Design-Space Exploration with MPARM

1 Introduction

This document is intended to be a guide through the design of embedded systems implemented on multiprocessor platforms. It is supposed to be studied as a preparation for the lab assignments in the courses TDTS07 and TDDI08. You will learn how to customize the hardware platform such that the resulting system is optimized for a given functionality. Most of the assignments will use as an example software application the GSM codec. You will work on a realistic multiprocessor system, composed of several ARM processor cores, private and shared memories, interconnected by an Advanced Microcontroller Bus Architecture (AMBA) bus. This guide will present the hardware platform and the software development process for it, as well as possible design alternatives.

2 MPARM Platform Description

In this section we will introduce the hardware and the software infrastructure.

2.1 Hardware Platform

The target architecture is a general template for a distributed multiprocessor system on a chip (MPSoC). The platform consists of computation cores, a communication bus, private memories (one for each processor) and of a shared memory for interprocessor communication (see Figure 1). The multiprocessor platform is homogeneous and consists of ARM7 cores with instruction and data caches and tightly coupled software-controlled scratchpad memories.

The virtual platform environment provides power statistics for ARM cores, caches, on-chip memories and bus, leveraging technology-homogeneous power models for a 0.13 μm technology provided by STMicroelectronics.

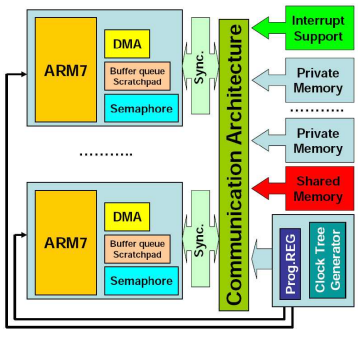


Figure 1: MPARM platform.

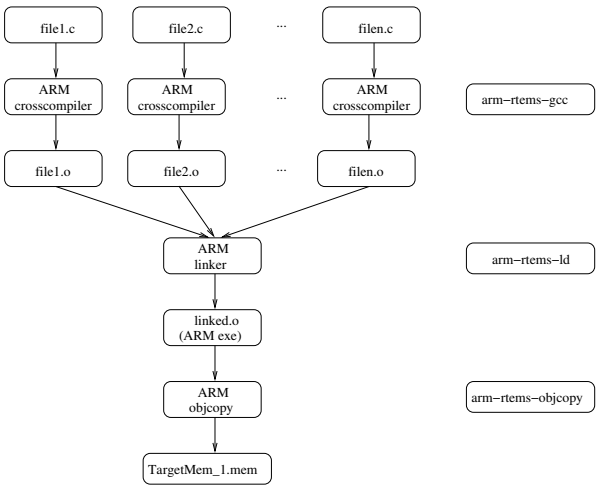


Figure 2: Cross-compilation of C programs.

2.2 Software Platform

In this section we will describe how to implement and run an application in MPARM (a step-by-step example is given in Section 6). Since MPARM is a cycle-accurate simulator, any binary executable (actually a memory image obtained from the binary) for an ARM7 processor will run out of the box.

In order to be able to run any program, it must be compiled first. In order to make the compilation process easier, especially for large applications with many source files, we will use a Makefile. (If you are not familiar with the `make` tool, refer to `http://www.gnu.org/software/make`.) The actions executed by the Makefile are captured by Figure 2.

Note that we do not use the standard GNU compiler toolchain available in many GNU/Linux distributions. When developing applications for embedded systems, the code is written and debugged on desktop computers. The reasons are obvious: imagine writing an MPEG decoder on your mobile phone! The compilers shipped with desktop systems generate code for these very systems. However, embedded systems typically have different processors (such as ARM and PowerPC) and operating systems (such as RTEMS and RT Linux). Consequently, any code compiled with a desktop compiler will not run on the target embedded platform. In order to compile the application code for an ARM7 processor, a special compiler should be used: a cross-compiler. A cross-compiler is a compiler that is running on one system and generating binaries for another system. In the case of this lab, the cross-compiler runs on an i386 platform under GNU/Linux and generates code for ARM7 processors. An example Makefile is given in Section 6.2

3 Design-Space Exploration for Energy Minimization

3.1 Introduction

One of the advantages of having a simulation platform is that it can be integrated early in the design flow. Traditionally, the final validation is performed at the end of the design flow when the first prototype is available, which is illustrated in Figure 3. In order to obtain a correct and efficient product, the system model must be very accurate. If a cycle-accurate simulator of the target platform is available at the early design stages, the system model may be complemented or even replaced by this simulator, as shown in Figure 3. In this way, we would gain accuracy at the expense of using a slow simulator as opposed to fast models.

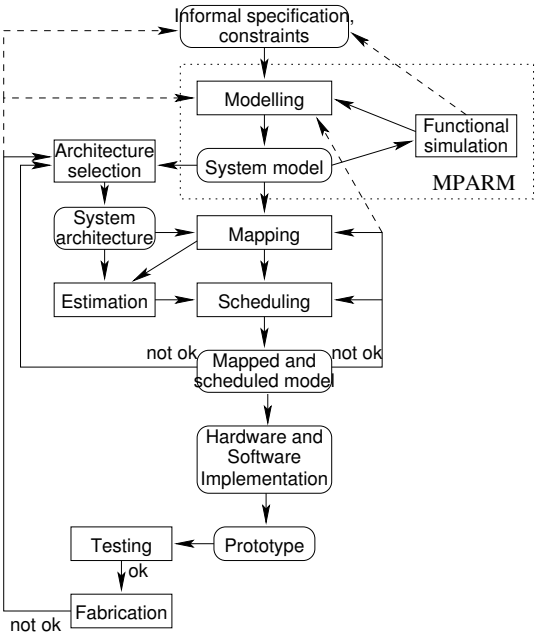


Figure 3: System design flow.

In this section, we will illustrate the usage of the MPARM simulation platform for a design-space exploration problem. The goal is to optimize a GSM encoder/decoder application, given as a source code written in C. This application will run on an instance of the MPARM platform with one ARM7 processor. The size and associativity of the cache as well the frequency of the processor are variable. We will try to find an assignment of these three quantities (cache size, cache associativity, and processor frequency) such that the energy consumed by the system is optimized.

3.2 Minimizing Energy by Frequency Scaling

The frequency of the processor is a key parameter, which affects the speed of the application and the power consumption of the platform. Typically, the power consumption is proportional to the frequency and to the square of supply voltage:

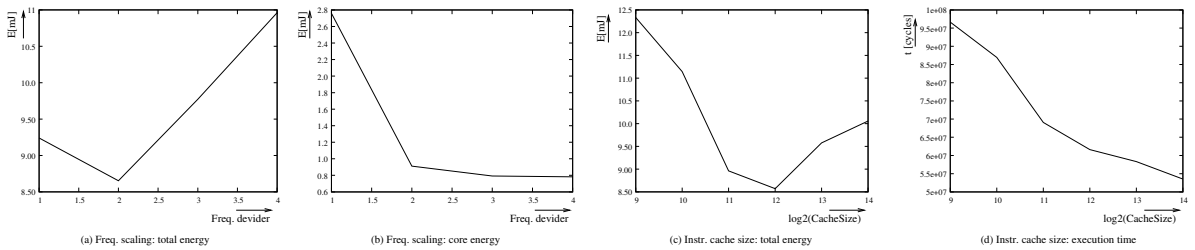
$$P = f \cdot C_{eff} \cdot V_{dd}^2$$

In most systems, choosing a lower frequency implies using a lower supply voltage. So the power consumed by the processor is reduced cubically. Actually, for battery-powered systems, we are interested in the energy consumption. The energy is defined as the product of the power over time:

$$E = P \cdot t = f \cdot C_{eff} \cdot V_{dd}^2 \cdot \frac{NC}{f} = C_{eff} \cdot V_{dd}^2 \cdot NC$$

where t is the execution time in seconds, and NC is the execution time expressed in number of clock cycles. So, if we scaled down only the frequency without scaling down the supply voltage, although the power would be reduced, the energy would remain constant.

It is important to note that by scaling down the frequency (and thus the supply voltage), only the energy consumed by the processor is reduced. The power consumption of the rest of the components other than the CPU cores (such as caches and memories) remains constant. We could express this



(a) Freq. scaling: total energy (b) Freq. scaling: core energy (c) Instr. cache size: total energy (d) Instr. cache size: execution time

Figure 4: MP3 decoder design alternatives.

mathematically as follows:

$$E_{system} = \sum_{CPU_i} P_{CPU_i}(f) \cdot t(f) + \sum_{nonCPU_i} P_{nonCPU_i} \cdot t(f)$$

The energy consumed by a processor decreases as its frequency decreases. However, the energy consumed by the other components increases as the processor frequency decreases: the power of the other components does not depend on the processor frequency while the execution time grows if the processor frequency gets reduced. To conclude, the total energy of the system is achieved at an optimal frequency, depending on the application. Scaling the frequency below that value will result in an increase in the energy due to components other than processor cores (caches, memories, and bus).

We have studied the frequency scaling for an MP3 decoder running on one processor in MPARM. The results are plotted in Figure 4(a). On the X axis, we have varied the frequency from 200 MHz to 50 MHz in four steps: 200, 100, 66, and 50 MHz. The values for the total energy obtained for each frequency are represented on the Y axis. Clearly, running the MP3 decoder at 100 MHz provides the best energy. Scaling down the frequency below this value results in an increase in the total energy. Figure 4(b) presents the energy of the processor core when the frequency is scaled. For the processor core, the lowest frequency provides the lowest energy.

3.3 Cache Influence on Energy Consumption

Instruction and data caches are used to improve the performance of applications by achieving a faster execution time. From the power perspective, the cache hardware cannot be neglected, which is particularly important for embedded systems. The power consumption of the cache and its efficiency depend on the associativity (direct mapped, k-associative, and fully associative) and size. Intuitively, a large cache is more efficient. However, it will consume more power. An interesting aspect is the interaction in terms of energy between the cache and the CPU. A larger cache, while consuming more power, could reduce the execution time of the application and, thus, the energy consumed by the processor.

Similar to frequency scaling, we have performed experiments on the MP3 decoder, monitoring the influence of the size of the instruction cache on the total energy of the system and speed of the application. The results are presented in Figure 4(c) and (d). We have considered a split instruction and data cache. Clearly, a big instruction cache translates in a shorter execution time. This can be observed in Figure 4(d). The more interesting result is the one on energy, presented in Figure 4(c), showing that very small instruction caches are bad for energy as well. Big caches provide a good execution time at the expense of a higher energy. We notice that the energy curve has an optimum when the instruction cache size is 4096 bytes.

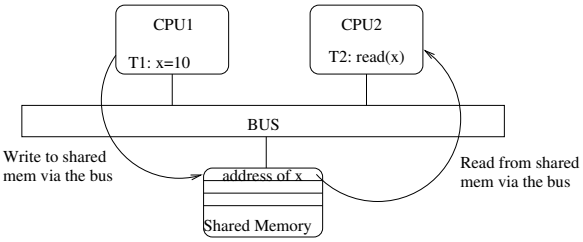


Figure 5: Interprocessor communication via shared memory.

4 Communication in Multiprocessor Systems

An important aspect during the design of multiprocessor systems is the exchange of data between tasks running on different processors. In this section, we will present two communication alternatives. Another important issue addressed in this section, which is tightly coupled with the communication, is the synchronization.

4.1 Shared Memory Based Communication

Passing the data through the shared memory is the most common way of interprocessor communication. An example is given in Figure 5. Let us assume that the task running on CPU2 (consumer) needs some data that was produced by the task running on CPU1 (producer). The easiest way to communicate these data is to store it on a common resource (the shared memory). An interesting question is raised by this implementation: What happens if the consumer starts reading from the shared memory before the producer has finished to write? We will answer this question in the next section.

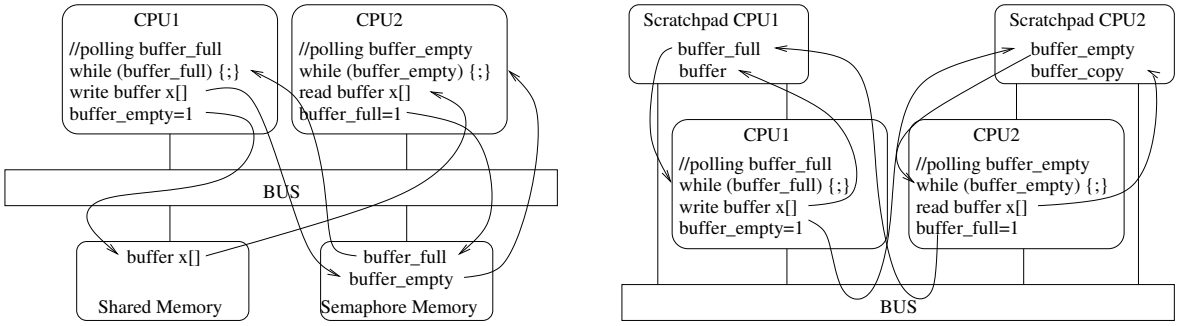
4.2 Synchronization

In this section, we will describe the synchronization mechanism present in MPARM. We start by coming back to the question from the end of the previous section. In the previous example, the value of variable `x` from task T2 is undefined. Task T2 reads values from a certain memory address. When running that example several times, the values displayed in task T2 could be different. The correct functionality would be achieved if task T2 would wait until task T1 finishes updating the shared memory with the correct `x` values. The MPARM platform provides a set of hardware semaphores (that can be accessed via the software using the `*lock` variable), as well as a simple software API with the functions `WAIT(int lock_id)` and `SIGNAL(int lock_id)`. The semantic of `WAIT` and `SIGNAL`, operating both on the same hardware lock identified by the parameter `lock_id`, is the following: the task that calls `WAIT(lock_id)` will stop its execution and wait until another task running on a different processor calls `SIGNAL(lock_id)`. A code example that uses shared memory based communication is given in Section 6.6.2

4.3 Distributed Message Passing

We have introduced in Section 4.1 a widely used interprocessor communication method, namely the usage of shared memory. In order to safely access the shared memory, we have shown that semaphores must be used. We will now present another approach to interprocessor communication.

Let us look closer at the shared-memory-based communication. The main drawback here is the bus traffic generated due to the synchronization. MPARM has a hardware semaphore device implemented as a small memory that can be accessed by the processors via the bus. Such an architecture is depicted



(a) Communication via shared memory

(b) Communication via distributed queues

Figure 6: Shared memory vs. distributed queues.

in Figure 6(a). T1 running on CPU1 produces some data and writes it in a shared memory buffer that is later read by the task running on CPU2. We assume that the buffer size is 1 and the two tasks repeatedly produce and respectively consume these data. Before T1 writes a new value to the buffer, it has to make sure that T2 has read the previous value. A semaphore `buffer_full` is used for this purpose. On the other hand, before reading, T2 has to wait until new data are available in the buffer, and uses the semaphore `buffer_empty` to do that. Waiting on a semaphore is performed by repeatedly checking the semaphore (polling) until is becomes free, potentially causing a lot of traffic on the bus. An obvious reason why this is bad is power consumption: this traffic on the bus consumes energy from the battery. Another system parameter that is indirectly affected by this polling is the execution time of the application. The semaphore polling generates traffic on the bus, interfering, for example, with the traffic due the cache misses (in case of a cache miss, the information is retrieved in the cache from the main memory). In this way, the miss penalty increases and the processor is stalled waiting for the cache refills to finish.

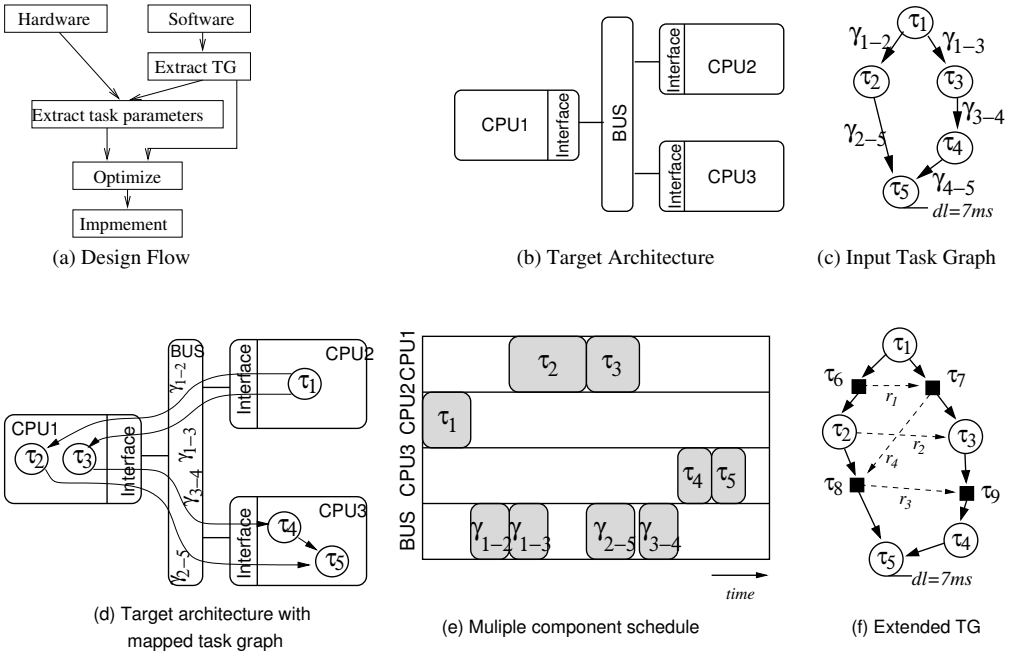
Shared-memory-based communication is not the only available design choice. MPARM provides a direct processor-to-processor communication infrastructure with distributed semaphores. This alternative is depicted in Figure 6(b). Instead of allocating the semaphores on a shared hardware, the two semaphores are each allocated on a scratchpad memory directly connected to each processor. In order for a processor to access data on its scratchpad memory, it does not need to go via the bus; the scratchpad is small and fast compared to a RAM. In our example, `buffer_full` is allocated on `Scratchpad_CPU1`, while `buffer_empty` is allocated on `Scratchpad_CPU2`. Consequently, when tasks T1 and T2 are doing busy waiting, they are not generating any bus traffic. The scratchpad memories have to be connected to the bus, in order to be accessible to the other processors. In this way, when, for example, T2 running on CPU2 releases the semaphore `buffer_full`, it can access the corresponding location on the `Scratchpad_CPU1`.

Code examples on how to use the two methods of interprocessor communication in MPARM are given in Section 6.6.

5 Mapping and Scheduling in Multiprocessor Systems (TDTS07 only)

Note that this section is required for TDTS07 but is only optional for TDDI08.

The purpose of this section is to illustrate two key design issues related to the implementation of software applications on multiprocessor systems: the mapping and scheduling of the software tasks. This section is based on the GSM voice codec implemented on the MPARM platform.



(a) Design Flow (b) Target Architecture (c) Input Task Graph

(d) Target architecture with mapped task graph (e) Muliple component schedule (f) Extended TG

Figure 7: Application synthesis on a target architecture.

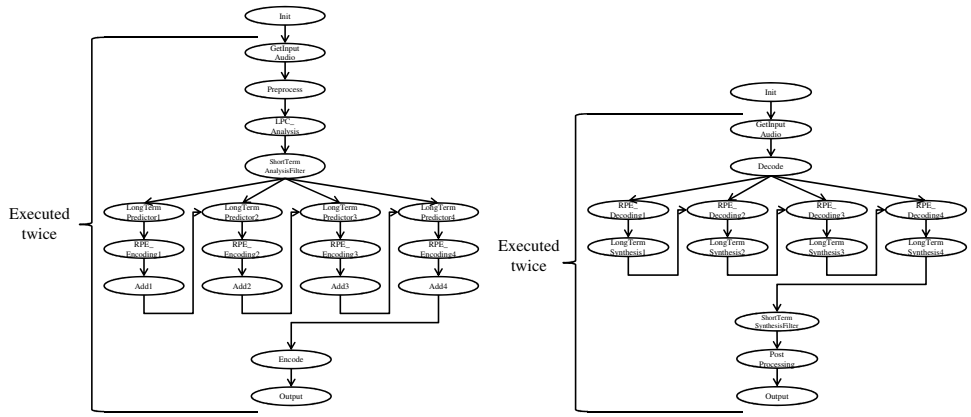
5.1 Introduction to Mapping and Scheduling

During the design of embedded systems composed of software applications and a multiprocessor hardware platform, an important decision is how to assign the various software components (tasks) to the processors (referred to as mapping) and in which order to execute the software tasks running on each of the processors (referred to as scheduling). This decision can be made at the design time, because embedded systems have a dedicated (known) functionality as opposed to general purpose computers, which work with a variety of (unknown) applications. In the following, we will introduce a simple embedded application design flow (see Figure 7(a)), emphasizing certain aspects related to the mapping and scheduling of the application on the target hardware platform.

The input of our design flow is the software application, written in C, and the hardware platform. The specification of the application consists of code that can be compiled and executed on a single processor. For example, in case of the GSM codec, the code is composed of two distinct parts, the encoder and the decoder, that can be independently compiled and executed on a single processor instance of MPARM. The first step of the design flow is to extract a task graph from the application.

The functionality of applications can be captured by task graphs, $G(\mathcal{T}, \mathcal{C})$. An example is depicted in Figure 7(c). Nodes $\tau \in \mathcal{T}$ in these directed acyclic graphs represent computational tasks while edges $\gamma \in \mathcal{C}$ indicate data dependencies between these tasks (i.e., communications).

Such a task graph is extracted (manually or partially/fully automatically) from the source code of the application. The task graph captures two important aspects: dependencies (tasks that must be executed in a certain order) and parallelism (tasks that can be executed in parallel, potentially on different processors). Partitioning the application code into tasks is not an exact science. For example, the granularity of the tasks can be finer or coarse. A fine granularity can offer flexibility for various optimizations. There are, however, drawbacks in having a fine granularity: the overhead due to context switches, the amount of communication, or the complexity of analyzing and performing optimizations can be very high. For example, Figure 8 shows potential task graphs of the GSM voice codec. Another possible pair of task graphs for the decoder and encoder with a finer granularity is given in Figure 9



(a) GSM Encoder (b) GSM Decoder

Figure 8: GSM voice codec task graph.

and 10. Note that, in case of the encoder, there are 16 tasks in Figure 8 and 53 tasks in Figure 10.

Given a task graph (Figure 7(c)) and a target hardware platform (Figure 7(b)), the designer has to map and schedule the tasks on the processors. Mapping is the step in which the tasks are assigned for execution to the processors and the communications to the bus(es). We have depicted a possible mapping for the task graph from Figure 7(c) in Figure 7(d). The next step is to compute a schedule for the system, i.e., to decide in which order to run the tasks mapped on the same processor. One important set of constraints that have to be enforced during mapping and scheduling are the precedence constraints given by the dependencies in the task graph. A potential schedule is depicted in Figure 7(e). Note that task τ_2 starts only after task τ_1 and the communication γ_{1-2} has finished. Most embedded applications must also respect the real-time constraints such as the application deadline.

Computing the task mapping and schedule is, in general, a complex problem, and exact solutions usually belong to the NP class. Nevertheless, there exist tools that perform mapping and scheduling with different objectives. For example, given a task graph with dependencies, each task being characterized by an execution time and a deadline, there exist algorithms for computing a mapping and a schedule such that the precedence constraints and deadlines are satisfied. If the objective is the energy minimization, besides the precedence and deadline constraints, there exist algorithms that compute the mapping, schedule, and task speed that provide the minimum energy. The details of these mapping and scheduling algorithms are beyond the scope of this lab. The purpose of the next assignment is to study, using the MPARM cycle-accurate simulator, the impact of the mapping and schedule on the timing of the application and on the energy consumed by the system.

5.2 Mapping and Scheduling in MPARM

Detailed examples on how to map tasks on the MPARM platform are given in Section 6.4. As far as the schedule is concerned, this is simply given by the order of execution on each processor.

6 MPARM Tutorial

6.1 Writing a Program for MPARM and Using the Lightweight Functions

We will start with a very simple C application, show its particularities, and then compile and execute it in the simulator.

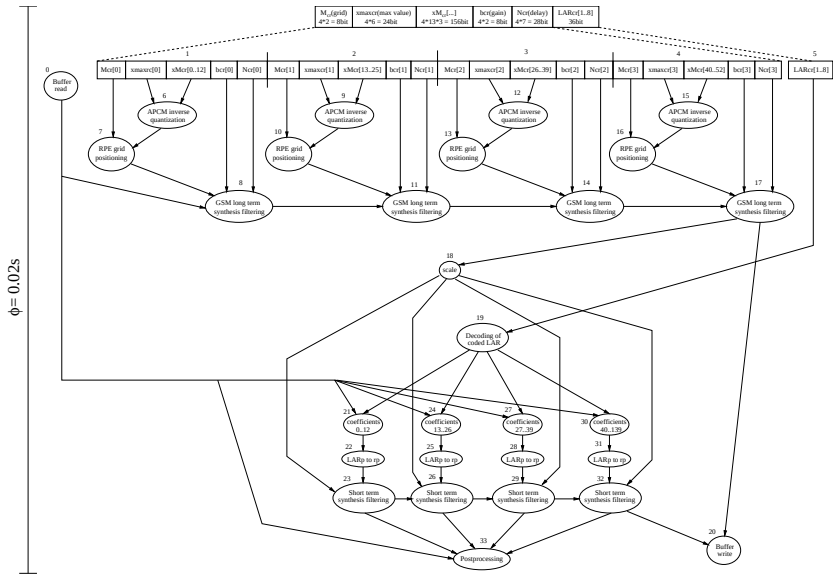


Figure 9: GSM voice decoder task graph.

```
void main1() {
  int i;

  start_metric();
  pr("Starting Hello World", 0, PR_CPU_ID | PR_STRING | PR_NEWL );
  for(i=0;i<5;i++) {
      pr("", i, PR_CPU_ID | PR_DEC | PR_NEWL );
  }

  stop_metric();
  stop_simulation();
}
```

Let us have a look at this simple C program and notice a few particularities. The first particularity is that the main entry to a "classical" C program (`int main`) is replaced by a `void main1`. We also note the function calls `start_metric()`, `stop_metric()` and `stop_simulation()`. Remember that the main benefit of having a simulator for a particular hardware platform is the ability to collect various statistics about the hardware itself or about the software program running on it. `start_metric()` must be called when we want to switch on the statistics collection and `stop_metric()` at the end. `stop_simulation()` must be called when we want to finish the simulation (you can see it as an equivalent to `exit()`).

Another useful function is `pr`. This is a very simple implementation of a `printf` equivalent for the output of integers or strings on the console. The signature is:

```
void pr(char *text, unsigned long int x, unsigned long int mode)
```

The `mode` parameter specifies what to be printed, and it can be the result of a bitwise OR of the following: `PR_CPU_ID` (prints the id of the processor where the code runs), `PR_DEC` (prints the integer variable `x`), `PR_STRING` (prints the string `text`), and `PR_NEWL` (prints a newline). In order to use these functions, include `appsupport.h` in your programs. For the other lightweight functions available to the applications in the MPARM platform, check the file:

```
/courses/TDTS07/sw/mparm/MPARM/apps/support/simulator/appsupport.h
```

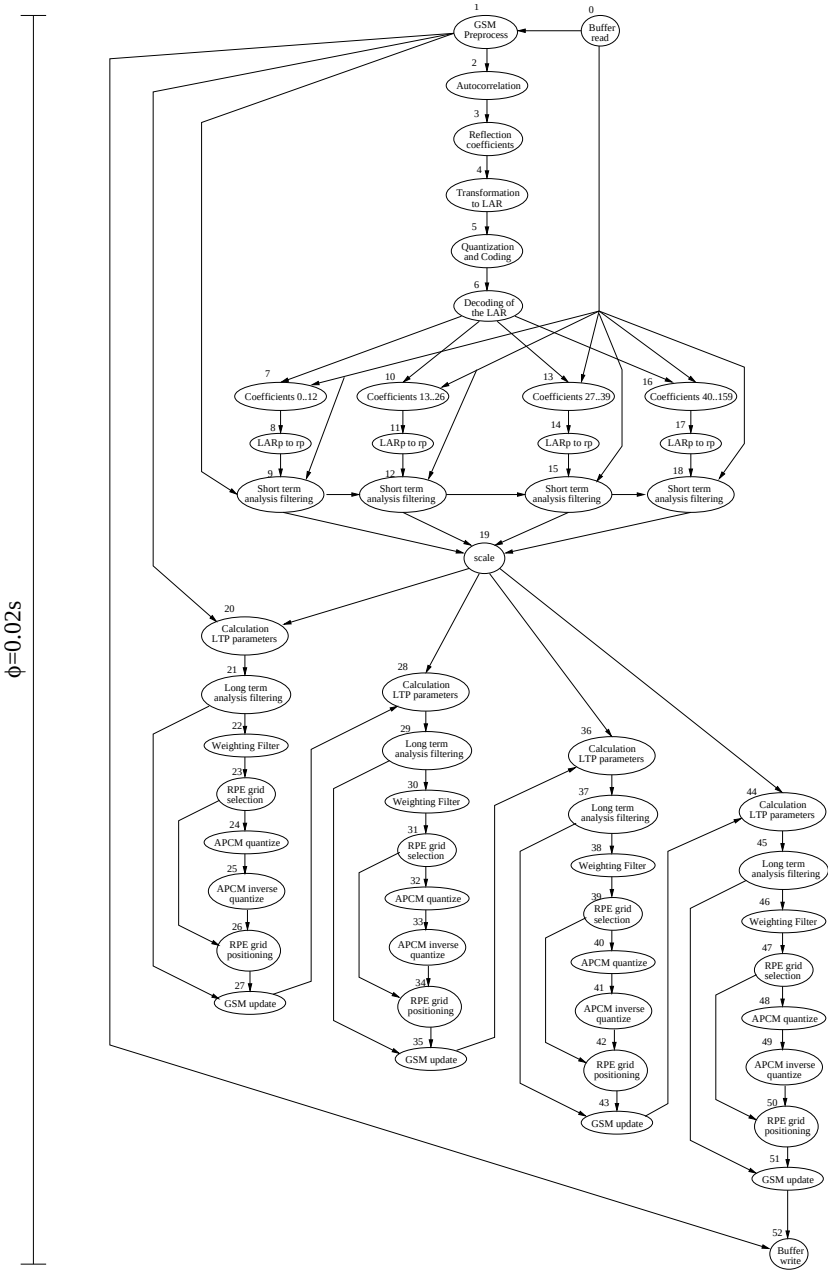


Figure 10: GSM voice encoder task graph.

6.2 Compiling the Program

The Makefile used for building our simple program is the following:

```
INCDIR       = -I. -I${SWARMDIR}/core \
               -I${SWARMDIR}/../apps/support/simulator
OPT          = -g -O3
CFLAGS       = $(INCDIR) $(OPT)

all:
        arm-rtems-as  -mfpu=softfpa -o test.o \
                      ${SWARMDIR}/../apps/support/simulator/test.s
        arm-rtems-gcc $(CFLAGS) -c -o appsupport.o \
                      ${SWARMDIR}/../apps/support/simulator/appsupport.c
        arm-rtems-gcc $(CFLAGS) -c -o hello_world.o \
                      hello_world.c
        arm-rtems-ld -T ${SWARMDIR}/../apps/support/simulator/test.ld \
                     -o linked.o \
                     test.o appsupport.o hello_world.o
        arm-rtems-objcopy -O binary linked.o TargetMem_1.mem
```

For setting up the correct paths for the MPARM simulator and cross-compiler, run the following command from your terminal:

```
source /courses/TDTS07/sw/mparm/go_mparm_bash.sh
```

In order to compile the program, type

```
make -f Makefile.mparm
```

in the directory holding your Makefile (we assume that a Makefile named Makefile.mparm is located in the directory containing the application code). The compilation will look similar to the following:

```
arm-rtems-as  -mfpu=softfpa -o test.o
              /courses/TDTS07/sw/mparm/MPARM/swarm
              /../apps/support/simulator/test.s
arm-rtems-gcc -I. -I/courses/TDTS07/sw/mparm/MPARM/swarm/core
              -I/courses/TDTS07/sw/mparm/MPARM/swarm
              /../apps/support/simulator
              -g -O3 -c -o appsupport.o
              /courses/TDTS07/sw/mparm/MPARM/swarm
              /../apps/support/simulator/appsupport.c
arm-rtems-gcc -I. -I/courses/TDTS07/sw/mparm/MPARM/swarm/core
              -I/courses/TDTS07/sw/mparm/MPARM/swarm
              /../apps/support/simulator
              -g -O3 -c -o hello_world.o
              hello_world.c
arm-rtems-ld -T /courses/TDTS07/sw/mparm/MPARM/swarm
/../apps/support/simulator/test.ld
             -o linked.o test.o appsupport.o hello_world.o
arm-rtems-objcopy -O binary linked.o TargetMem_1.mem
```

As a result, you should obtain a memory image in the file `TargetMem_1.mem`.

6.3 Running the Application

In order to run it in MPARM, use the `mpsim.x` command as follows:

```
mpsim.x -c1
```

The option `-c1` indicates how many processors will be used. The maximum is eight ARM7 processors. In this case, we have selected to run the program on one processor. As a result, the file `stats.txt` (described in Section 6.5.1) will be created in the current directory, and you will see the following output in the terminal:

```
             SystemC 2.0.1 --- Jan 12 2005 11:34:16
        Copyright (c) 1996-2002 by all Contributors
                    ALL RIGHTS RESERVED
Uploaded Program Binary: TargetMem_1.mem
Processor 0 starts measuring
Processor 0 - Hello World
Processor 0 - 0
Processor 0 - 1
Processor 0 - 2
Processor 0 - 3
Processor 0 - 4
Processor 0 stops measuring
Processor 0 shuts down
Simulation ended
SystemC: simulation stopped by user.
```

Let us try to start the same program but with `-c2`. First, we need to copy `TargetMem_1.mem` to `TargetMem_2.mem` as shown below:

```
cp TargetMem_1.mem TargetMem_2.mem
```

Then, run `mpsim.x -c2` and observe the output:

```
             SystemC 2.0.1 --- Jan 12 2005 11:34:16
        Copyright (c) 1996-2002 by all Contributors
                    ALL RIGHTS RESERVED
Uploaded Program Binary: TargetMem_1.mem
Uploaded Program Binary: TargetMem_2.mem
Processor 0 starts measuring
        Processor 1 starts measuring
Processor 0 - Hello World
        Processor 1 - Hello World
Processor 0 - 0
        Processor 1 - 0
Processor 0 - 1
        Processor 1 - 1
Processor 0 - 2
        Processor 1 - 2
Processor 0 - 3
        Processor 1 - 3
Processor 0 - 4
        Processor 1 - 4
Processor 0 stops measuring
        Processor 1 stops measuring
Processor 0 shuts down
```

```
        Processor 1 shuts down
Simulation ended
SystemC: simulation stopped by user.
```

The output is as expected. A copy of the program is running on each processor.

6.4 Mapping the Application

One way of specifying what code is to be executed on each processor is the one above: each program is written and compiled independently. Each generated memory images is copied to `TargetMem_i.mem` where `i` is the number of the desired processor. The simulator is then started with the option `-cn`. Remember that `n` is the number of processors to be used; consequently, there have to be `n` memory images: from `TargetMem_1.mem` up to `TargetMem_n.mem`. This coding style makes the process of porting existing applications to MPARM more difficult. Another disadvantage is the lack of flexibility when it comes to exploring new mappings.

An alternative way of specifying the mapping of the code to the processors is to use the function `get_proc_id()`. When using this method, we can take the code written as if it was targeted to be run on a single processor, and just mark in the source code which block is executed on which processor. For example, in the following small application, the first `for` loop is executed on CPU1 and the second `for` loop is executed on CPU2. The disadvantage of this method is that we generate one big binary that contains the code to be executed on all the processors.

```
void main1() {
  int i;
  start_metric();

  if (get_proc_id() == 1) { // CPU1?
    for (i = 0; i < 5; i++) {
      pr("", i, PR_CPU_ID | PR_DEC | PR_NEWL);
    }
  }

  if (get_proc_id() == 2) { // CPU2?
    for (i = 6; i < 10; i++) {
      pr("", i, PR_CPU_ID | PR_DEC | PR_NEWL);
    }
  }

  stop_metric();
  stop_simulation();
}
```

6.5 Collecting Simulation Statistics

6.5.1 stats.txt

As mentioned in Section 6.3, after the simulation of the application has finished, MPARM dumps various useful statistics in the file `stats.txt`. These statistics consist of data collected from the processors, caches, buses, and memories. By default, only performance-related statistics are collected. If the simulator is started with the `-w` command-line argument (`mpsim.x -c1 -w`), the power/energy consumed by the system components is also dumped. For the details regarding the precise data collected by the simulator, read the file:

/courses/TDTS07/sw/mparm/MPARM/doc/simulator_statistics.txt

6.5.2 stats_light.txt

During the design of an embedded system, it is often interesting to analyze various statistics concerning individual tasks (snapshots of the system at various points during the execution of an application). It is possible to do that in MPARM by calling the function `dump_light_metric(int x)` at different points of the application. The output is produced in the file `stats_light.txt` and consists of the number of clock cycles executed since the start of the simulation until `dump_light_metric(int x)` was called. Statistics about the interconnect activities are also collected. This is an example of output resulted from calling `dump_light_metric(int x)`:

```
-----------------------
Task 1
Interconnect statistics
-----------------------
Overall exec time          = 287 system cycles (1435 ns)
Task NC                    = 287
1-CPU average exec time    = 0 system cycles (0 ns)
Concurrent exec time       = 287 system cycles (1435 ns)
Bus busy                   = 144 system cycles (50.17% of 287)
Bus transferring data      = 64 system cycles (22.30% of 287, 44.44% of 144)
-----------------------
Task 2
Interconnect statistics
-----------------------
Overall exec time          = 5554 system cycles (27770 ns)
Task NC                    = 5267
1-CPU average exec time    = 0 system cycles (0 ns)
Concurrent exec time       = 5554 system cycles (27770 ns)
Bus busy                   = 813 system cycles (14.64% of 5554)
Bus transferring data      = 323 system cycles (5.82% of 5554, 39.73% of 813)
```

The per-task statistics are the overall execution time (the number of clock cycles executed by the system since the application start), Task NC (the number of clock cycles executed by the current task), bus busy (the number of clock cycles the bus was busy), and bus transferring data (the number of clock cycles when the bus was actually transferring data).

6.6 Implementing Interprocessor Communication in MPARM

6.6.1 Shared Memory

Let us illustrate this type of communication using the following code example:

```
void main1() {
  int i, j;
  if (get_proc_id() == 1) { // The producer: T1 on CPU1.
    int *x;
    x = (int *)SHARED_BASE; // The address in the shared memory
                            // where to store the variable.
    // Produce the message
    for (j = 0; j < 5; j++) {
      x[j] = 5 + j;
    }
```

```
  }
  else { // The consumer: T2 on CPU2.
    int *x;
    x = (int *)SHARED_BASE; // The address in the shared memory
                            // where to read the variable.
    for (j = 0; j < 5; j++) {
      pr("", x[j], PR_CPU_ID | PR_DEC | PR_NEWL);
    }
  }
}
```

Looking closer at the code above, we can see that task T1 writes in the shared memory starting from the address `SHARED_BASE` the values for five integer variables. T2 reads these variables and displays their value on the console. The `SHARED_BASE` definition can be used by MPARM programs to denote the start address of the shared memory, if `appsupport.h` is included.

6.6.2 Synchronization

One problem with the code above is that no synchronization is done. The following example code shows how this can be fixed by using the semaphore mechanism in MPARM:

```
extern volatile int *lock;

lock[0] = 1; // lock[0] is taken.
if (get_proc_id() == 1) {
  int *x;
  x = (int *)SHARED_BASE;
  // Produce the message.
  for (j = 0; j < 5; j++) {
     x[j] = 5 + j;
  }
  // Signal data ready to the consumer.
  SIGNAL(0);
}

if (get_proc_id() == 2) {
  int *x;
  x = (int *)SHARED_BASE;
  WAIT(0);
  for (j = 0; j < SIZE; j++) {
    pr("", x[j], PR_CPU_ID | PR_DEC | PR_NEWL);
  }
}
```

6.6.3 Distributed Message Passing

The following code is an example of exchanging data between two tasks running on different processors using this distributed message passing infrastructure in MPARM. Relate the code presented in this section with the examples from Section 4.

```
scratch_queue_autoinit_system(0, 0);

if (get_proc_id() == 1) {
```

```
  int *buffer;
  SCRATCH_QUEUE_PRODUCER *prod =
    scratch_queue_autoinit_producer(2, 1, MAX_QUEUE_SIZE, OBJ_SIZE, 0);
  // Ask for space in the queue buffer.
  buffer = (int *)scratch_queue_getToken_write(prod);
  for (j = 0; j < SIZE; j++) {
    buffer[j] = 5 + j;
  }
  // Signal data ready to the consumer.
  scratch_queue_putToken_write(prod);
}

if (get_proc_id() == 2) {
  int *buffer_copy;
  int i;
  SCRATCH_QUEUE_CONSUMER *cons = scratch_queue_autoinit_consumer(1, 0);
  buffer_copy = (int *)scratch_queue_read1(cons);
  for (j = 0; j < SIZE; j++) {
    pr(NULL, buffer_copy[j], PR_DEC | PR_NEWL);
  }
}
```

7 Assignments

7.1 Assignment 0: Getting Started

- Open a terminal window.
- Copy the code of a hello-world application to your home directory from:

  `/courses/TDTS07/sw/mparm/benches/hello_world`

- In order to set up the simulator and compilers, for each session, run:

  `source /courses/TDTS07/sw/mparm/go_mparm_bash.sh`

- Compile the code with:

  `make -f Makefile.mparm`

- Run it with various simulator flags and study the statistics collected by the simulator. For a complete list of the command-line parameters, run `mpsim.x -h` or study:

  `/course/TDTS07/sw/mparm/MPARM/doc/simulator_switches.txt`

7.2 Assignment 1: Design-Space Exploration for Energy Minimization

In this assignment, you will use the MPARM platform in order to optimize a multimedia application. You are given the source code of a GSM voice codec.

- Copy the code to your home directory from:

`/courses/TDTS07/sw/mparm/benches/gsm-good/single-newmparm/gsm-1.0-pl10-one-task-mparm`

- Set up the simulator and compilers and compile the code (see the previous assignment). You should now have the memory image file `bin/TargetMem_1.mem`.

The GSM codec is running on one processor. You are allowed to set the cache associativity (direct mapped or k-way associative) and size as well as the frequency of the processor. One encoding of a GSM frame has to finish in 20 ms. Find a configuration for the processor such that the energy is minimized. Report the best configuration obtained as well as the ones that you have tried (at least six). Explain the obtained results. Which parameter has the greatest impact on energy consumption according to your experiments?

Note! Remember that the energy statistics are disabled by default, so you have to enable them by specifying the `-w` command-line argument when you simulate.

Note! There are two known bugs in the simulator:

- You will not be able to simulate the system with a unified cache (corresponding to the `-u` command-line flag).
- There is no power model associated to a fully associative cache (corresponding to the `--it=0` or `--dt=0` options).

7.3 Assignment 2: Shared Memory vs. Distributed Message Passing

In this assignment, you will compare the efficiency of two communication approaches: shared memory and distributed message passing. The comparison is based on simulation results of the GSM voice codec implemented using the two alternatives. The comparison will include the bus utilization, the overall application runtime, and the system energy consumption.

- Copy the code to your home directory from:

`/courses/TDTS07/sw/mparm/benches/gsm-good/mproc/queues/gsm-mparm-multiproc-map1`

`/courses/TDTS07/sw/mparm/benches/gsm-good/mproc/shared/gsm-mparm-multiproc-map1`

- Check the code related to the communication in the `process_codec` function located in the file `src/toast.c`.
- Compile both versions (shared memory and scratchpad queues) using the corresponding Makefiles (`Makefile.mparm`).
- Run the two versions. In both cases, the application is mapped on three processors. Use the `-c3` flag when you run the simulator. The distributed message passing version needs to allocate data on the scratchpad memories. By default these are disabled in MPARM. When running the GSM with distributed message passing (the one copied from the `queues` directory), use the `-C` option of the simulator. Report the simulation results that you think are relevant to this comparison.
- Try to reduce the amount of traffic due to the synchronization in the shared memory implementation by frequency selection. The frequency for a processor can be selected statically (with the `-F x,y` command-line option), or changed dynamically while the application is running. At runtime, while the application is executing, the function

  ```
  scale_device_frequency(unsigned short int divider, int ID)
  ```

  can be called to change the frequency of a processor (the `-f --intc=s` command-line options are needed when using this function). Other functions related to frequency selection are in

  `/course/TDTS07/sw/mparm/MPARM/apps/support/simulator/appsupport.h`

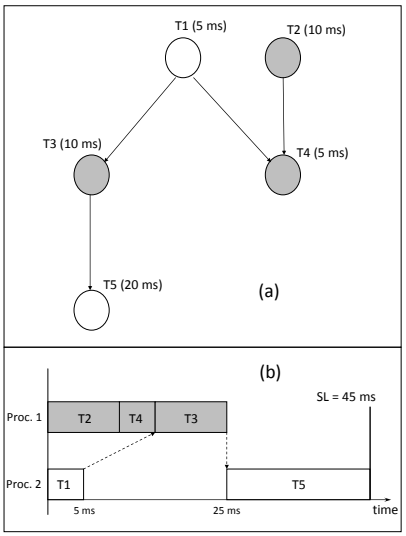


Figure 11: Task graph and schedule.

7.4 Assignment 3: Mapping/Scheduling Exercise (TDTS07 only)

Note that this assignment is required for TDTS07 but is only optional for TDDI08.

This assignment is only theoretical and does not require simulations in MPARM. Figure 11(a) shows an application comprising five tasks with data dependencies. The execution times of the five tasks T1–T5 are shown in the figure inside parentheses. The tasks are mapped to a system with two processors (CPUs). The gray tasks are running on CPU1, and the white tasks are running on CPU2. One possible schedule is shown in Figure 11(b). For the schedule of CPU1, we can see that T2 executes first and finishes at time 10 ms; after that, T4 and T3 run for 5 and 10 ms, respectively. Study the schedule for CPU2 as well. Note that the constraints imposed by the data dependencies are satisfied. The idle time of CPU2 between 5 ms and 25 ms is partly due to the data dependency between T3 and T5. Finally, note that the schedule length (the end-to-end delay of the whole application) is 45 ms (denoted SL in the figure).

Your assignment is to construct two new schedules, both with the same minimal length. One should keep the same mapping, and for the other you are allowed to change the mapping of tasks to processors. Remember to consider the data dependencies when constructing the schedules. Present your solution together with an explanation and motivation.

Note! To be able to schedule the application tasks, the designer needs knowledge about the execution-time characteristics of the individual tasks. This can be obtained by formal methods (such as worst-case execution time analysis) or by simulation, for example, using the MPARM simulation platform.