program memory between processors

As mentioned in section 3.1, the implementation of the different threads is duplicated between the different processors that are setup for code migration.

Similar to sharing data memory between the manager and a client, memory is also shared between pairs of processors. This is depicted in figure 12.

The reason is that only one block of memory is used to hold the program data of two processors. In effect only half of the memory is used compared to the normal setup.

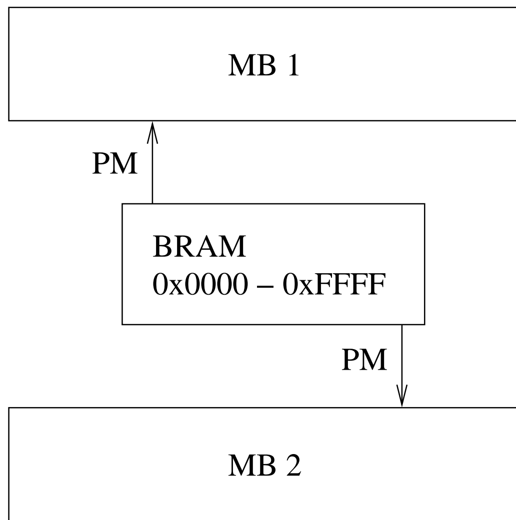

Figure 12: Sharing program memory between two processors

4.3.1 Prototyping requirements

While there is no inherit reason in the design, the XST synthesizer refuses a system in which the amount of program memory is less than the amount of data memory, when the data memory overlaps the program memory.

That is, during prototyping, extra memory is used simply to make the program memory completely overlap the data memory.

Depending on the amount of stack memory used (a fixed amount per thread), a higher amount of data memory is required - and thus also more program memory is allocated (even though the only reason is to overlap the data memory).

This is a limitation in the XST synthesizer - custom configuration may avoid the limitation but we have not researched what would need to be adapted.

5 Low-level software implementation for code migration

As described before in section 3.2, code migration is implemented using threads that may be stopped on one processor and restarted on another.

This section will give a more detailed explanation how this is done. A number of problems are solved:

- Stopping a thread
- Restarting a thread
- How the migration manager leads the migration

A thread has a number of possible stages ("runstates") it can be in:

- `idle`: The thread is not running
- `do_start`: The manager has requested to (re)start the thread.
- `active`: The thread is running
- `do_stop`: The manager has requested to stop the thread.
- `finished`: The thread has finished all its work.

When a thread is started, it receives a `state` pointer. It contains the `runstate`, a set of iterators, and a set of fifo addresses.

- The `runstate` is as described above,
- The iterators indicate which iteration to start with of the `KPN` node,
- The fifo addresses indicate what fifos to read from and where to write results to.
  These can be adapted by the migration manager during the run time of the thread.

5.1 Stopping a thread

There are two situations in which a thread stops execution:

- All iterations have been finished, so the function exits.
- The manager indicates to stop the thread, by updating the `runstate` to `do_stop`.

When the manager sets the `runstate` to `do_stop`, the following will happen in the running thread:

1. the function continues running until a new iteration starts.
2. the iterator structure is updated to contain the new iteration values.
3. the `runstate` is set to `idle`.
4. the function exits.

When all iterations have been finished, the `runstate` is instead set to `finished`.

5.2 Restarting a thread

The migration manager will indicate a thread has to be (re)started after it has finished migrating it from another processor. It will set the `runstate` of the corresponding thread to `do_start`.

On each processor, the `main` thread is periodically checking whether any thread is to be restarted in a polling fashion.

The polling algorithm is as follows:

- While any thread is running or is requested to (re)start,
  - Restart threads that are to be (re)started.
  - Perform a `yield` to let the other threads continue their work.
  - After all threads have run for a while (bounded by a maximum amount of time), the polling checks are redone.
- When no thread is either running or requested to (re)start, execution finishes.

After a `yield`, all the other threads start in turn, until they do a `yield`.

When the `main` thread starts, first it waits until the migration manager has done its initialization. Afterwards, it simply starts this polling loop with the `state` corresponding to the running processor.

It has been investigated whether it was possible to let the polling thread continue after every `yield` of an other thread, but this is not easy to do. There is the possibility to give higher priority to a given thread, but the `xilkernel` OS does not implement priority scheduling in the usual fashion (instead, threads of a lower priority won't execute at all).
To make this work, it would require constant priority readjusting, which reduces performance.

5.3 Corrections to the crossbar read primitive

As indicated in section 2.4.2 we use a crossbar component to allow for communication between KPN nodes.

The crossbar had not been used in combination with threads before. A number of changes were necessary to improve it's functionality.

First these changes will be presented, in 5.3.1, after that the changes specific for migration support are described in 5.3.2.
Then, remaining issues will be described in 5.3.3.

5.3.1 Threading-related changes to the crossbar read primitive

The crossbar components performs reads in a number of steps:

1. The port to read from is specified. This selects a certain fifo which corresponds to a certain processor.
2. For every word to be read (of four bytes), a number of steps is performed:
   (a) First a check is done whether the crossbar is ready to service the request. As long as it isn't, this check is repeated, in a polling fashion.
   (b) In case the fifo is empty, the port is deselected and a `yield` is done. A `yield` call tells the operating system to switch to another thread, pausing the current one.
   (c) If the fifo is not empty, a word is read from the fifo.
3. The specified port is deselected again.

The check of step 2a was originally not implemented.

The crossbar component isn't always able to immediately respond to a read request, because it checks each fifo corresponding to the processor in turn. It takes a number of clock ticks before a specific fifo is checked again to be service.

Without the check of 2a, when the crossbar couldn't service the request yet, this was misinterpreted to mean that the fifo was empty, and the code would `yield`.
A situation could arise that no progress would be made at all:

- A processor would request to read to a fifo, and find it couldn't be serviced yet.
- It would conclude the fifo to be empty and `yield`, switching to another thread.
- This thread would request a read to a second fifo, and find it again couldn't be serviced yet.
- It would conclude the second fifo to be empty, and `yield` again.
- It would keep yielding, either to a third thread or back to the first, without any read actually being done. Resulting in the system being stuck.

Another change is, that the complete code for the read primitive should be in a critical section. Otherwise the system could (time dependent) decide to do a yield, in the middle of the reading code, which would keep the fifo selected erroneously.
This would mean that no other fifo corresponding to that processor could be read afterwards, because the hardware requires a fifo to be deselected before allowing to select it again.

5.3.2 Fifo switching support in the crossbar primitives

To allow the fifo switching, the read and write primitives of the crossbar have to be adapted.

Originally, the read primitive took a constant corresponding to the fifo to be read. In the new setup a new address can be indicated, which is handled in this way:

- As long as there are tokens to be read from the current fifo, those are being read.
- When the fifo is empty, it is checked whether there is a new address to continue reading from.
  - If there is, the reading address is adjusted to continue reading on the new address
  - Otherwise, a `yield` is performed as before to allow other threads to continue while this thread waits for the required input.

In the case of the write primitive, the change is simpler. Because the fifo address only changes because of a migration, a thread is only to use an adjusted address when it starts, not in the middle of a series of writes.

Because of this, the first step of the crossbar writing primitive is to adjust to the updated address; afterwards, no checking is done for updates to the address.

5.3.3 Remaining problems using the crossbar component

The crossbar component had never been tested before in the context of threading.

The changes described in 5.3.1 were implemented to improve the reading primitive of the crossbar.

However, even with these changes implemented, the component is still not always functioning correctly. During tests, we have seen many times that, small code changes, unrelated to the crossbar component, could cause hangs (or appear to hang, being very much slower than before).
This even happened when adding debugging code that didn't cause any functional change.

This can be explained in the fact that changing code causes different timings, which then trigger a situation in which the reading will be slowed down significantly.

A hacky way around this issue is to try and add *more* delays with debugging code. This can undo this hangy effect. However, this is very error-prone.

This is obviously a big problem for implementing new code. Even while confident that our design should work, finding a system suddenly hanging is very frustrating. A lot of time has been lost in backtracking, trying to find the cause of these hangs.

It became clear this problem was still present after the mentioned changes in 5.3.1 were implemented.
However we found it would require quite some time to debug this problem in the crossbar component, so it was decided not to do so,

5.4 Migration manager implementation

As discussed in section 3.3, the migration manager is responsible for starting a migration. This includes ensuring enough fifos are available on a given processor to migrate a thread to it.

Also, when the system starts up it expects the manager to give start requests for threads on the different processors, according to an initial mapping.

In order to do this the manager needs to do some bookkeeping with the fifos.

- it needs to keep track of which fifos are used by a given thread.
- it needs to keep track of the number of available fifos on a given processor.
- it also needs to keep track of where each thread is mapped.

5.4.1 Tracking hardware fifos

To keep track of the available fifos per processor, a two-dimensional array is defined of `fifo_state`s, one dimension to indicate the processor, the other to indicate the fifo. It's declaration is shown in 13.

The three possible values for the `fifo_state` are:

- `fifo_free` meaning the hardware fifo is not used yet.
- `fifo_taken` meaning the hardware fifo is in use.
- `fifo_unavailable` meaning there is no hardware fifo corresponding to the given index.
  This is useful when, given a specific application, less fifos than `FIFO_COUNT` are needed on a specific processor.

`FIFO_COUNT` is the number of fifos used in all threads. It is the number of fifos required on one processor, if all threads would be mapped on that processor.

5.4.2 Tracking thread-related fifo views

Per thread the following information is used:

- the processor on which it (currently) is executing,
- the number of fifos it needs for it's communication,

```
enum fifo_state {fifo_free,fifo_taken,fifo_unavailable};
static fifo_state fifo_layout[2][FIFO_COUNT];
```

Figure 13: fifo_state usage in the migration manager

- an array of fifo mappings, with a `fifo_map_state` structure.

  The fifo mappings are needed during a migration. Two views of fifos need to be updated:

  - The hardware fifos on the destination processor need to be selected. This is held in a `fifo_state` reference.
  - The new fifo addresses need to be made visible to the client processors. To do this, the `logical` offset in the array of fifo addresses is used.
    This offset is chosen during initialization. The offset itself won't change afterwards.

An array of these fields is defined, as shown in figure 14 .

```
struct fifo_map_state
{
  unsigned logical;
  fifo_state *physical;
};

struct thread_map_state
{
  proc_select proc;
  unsigned int fifo_count;
  fifo_map_state fifo[FIFO_COUNT];
};

static thread_map_state thread[THREAD_COUNT];
```

Figure 14: thread-related data structures used in the migration manager

5.4.3 Fifo management functions

In order to perform a migration of a thread, a number of fifos need to be selected on the target processor, corresponding to the number of fifos used by the relevant thread.

To select a single fifo on a processor, the `selectFifo` function is used. It will return a number corresponding to the index in the `fifo_layout` (which is described in 5.4.1). The code for this function is shown in figure 15.

In case no more fifo is available on the relevant processor, the special `-1` value is returned. The caller is to respond accordingly.

The migration code deals with this failure in this way:

- selected fifos for the new thread on the target processor are deselected,
- the migration is canceled

```
int selectFifo(proc_select proc)
{
  for (unsigned i = 0; i < FIFO_COUNT; i++)
    if (fifo_layout[mb][i] == fifo_free)
    {
      fifo_layout[mb][i] = fifo_taken;
      return i;
    }
  return -1;
}
```

Figure 15: Function to select an available fifo

The migration code stores all selected fifos in an array.

The deselecting is done by a function `deselectFifos`. It takes an array of `fifo_states` to be deselected. They are simply reset to `fifo_free`, as shown in figure 16.

```
static void deselectFifos(fifo_state *fifo_set[],unsigned count)
{
  for (unsigned i = 0; i < count; i++)
    *fifo_set[i] = fifo_free;
}
```

Figure 16: Function to deselect an array of fifo

Only after all needed fifos have been selected successfully, they are copied to the `thread_map_state` structure. It is done this way to ensure that no data structures are changed, unless it is certain that the migration can be executed.

Updating the fifos corresponding to a thread is done using function `replaceFifosForThread`, as shown in figure 17.

```
static void replaceFifosForThread(unsigned threadID,fifo_state *fifo_set[])
{
  for (unsigned i = 0; i < thread[threadID].fifo_count; i++)
  {
    *thread[threadID].fifo[i].physical = fifo_free;
    thread[threadID].fifo[i].physical = fifo_set[i];
  }
}
```

Figure 17: Function to update fifos corresponding to a given thread

5.4.4 The migration algorithm

The actual migration algorithm is implemented in `migrateThread`. It takes a thread number and a target processor selector, as the declaration indicates in figure 18.
The function returns a boolean value, which is `true` when the requested migration has succeeded.

```
static bool migrateThread(unsigned threadID, proc_select to_proc);
```

Figure 18: The migrateThread declaration

The first step is to select the required number of fifos for the given thread on the indicated processor, using the helper functions described in 5.4.3.

The fifos are selected in a loop, and read and write addresses to communicate to the crossbar are calculated;

When selecting a fifo fails, the `deselectFifos` function is called and migration is canceled. The code is shown in figure 19.

```
struct
{
  unsigned read_address;
  unsigned write_address;
} migrated[FIFO_COUNT];
fifo_state *fifo_set[FIFO_COUNT];

for (unsigned i = 0; i < thread[threadID].fifo_count; i++)
{
  int fifo = selectFifo(to_proc);

  if (fifo == -1)
  {
    deselectFifos(fifo_set, i);
    return false;
  }
  fifo_set[i] = &fifo_layout[to_proc][fifo];

  migrated[i].read_address = FIFOreadAddr(to_proc,fifo);
  migrated[i].write_address = FIFOwriteAddr(fifo);
}
replaceFifosForThread(threadID, fifo_set);
```

Figure 19: The fifo selecting loop

The functions `FIFOreadAddr` and `FIFOwriteAddr` simply map the logical fifo number to an address in the crossbar. Their implementation is dictated by that

of the crossbar component.

After all the fifos have been preselected, the actual migration starts. This is indicated by setting a `configuring` boolean to true.

Then, the new fifo addresses are made available to the client processors, as in figure 20.
Here, `first` and `second` correspond to the different client processors.

```
for (unsigned i = 0; i < thread[threadID].fifo_count; i++)
{
  unsigned j = thread[threadID].fifo[i].logical;
  first->fifo[j].read.next =
    second->fifo[j].read.next =
    migrated[i].read_address;

  first->fifo[j].write.next =
    second->fifo[j].write.next =
    migrated[i].write_address;
}
```

Figure 20: Loop to make the new fifos available to the client

The following step is to make the thread stop on the original processor. This is shown in 21.
While simple in concept, this part can take quite some time.

```
from->state[threadID].runstate = do_stop;
while(from->state[threadID].runstate != idle);
```

Figure 21: Making a thread stop it's execution

The next step is to copy the iterators from the original processor to the target.
The final part is to restart the thread on the target processor. This is simply done by requesting the thread to be started.
The the state `configuring` is also marked `false` again.
The final work is shown in 22.

```
to->state[threadID].runstate = do_start;
thread[threadID].proc = to_proc;
```

Figure 22: Restarting a thread on it's target processor

5.4.5 Initialization work

When the migration manager starts, it performs the following tasks:

- It indicates to the client processors which thread to start.
  This is based on the mapping indicated by the `thread` array. This is the array described in 5.4.2.

- It allocates (selects) the hardware fifos needed for each thread to be able to execute.
  Also, the `logical` offset in the `fifo` array is chosen here. (which won't change afterwards).
  This `fifo` array is shared between the manager and the client processors.

After these tasks are finished for all threads, the migration manager indicates to each client processor that execution can start, by setting the `initialised` flag to `true`.

6 Performance

6.1 System wide timings

In table 1, the performance influence of code migration is measured , in percentages. These are numbers from actual runs on an FPGA.

- `basis` is a system running MJPEG in a configuration with two processors (and a third processor that doesn't execute any code) without any setup for migration.
- `migratable` is a system running in the same configuration, adapted to allow migration. However, no actual migration happens.
- `migrating once` is the same system, migrating one function
- `migrating twice` is again the same system, migrating yet another function

Name	System time		MB_1 time		MB_2 time	
	Ticks	Percent	Ticks	Percent	Ticks	Percent
basis	531 M	100.0 %	481 M	100.0 %	481 M	100.0 %
migratable	532 M	100.2 %	482 M	100.2 %	482 M	100.2 %
migrating once	512 M	96.5 %	462 M	96.1 %	462 M	96.1 %
migrating twice	572 M	107.7 %	520 M	108.2 %	522 M	108.5 %

Table 1: Performance as measured with MJPEG example

From these results, the following conclusions can be drawn:

- The influence of code changes to allow for migration is very small, as can be seen with the performance of `migratable`. In this example, all code changes to allow for migrating has been put in place, but they are not used. This effectively measures the overhead by code changes.
- The case of `migrating once` was an effective migration. It shows that the (additional) small overhead of migrating can be worth its investment, when choosing correctly which thread to migrate.
- The case of `migrating twice` shows that making an ill choice in migrating can have a negative influence on the overall performance of the system.

The differences in table 1 are rather small, but they might easily become (much) bigger when migrating in a more optimal way.

These results are mostly a proof of concept.

6.2 Migrating performance

From a request to migrate a thread, till the end of the migration, there are a number of steps taken:

1. The manager selects the required number of fifos to be used (T_{req}).
2. The manager makes the request for the thread to finish.
   The thread continues work until it finishes one iteration. ($T_{markstop}$).
3. The thread copies iterator values and stops execution. (T_{stop}).
4. The manager copies the iterator values to the target processor. (T_{copy}).
5. The manager marks the thread to be restarted on the target processor. ($T_{markstart}$).
6. The target processor restarts the thread (T_{start}).

In general, it is preferred that a migration is executed as fast as possible. In the chosen design, there are a number of possibly long waits before a migration is finished.

We will discuss each interval in turn next.

- The time between T_{req} and $T_{markstop}$ is short.
- The time between $T_{markstop}$ and T_{stop} can be considerable.
  In the worst case, the stop marking is done just after the thread starts a new iteration. In this case. a complete execution of the `KPN` node functionality is executed before the thread status is checked again.
  That means every read, the execution of the function itself, and every write, will be executed before the migration finishes.
  Worse, because the thread may block because required inputs are not available, or because an output fifo is full, other threads may continue their work before the thread can finish it's iteration.
  Usually this is seen as a good thing: time blocking for I/O is used to let other threads continue. But it causes the time to migrate to take more time.
- The time between T_{stop} and T_{copy} is short. Only a few bytes are copied, and the manager immediately responds when the thread's state is updated to `idle`.
- The time between T_{copy} and $T_{markstart}$ is also short. Again, this only copies a few bytes.
- The time between $T_{markstart}$ and T_{start} can be considerable. However, unlike before, here an exact worst case can be calculated.
  The worst case scenario is when the `thread_main` loop has just finished. In that case, first all the `KPN` nodes continue, one after another, before the

thread_main loop checks for restart requests again.
This takes the most time in the case that all threads have their inputs available, and are executing the KPN node functionality for a lengthy amount of time.

Using xilkernel, a maximum execution time per thread can be indicated. Each thread is stopped after a fixed (and configurable) amount of time, indicated by the systmr_interval flag. The minimum interval is 1 millisecond.
This means that with N threads running, the worst delay would be N milliseconds.

A possible way to shorten the second delay, between $T_{markstop}$ and T_{stop}, would be to allow a thread to stop after its reading phase or after its executing phase.

The problem with this idea is that the internal data used in the KPN will need to be stored. As the amount of bytes used, is defined by the application, this is less general than our current solution.

This possible solution has not been implemented.

7 Challenges and future work

7.1 Hanging situations with the crossbar

Even after improving the reading primitive of the crossbar component, a system with a crossbar still can get in a situation of hanging.

The cause of this is currently unknown.

A workaround is to add some extra delays (writing to volatile memory). This tends to cause the system to run successfully again.

7.2 Improving on the time to migrate

As described in section 6.2, a migration may take some time to finish because of the way we embedded the migration code in the `Espam` framework.

This could be investigated to improve the time to migrate.

8 Conclusion

A method was researched and developed to make code migration possible within the Espam framework.

A simple migration scheme was implemented, which should work in general. The performance of the migration scheme may need to be improved, though this could mean sacrificing simplicity (possibly more hand-tuning needed)

More researching is needed to make the crossbar component more reliable in the context of threading.

9 References

[1] Dejan S. Milojičić Fred Douglis, Yves Paindaveine, Richard Wheeler and Songnian Zhou, “Process Migration”, ACM Computing Surveys. Vol. 32, No. 3, September 2000, pp. 241299.

[2] Stefano Bertozzi, Andea Acquaviva, Davide Bertozzi, and Antonio Poggiali, “Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study”

[3] Stefano Bertozzi, Andea Acquaviva, Davide Bertozzi, and Antonio Poggiali, “Impact of Task Migration on Streaming Multimedia for Embedded Multiprocessors: A Quantitative Evaluation”

[4] V. Nollet, P. Avasare, J-Y. Mignolet and D. Verkest, “Low Cost Task Migration Initiation in a Heterogeneous MP-SoC”