Seventh Framework Programme – Theme 3.6: Computing Systems

Grant agreement n°248776

D2.2 "Report on System-Level Dynamic Mapping & Tool Support in DOL3D"

WP2: "System-Level Exploration & Analysis of 3D Designs"

Editor: Iuliana BACIVAROV (ETHZ)

Date: Tuesday 13th March, 2012

1	CEA (Coord.)	Commissariat à l'énergie atomique et aux énergies alternatives, Laboratoire d'électronique et des technologies de l'information
2	UJF	Université Joseph Fourier Grenoble 1 – VERIMAG
3	ETHZ	Eidgenössische Technische Hochschule Zürich
4	UNIBO	Università di Bologna
5	STM	STMicroelectronics
6	EPFL	École polytechnique Fédérale de Lausanne

Version: v1.1 – Public

CEA ref.: DRT/LETI/DACLE/12-0200

Versions of the Document

Version	Date	Author	Comment
0.1	Nov., 14th, 2011	P. KUMAR	Outline
0.2	Dec., 12th, 2011	P. KUMAR, D. CHOKSHI	First draft
0.3	Dec., 14th, 2011	I. BACIVAROV, D. CHOKSHI	Second draft
0.4	Dec., 15th, 2011	I. BACIVAROV	Third draft
1.0	Dec., 22nd, 2011	I. BACIVAROV	Final version
1.1	Mar., 12th, 2012	C. FABRE	Public version

Contributors

Iuliana BACIVAROV (ETHZ), Devesh CHOKSHI (ETHZ), Pratyush KUMAR (ETHZ),
Lothar THIELE (ETHZ), Hoeseok YANG (ETHZ)

Contents

1 Introduction

3D-integration has been proposed as an innovative design technique meant to advance the design of multiprocessor systems by increasing their integration scale and therefore improving their computational performance. However, the new introduced architectural features impose new design challenges in the software development phase. In particular, as a consequence of greater integration, power densities increase and hence on-chip temperatures can exceed maximum admitted thresholds. The software design must thus be aware of the physical process of heat flow in the chip. This leads to unprecedented amount of low-level hardware information that needs to be integrated to the software development process. A second difficulty in large 3D architectures is the extensive communication network. In the architectures proposed in PRO3D, networks-on-chip (NoCs) are used for communication. As a consequence, the latency incurred in communication between two points is more difficult to estimate. A third difficulty that has to be considered is how to efficiently make use of the memory locality in such a 3D structure. All these features play a crucial role in identifying a mapping that is optimized with respect to performance and that is thermal aware.

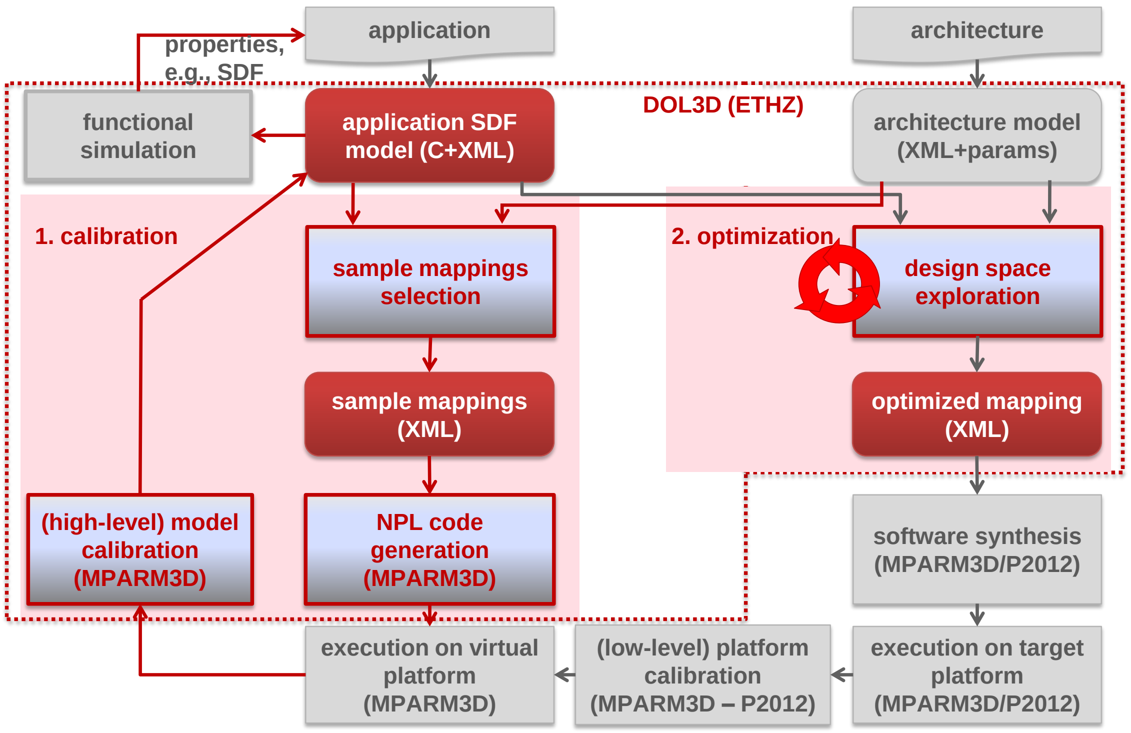

application
architecture
properties, e.g., SDF
DOL3D (ETHZ)
functional simulation
application SDF model (C+XML)
architecture model (XML+params)
1. calibration
2. optimization
sample mappings selection
design space exploration
sample mappings (XML)
optimized mapping (XML)
(high-level) model calibration (MPARM3D)
NPL code generation (MPARM3D)
software synthesis (MPARM3D/P2012)
execution on virtual platform (MPARM3D)
(low-level) platform calibration (MPARM3D – P2012)
execution on target platform (MPARM3D/P2012)

Figure 1: DOL3D design flow.

Figure 1 identifies the main design phases in DOL3D design flow. Implementing the Y-chart design paradigm, DOL3D considers parallel streaming applications represented as synchronous dataflow graphs (SDFs) and specified independently from the architecture. An optimized mapping is found after the system-level exploration of different design alternatives. Mapping here means binding application elements to computation and communication resources of the considered platform. Ultimately, the same system-level specification of the application and architecture together with the optimal mapping specification form the synthesizable system specification, which will be implemented on the final system or can be simulated on the virtual platform. Typically, we use this low level simulation in a feedback loop for automatically calibrating our time and thermal analysis models used for comparison of different mapping alternatives, by using techniques similar to those described in [11] [19].

This document will detail each of the design phases in DOL3D tool-flow and introduce our advances in performance and temperature analysis, mapping optimization, and design space exploration.

2 Application, architecture, and mapping representation

In this section, we will briefly discuss the representations of applications (section 2.1), MPARM3D/P2012 architecture (section 2.2), and mapping of an application to the architecture (section 2.3).

2.1 Application representation

This section gives an overview of the DOL3D application representation. For a complete description, please refer to deliverable D3.1 [4]. As stated in the deliverable D3.1 [4], synchronous dataflow graphs (SDF) [12] is the model of computation used in PRO3D. In DOL3D, an application specification consists of (a) a structural representation, as defined by the structure of the SDF graph, and (b) the functional behavior of each individual actor in the SDF. We now describe each of these two aspects.

2.1.1 Structural specification

The structure of an application is defined by the structure of an SDF graph. SDFs are directed graphs, with nodes representing actors (i.e., processes) and directed edges representing communication channels between actors.

In DOL3D, these SDFs representing application structures are syntactically represented in XML format. This representation captures the interconnection of the following elements: `process`, `sw_channel`, `connection`, and `port`. An example is shown in Figure 2, where for the directed communication between two processes P and C (Figure 2(a)), a channel S is explicitly represented with the corresponding ports and connections (Figure 2(b)).

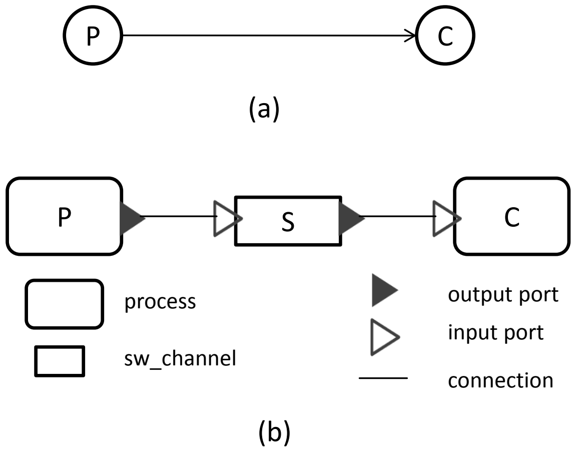


Figure 2: Process network representation.

Each of the above elements in Figure 2(b) is explicitly represented in the XML file and has specific parameters. For instance, a `process` has parameters that assign it a unique name, the `C/C++` source code which specifies its functionality, and a list of input/output ports. In addition, benefitting of the extensibility of the XML format, other details can be annotated during the design flow. For instance, after the functional simulation (see section 3.1 for the description of the DOL3D functional simulation), ports of each process are annotated with the number and the size of tokens communicated through them during execution. Also after the calibration step on the actual PRO3D platform (see section 3.3), estimates of execution times on this platform are also annotated in the application specification.

For a complete listing of the application XML file and detailed explanations of different elements, please refer to deliverable D7.1 [5].

2.1.2 Functional specification

The behavior of the DOL3D application is given by the functionality of individual processes. In DOL3D, the functionality of the processes is written in separate C/C++ files. These files contain two basic methods, namely `DOL3D_init()` and `DOL3D_fire()`. `DOL3D_init()` is executed once at the beginning, while `DOL3D_fire()` is executed repeatedly until the application is terminated with the `DOL3D_detach()` command.

The communication between processes is done through a specific communication API, which consists of two basic primitives `DOL3D_read()` and `DOL3D_write()`. The semantics of these primitives will finally be determined in the moment of the implementation, and will actually depend on platform-specific design choices.

As the model of computation is SDF, the total number of tokens consumed or produced by an actor on each channel must remain *constant* across all firings. In DOL3D, we interpret consuming/producing a token as a single call to the primitive `DOL3D_read()` or `DOL3D_write()`, respectively. Hence, the number of `DOL3D_read()` and `DOL3D_write()` calls must be *constant* for each actor, across firings. However, note that the amount of data read/written by `DOL3D_read()` or `DOL3D_write()` can vary across firings. Validation if a given application satisfies this property is performed by the DOL3D functional simulation, as discussed in section 3.1.

2.2 PRO3D architecture representation

For internal use in DOL3D, we have described the structure of the target multi-/many-core platform, complying with both P2012 [16] and MPARM(3D) [5]. Please note that in this phase of the project, the focus has been more on MPARM(3D), as we can generate code and freely execute specifications on this platform.

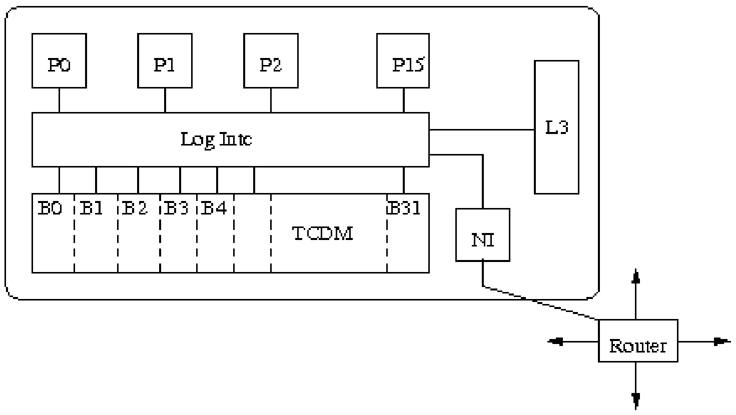


Figure 3: A cluster of the P2012/MPARM(3D) platform.

A single cluster (tile) of the considered architecture is shown in Figure 3. The underlying architecture consists of four clusters connected by a NoC in a 2 × 2 mesh topology. Each cluster consists of a configurable number of cores (16 cores `P_0` ... `P_15` are shown in Figure 3), a tightly coupled data memory (`TCDM`), the `L3` memory, a logarithmic interconnect (`LogIntc`), and a network interface (`NI`). The `LogIntc` acts as the main connecting fabric inside the cluster. It connects the cores with the `TCDM` and `L3` memory, and also with the `NI`. `TCDM` contains multiple memory banks.

Listing 1 is an example of minimal specification of core 2 in cluster 1 of the platform. However, benefitting of XML extensibility, more information can be added at design time, when needed by design tools. For the complete XML description of the architecture, please refer to deliverable D5.2 [6].

```
<processor name=``1.2'' basename=``processor'' range=``8''
 type=``RISC''>
<frequency value="10^9" />
</processor>
```

Listing 1: Core specification.

2.3 Mapping representation

The mapping of the application on the architecture defines where and how the components of an application are executed on a hardware platform. In DOL3D, mapping specification defines the binding of individual processes of the application onto cores of the platform. In this version of DOL3D, for simplicity reasons, the communication buffer is supposed to be mapped to the memory of the process consuming (i.e., reading) data/tokens from the channel. Please note that this is just a simplification, illustrating a possible design choice and other options might be equally explored as well.

Listing 2 specifies the binding of a process named `producer` on `core 1.2` (i.e., `cluster 1 - core 2`). Details regarding the complete XML specification of mapping, please refer to deliverable D2.1 [3].

```
<binding name="binding_producer" xsi:type="computation">
    <process name="producer"/>
    <processor name="1.2"/>
</binding>

<binding name="binding_consumer" xsi:type="computation">
    <process name="consumer"/>
    <processor name="3.1"/>
</binding>
```

Listing 2: Mapping specification of a process onto the core.

3 DOL3D design flow

This section describes the various stages of the DOL3D design flow, as represented in Figure 1, and how they contribute to the final goal of mapping optimization in PRO3D.

3.1 Functional verification

The applications we consider in PRO3D are based on the synchronous dataflow (SDF) model of computation, as stated in section 2.1. However, most applications, in their original form, are provided in a sequential way. Transforming a sequential application in to an SDF application can be challenging. In particular, when aiming for such a transformation (parallelization) one would like to verify if the obtained SDF specification is (a) functionally correct and (b) a valid SDF graph. Importantly, such verifications would have to be done early in the design cycle and independent of the platform of execution.

3.1.1 Automatic generation of SystemC code

To facilitate functional validation and verification of some functional properties, the first step in DOL3D tool-chain is to generate functional SystemC executable code from the application specification, which can be simulated on a general purpose computer. In particular, we use `sc_thread`, `sc_port`, and `sc_channel` to model the SDF processes, ports, and software channels, respectively. Figure 4 shows an example SystemC model of a process network with three process (in which the `sc_thread` shown in the middle reads the data produced by the left `sc_thread` and writes the data for consumption by the right `sc_thread`).

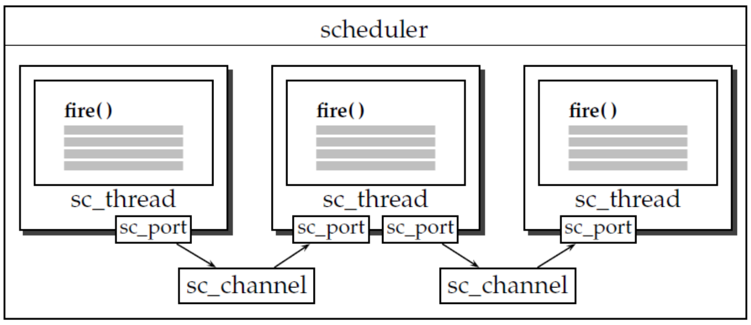


Figure 4: SystemC model of the application.

Description of code generation and corresponding commands within the DOL3D tool-chain, are presented in a separate document made available in the PRO3D repository [10].

In the following section, we discuss how generated SystemC can be used to functionally verify an application, and in particular to check if the application indeed conforms to the SDF model. As well, we discuss the profiling of first performance data from inspecting the communication patterns between the actors of the SDF graph.

3.1.2 Validating SDF properties

First and foremost, the generated code when executed on any standard machine, can be used to functionally validate the SDF application. This validation would confirm that the SDF processes work correctly in parallel and communicate correctly over FIFO channels.

In SDF graphs, each process reads/writes a fixed number of tokens on each of its communication channels each time it executes (fires). The channels in an SDF graph may contain initial tokens, produced during the initialization routine.

In the DOL3D framework, we interpret each `DOL_read()` and `DOL_write()`, as generating and consuming a single token, respectively. Thus, a given process network is not SDF if the number of `DOL_read()` and `DOL_write()` calls on any of the channels is not constant across all invocations of `fire` routines. (The number of tokens on each channel at the end of the `init` procedure of all processes defines the number of initial tokens.)

For example, Listing 3 describes the `fire` procedure of a process that has a constant number of calls to `DOL_write()`, namely 1.

```
int generator1_fire(DOLProcess *p) {
    if (p->local->index < p->local->len) {
        int x = (int)p->local->index;
        DOL_write((void*)PORT_OUT, &(x), sizeof(int), p);
        p->local->index++;
    }
    if (p->local->index >= p->local->len) {
        DOL_detach(p);
        return -1;
    }
    return 0;
}
```

Listing 3: Constant number of tokens at port `PORT_OUT`.

As counter-example, Listing 4 describes the `fire` procedure of a process that does not have a constant number of calls to `DOL_write()`.

```
int generator2_fire(DOLProcess *p) {
    if (p->local->index < p->local->len) {
        if(p->local->index % 2){
            int x = (int)p->local->index;
            DOL_write((void*)PORT_OUT, &(x), sizeof(int), p);
            p->local->index++;
        } else{
            char c = 'a';
            DOL_write((void*)PORT_OUT, &(c), sizeof(char), p);
            DOL_write((void*)PORT_OUT, &(c), sizeof(char), p);
            DOL_write((void*)PORT_OUT, &(c), sizeof(char), p);
            DOL_write((void*)PORT_OUT, &(c), sizeof(char), p);
            p->local->index++;
        }
    }
    if (p->local->index >= p->local->len) {
        DOL_detach(p);
        return -1;
    }
    return 0;
}
```

Listing 4: Variable number of tokens at port `PORT_OUT`.

3.1.3 Generation of platform-independent parameters

As specified in the previous section, the DOL3D tool-chain provides the first functional profiler, which can be used to generate platform-independent parameters, as follows:

- `numOfFires`: number of activations of each process;
- `initialAccesses`: number of read/write accesses or tokens on each port in `init` of each process;
- `initialtokensize`: number of bytes read/written on each port in `init` of each process;
- `accesses`: number of read/write accesses or tokens on each port per `fire` of each process;
- `tokensize`: number of bytes read/written on each port in `fire` of each process;
- `numOfReads`: number of read accesses from the software channel;
- `numOfWrites`: number of write accesses to the software channel;
- `numOfInitialTokens`: number of initial tokens on the software channel.

The above information is extracted from the DOL3D functional simulation and can be used to both validate the design and suggest improvements in terms of parallelizing the applications. As well, using the above parameters, the tool-chain can validate if the application specification is indeed an SDF, based on the interpretation of tokens (as explained in the previous section 3.1.2). As an internal mechanism of the DOL3D tool-chain, the information collected by the profiler will be further used for performance evaluations, as described later in section 3.3.

3.2 Automatic code generation for PRO3D platform

A crucial step in the DOL3D tool-chain is to synthesize code for the MPARM(3D)/NPL platform from given application and mapping information. We call this a crucial step, since in order to generate an optimal mapping with respect to PRO3D performance and thermal objectives, one has to be able to characterize the behavior of the application on the specified architecture. One way to do it is to generate platform-specific executable code to be simulated on a platform's simulator and to extract meaningful performance data that will be used later on in performance predictions. Following this line of thought, we will generate from the DOL3D application specifications executable code according to the native programming layer (NPL) API introduced by UNIBO in the project deliverable D7.1 - section 4.1.3 *Software control.*

Transforming a DOL3D application to NPL-compatible code requires two specific tasks: (a) manage the execution of different processes on a given core and (b) transform the communication primitives `DOL3D_read` and `DOL3D_write` to NPL-specific commands. We detail these two actions in the remainder of this section.

Thread creation. The `main` program running on `cluster 0 - core 0` is responsible for initializing (a) different processes of the application onto the cores of the platform according to a given mapping specification and (b) software channels used to communicate between processes. Each process is executed as a separate thread.

The code snippet shown in Listing 5 illustrates the thread creation for a DOL3D process. In the above thread creation, we first allocate a stack segment for the thread to be created (on the cluster where the process is mapped to) and then we create the thread on the core given by the mapping specification. In particular, in Listing 5, a thread is created on `cluster 2 -`

core 3.

```
p12_alloc_cluster(2, 1000, make_alloc_attr(0,0,0,0), &stack_seg);
thread_create(&thread_id, make_procid(2,3), process_wrapper, 0,
                     &stack_seg, 0);
```

Listing 5: Thread creation.

Thread execution. Each thread executes first the `init` routine of the corresponding DOL3D process. Then, it repeatedly calls the corresponding `fire` routine, which is executed until the process is finished by `DOL3D_detach()`.

Inter-thread communication. Communication amongst threads is specified by the corresponding channels in the SDF process-network file. To enable communication along these channels, first, software-based communication channels are created using NPL APIs. Then, the `DOL3D_write` and `DOL3D_read` primitives are transformed using the NPL APIs for communication.

Listing 6 illustrates the initialization and allocation of the memory for communication channels. Here, the channel memory is allocated as specified in the mapping file. In the case shown in Listing 6, memory on `cluster 3` is allocated.

```
channel_buffer[0] = (char*)p12_alloc_cluster(3, 20*sizeof(char),
                        make_alloc_attr(0,0,0,0), &shs_seg);
tmp_segment.base = (void*)channel_buffer[0];
tmp_segment.size = 20*sizeof(char);
channel[0] = Q_CREATE(&tmp_segment, sizeof(char), QTT_BUFFER, NULL);
```

Listing 6: Communication channel creation.

The communication primitives `DOL3D_write()` and `DOL3D_read()` are implemented using `Q_OUT_AVAILABLE()`, `Q_PUSH()`, `Q_POP()`, and `Q_IN_AVAILABLE()` of NPL. Listing 7 shows the implementation of `DOL3D_write()`.

```
void DOL_write_implementation(void *port, void *buf, int len,
DOLProcess *process) {
    ProcessWrapper* process_wrapper = (ProcessWrapper*)process->wptr;
    int i, retval;
    segment_t tmp_segment;
    q_handle_t qbuf;

    for (i = 0; i < process_wrapper->number_of_out_ports; i++) {
        if (process_wrapper->out_port_id[i] == (int)port) {
        qbuf = channel[process_wrapper->out_channel_id[i]];
        tmp_segment.base = buf;
        tmp_segment.size = len;
           while (Q_OUT_AVAILABLE(qbuf) < len) {
              if (TEST_VERBOSE) _printstrp("wait for sufficient buffer");
           }
        retval = Q_PUSH(qbuf, NULL, &tmp_segment);
        TEST_ASSERT(retval == Q_SUCCESS);
        break;
        }
    }
}
```

Listing 7: `DOL3D_write` implementation.

3.3 Performance model and calibration

This section describes a simple performance model used in DOL3D and the calibration process used to obtain accurate model parameters from the PRO3D platform.

The model is built in the phase denoted as "calibration" in Figure 1. The purpose of this "calibration" phase is to acquire enough knowledge on PRO3D applications executed on PRO3D platforms, such that, based on this knowledge further performance predictions are made. After building the performance model, the "calibration" phase can be totally decoupled from the tool-chain. The obtained performance model can be thus used independently in the system "optimization" phase (highlighted in Figure 1 as well), for fast evaluations.

One has to note that during the calibration phase the entire PRO3D tool-chain has to be functional in order to execute DOL3D applications on the platform simulator. First, applications and architectures have to be specified in DOL3D format; different "calibration" mappings have to be generated; executable code has to be generated for the corresponding simulator/-platform; all "calibration" mappings have to be executed on the platform simulator; finally, performance data have to be captured during all these executions. In particular, for our evaluations we use MPARM(3D) with NPL layer, and we instrumented the simulation to acquire performance data during execution.

The execution time of an application is mainly indicated by the `fire()` procedures, describing the behaviors of each of the application processes. In the `fire()` procedure, one can distinguish between pure computation time and time spent in communication routines. For that reason, we instrument the simulation to capture individual timings for different *segments*. As illustrated in Figure 5, we distinguish between three types of segments: (1) *computation segment*, (2) *read segment*, and (3) *write segment*. While a *read segment* is just a `DOL3D_read()` and a *write segment* is just a `DOL3D_write()`, a *computation segment* is the maximal contiguous block of C-code in the `fire()` without containing any `DOL3D_read()` or `DOL3D_write()`. As an example, consider the `fire()` procedure of the `generator1` process in Listing 3. The process has three segments: (1) a *computation segment* consisting of `lines 2` and `3`, (2) a *write segment* at `line 4`, and (3) a second *computation segment* consisting of `lines 5` to `11`.

The execution time model is designed based on non-linear regression. In particular, inspired by different segments of a process execution, we distinguish in the model as well between computation time and communication time. Computation time directly depends on the utilization of the cluster on which the process (segment) is mapped. Communication time depends on both utilizations of source and destination of this communication, as well as on the relative distance between source and destination. While several communication models can be imagined depending on the NoC structure, we take a simple approach and distinguish between three cases, i.e., communication between processes on the same tile, communication between processes on neighbor tiles, and communication between processes on far-away tiles.

The following equation is used to describe the execution time of a particular segment in the application:

$$\begin{aligned} y \quad = \quad & a_1 + a_2x_1 + a_3x_1^2 + a_4x_1^3 + \\ & a_5x_1x_2 + a_6x_2 + a_7x_2^2 + \\ & a_8x_1x_3 + a_9x_3 + a_{10}x_3^2 + \\ & a_{11}x_1x_4 + a_{12}x_4 + a_{13}x_4^2 \end{aligned} \tag{1}$$

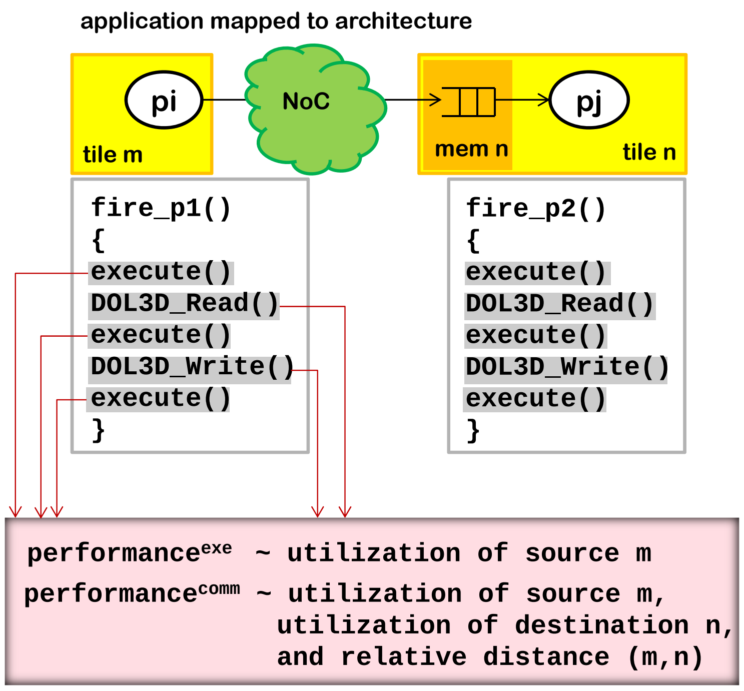

application mapped to architecture
pi
tile m
NoC
mem n
pj
tile n
fire_p1() { execute() DOL3D_Read() execute() DOL3D_Write() execute() }
fire_p2() { execute() DOL3D_Read() execute() DOL3D_Write() execute() }
performance^exe ~ utilization of source m
performance^comm ~ utilization of source m, utilization of destination n, and relative distance (m,n)

Figure 5: Analysis model generation - segment-based granularity.

The values of variables x_1, x_2, x_3, and x_4 depend on the type of the segment (i.e., computation, read, or write) and on the mapping, as follows:

- For *computation segments* or *read segments*, x_1 is the utilization of the cluster on which the segment is mapped to, while x_2, x_3, and x_4 are set to be `0`. The idea behind this is the following. The *computation* depends on the latency to access local memory on the cluster, and higher the number of processes on the cluster, the higher the interference on the shared memory of the cluster, and therefore a higher execution time for processes. In our current implementation of `DOL_read()`, as we map the software channel to the memory at the destination's cluster, the execution time of the segment depends on the utilization of that cluster. Otherwise, the discussion would be similar as for write segments.

- For a *write segment*, three cases are considered, depending on the mapping of source and destination processes:

  – *Both the source and destination processes are mapped to cores of the same cluster.* In this case, x_1 and x_2 are set to the utilization of that cluster, while x_3 and x_4 are set to `0`. (Here, x_2 indicates that both processes are on the same cluster.)

  – *The source and destination processes are on neighboring clusters.* In this case, x_1 is set to the utilization of the cluster on which the source process is mapped; x_3 is set to the utilization of the destination cluster; and x_2 and x_4 are set to `0`. (Thus, in this case, we use x_3 to identify the destination mapping.)

  – *The source and destination processes are neither on the same cluster nor on neighboring clusters (i.e, they are located more than 1-hop away from each other).* x_1 is set to the utilization of the source cluster; x_4 is set to the utilization of the destination cluster; and x_2, and x_3 are set to `0`. (Thus, in this case, we use x_4 for modeling of "far-away" communication.)

Model data is obtained after the calibration on a number of predefined mappings on the platform. For each mapping and for each segment, we collect two types of values: (a) values of variables x_1, x_2, x_3, and x_4, and (b) the execution time y for the considered segment. Once these parameters established, we train parameters $a_i, 1 \leq i \leq 13$ to obtain our complete performance model. The execution time of any segment given any mapping can be calculated with this model. The total execution time of a process is then obtained by summing up the individual execution times of all segments. Ultimately, the execution time of an application μ is computed as the maximum of the total execution times of each process in that application.

3.4 Worst-case thermal analysis

This section describes our investigations and some results on conservative thermal analysis of multi-core systems, done at design time.

3.4.1 Worst-case thermal model

A modular temperature analysis method based on the framework of real-time calculus has been introduced in D2.1 [3] and [14]. The aim is to bound the worst-case peak temperature of a system, that is, the maximum temperature that can occur under all possible scenarios of task executions. The work is similar to real-time analysis, where the time critical instant is identified as releasing all tasks/events as early as possible without violating arrival curve constraints [9].

For temperature analysis, the temperature critical instant for a single processor is identified among infinitely many traces that comply with the event stream specification in MPA, in [14]. We demonstrate that the workload trace that leads to the worst-case peak temperature at time instant τ is to release events in time interval $[\tau - \Delta, \tau)$ such that the processed computation in time interval $[\tau - \Delta, \tau)$ is maximized under arrival-curve constraints. For example, for periodic tasks with jitter, the worst-case execution time occurs if burst arrivals and jitters happen at system initialization, while the worst-case peak temperature occurs if the system is first warmed up with periodic arrivals and then suddenly heat up with burst arrivals and jitters. Therefore, $R^*(0, \Delta) = Q^*(0, \Delta) = \gamma(\tau) - \gamma(\tau - \Delta)$, with $\gamma(\Delta) = \inf_{0 \leq \lambda \leq \Delta}\{(\Delta - \lambda) + \alpha(\lambda)\}$ leads to worst-case temperature, among all possible traces. The upper bound on component temperature $T^*(\tau)$ is guaranteed at time τ, with a precision determined by steady-state temperature and thermal parameters of the underlying hardware, see [14].

However, for multi processor systems, besides the worst-case temperature due to self-heating described above, heat transfer between components needs to be equally considered. In addition to the upper bound on the temperature of each processing component k, $T_k^*(\tau)$, we have to construct the worst-case temperature sequence $Q_i{}^*(0, \Delta)$ for all neighbor components $i \in \{1, n\}$ with respect to the chosen component k. The worst-case neighboring sequences will differ for all analyzed components due to geometrical differences and imply different transfer delays between cores. Finally, the worst-case peak temperature of the system T_S^* is computed as the maximum of all component temperatures T_k^*, that is, $T_S^* = \max(T_1^*, \ldots, T_n^*)$.

Often, finding the exact solution on worst-case heat transfer is only possible via exhaustive search due to the huge number of possibilities in event patterns and undetermined locations of heat transfer functions maxima. To speed up the calculations, we practically over-approximate neighboring temperatures by simply maximizing the convolution between mirrored burst activities and accumulated maximum period of activity in the vicinity of the approximated maximum heat transfer function. In our experiments, the over-approximation comes with a very small error of below 1% in temperature variation, as reported in [13].

3.4.2 Illustrative results

DOL3D was used to optimize the mapping of a motion JPEG (MJPEG) decoder on three ARM cores communicating via a shared bus. Both *worst-case latency* and *worst-case peak temperature* are considered during optimization. The MPARM virtual platform is used to determine the computational demand of different processes and the power dissipation of ARM cores. Fixed priority preemptive scheduling is used on all processors, while a TDMA policy is employed on the bus. The application is driven by a periodic input stream, with a period of 110 kcycles and a jitter of 220 kcycles. Thermal parameters were obtained from HotSpot [15].

Trade-off solutions on one, two, and three cores are illustrated in Figure 6. The processors are identical, but they have different thermodynamical properties, due to their placement.

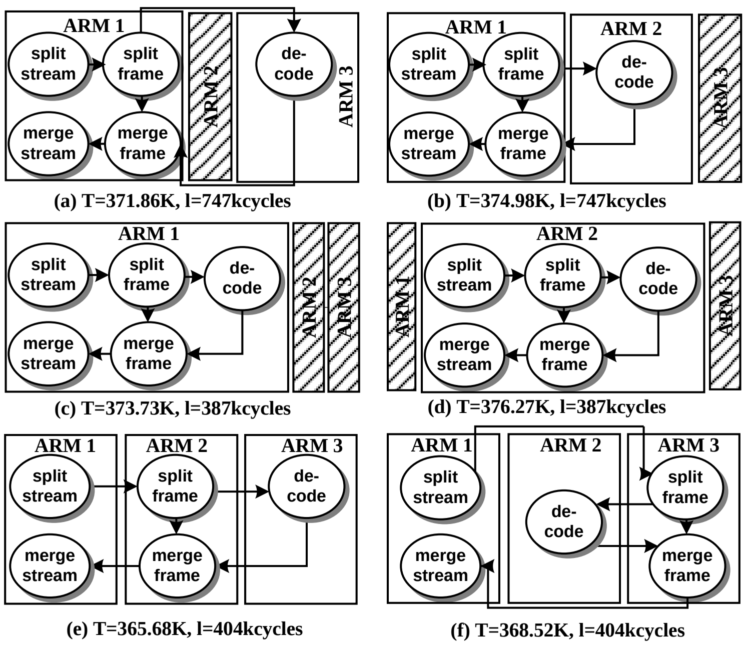

ARM 1
split stream
split frame
merge stream
merge frame
de-code
ARM 2
ARM 3
(a) T=371.86K, l=747kcycles
(b) T=374.98K, l=747kcycles
(c) T=373.73K, l=387kcycles
(d) T=376.27K, l=387kcycles
(e) T=365.68K, l=404kcycles
(f) T=368.52K, l=404kcycles

Figure 6: Some preliminary performance-temperature investigations on MPARM.

Solution pairs where only the placement the homogeneous processors has been changed (e.g., mappings a and b), indicate that physical placement cannot be ignored in temperature analysis. In our simple examples, we observe the same latency in all pairs because of the homogeneous processor structure and the symmetric shared bus, but more than 3K maximum temperature variation.

No solution is optimal with respect to both latency and temperature. The load balanced mapping (mappings e and f) is a “cool” solution, due to small neighboring heat transfer and shorter accumulated bursts. The two-processors solution (mappings c and d) has lower latency since it eliminates some of the inter-processor communication overhead, but it is characterized by the highest peak temperature due to neighboring effect and long accumulated bursts. Note, that mappings on fewer cores could present lower temperatures if unused processors would be turned off. In our experiments, unused processors still consume (non-negligible) idle power.

This section introduced a conservative thermal model and a set of experiments illustrating the non-trivial dependencies between performance and temperature. The next section will detail our design space exploration framework, while the last section of this deliverable will include more advanced thermal control aspects we plan to investigate in the next phase of WP2 in

PRO3D.

3.5 Design space exploration

In the previous section, we discussed our performance analysis models and methods for thermal analysis under investigation. In particular, it has been shown how to analytically obtain the execution time μ and the temperature T, given a binding $b \in B$ (where B is the decision space of mapping). Now, the questions we would like to address are: *how to search over the space of all possible bindings?*, and also, *how to compare two mappings, for instance, when one is having higher performance but high temperature and the other is having lower temperature, but lower performance as well?* These two issues will be addressed by using multi-objective evolutionary algorithms, this section being devoted to a discussion of our approach.

3.5.1 Multi-objective optimization: methods and goals

Evolutionary algorithms [21] are general-purpose stochastic search algorithms belonging to the class of evolutionary computation. They can be considered as black-box optimizers in multi-dimensional optimization spaces, i.e., aiming at optimizing a vector-valued objective function f over a decision space X. Two vector-valued functions can be compared using the notion of *Pareto-dominance.* Evolutionary algorithms consist of several phases such as *initialization* followed by iterative steps of *mating selection*, *crossover operation*, *mutation operation* and finally *environmental selection.* The *initialization phase* generates a population of solutions, typically on a random basis. In *mating selection*, a few "good" solutions are selected as *parent solutions.* Using the *cross-over and mutation operators* new *child solutions* are generated from chosen parent solutions, and are added to the population. At this step, the new population is compared with the population at the beginning of the previous iteration. If the new population is found better (based on pre-defined criteria) it is retained. The iteration then repeats.

In our problem, the vector-valued objective function is $f = (\mu, T) : B \to \mathbb{R}^2$. Optimizing the system means minimizing f, with respect to both parameters. The decision space consists of the binding function b which binds each process to a core. Given a solution (i.e., a mapping), we can compute the values of μ and T by using the models of time and temperature introduced in the previous sections.

The mating selection, cross-over and mutation operators can be defined in standard ways on the solution vector. For instance, a standard cross-over operator could choose the binding of an actor in the child solution to be equal to that of a randomly chosen parent. Please note that "standard" operators, without considering the semantics of actual solutions may represent an advantage. But as well, on specific problems, designers can intervene on these operators to actually guide the random evolution towards better solutions, using domain-specific knowledge. Then, during environmental selection, two populations can be compared by using the hyper-volume indicator [20], i.e., roughly speaking, the set of solutions covering a larger volume in the objective subspace are preferred.

The above procedure iteratively refines an initial population of solutions to finally obtain a Pareto-optimized population. All solutions in this Pareto-optimized population are incomparable with respect to the two objective functions. We can then choose one amongst the different mappings, with preference for one or another objective, e.g., performance or temperature.

We use the frameworks of EXPO [17] and PISA [8] for the mapping optimization and the Strength Pareto Evolutionary Algorithm (SPEA2) [22] as underlying multi-objective search algorithm. Using a standard search method relieves us from implementing the problem-independent parts of the evolutionary algorithm and allows us to focus on problem specific parts, that is, the generation and variation of solutions. In particular, by using this approach, we only need

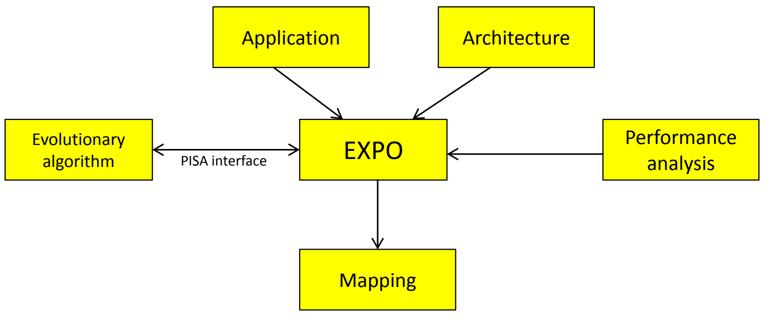


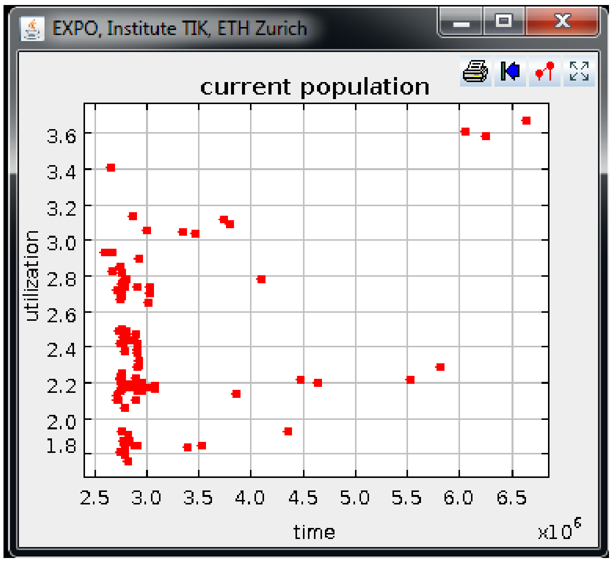
Figure 7: EXPO interfaces

to implement the evaluation and generation, because the selection is handled inside SPEA2. In the case of EXPO, the implementation is done in three Java classes:

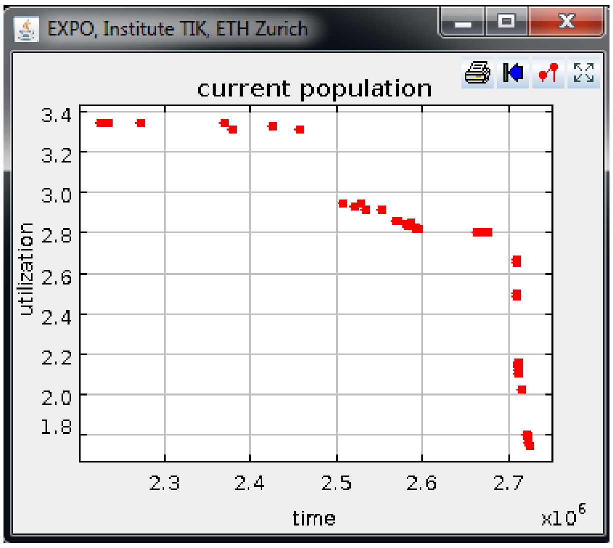
- *Specification.* In the *specification* class, the static components of the problem are modeled, such as the application structure and the architecture.
- *Gene.* In the *gene* class, a single candidate mapping is represented. In addition the variation methods for the crossover and mutation are implemented in this class.
- *Analyzer.* In the *analyzer* class, the objective values of a single candidate mapping are computed. Here, analysis models as those discussed in sections 3.3 and 3.4 for both performance and temperature are invoked.

3.5.2 Multi-objective optimization: some results

Some preliminary results of running EXPO for a test application of nine processes on the MPARM3D platform are presented, with the objectives of minimizing (1) the time taken to execute the application and (2) the maximum utilization of any of the four clusters of the platform (which is in direct correlation to cluster temperature, in a simplified abstraction). Figure 8 (a) plots the initial population, randomly generated for the search. Each red dot in the figure represents one mapping. After running EXPO, we obtain a set of optimized mappings with respect to both objectives, illustrated in Figure 8 (b). Our tool offers such a visual support for mapping investigations, and in this way designers can inspect mappings and finally select their design choice. For instance, by clicking on one particular dot (i.e., mapping) in Figure 8 (b) the mapping specification consisting of the binding of each process to the respective cores of the platform is displayed, as illustrated in Figure 9.

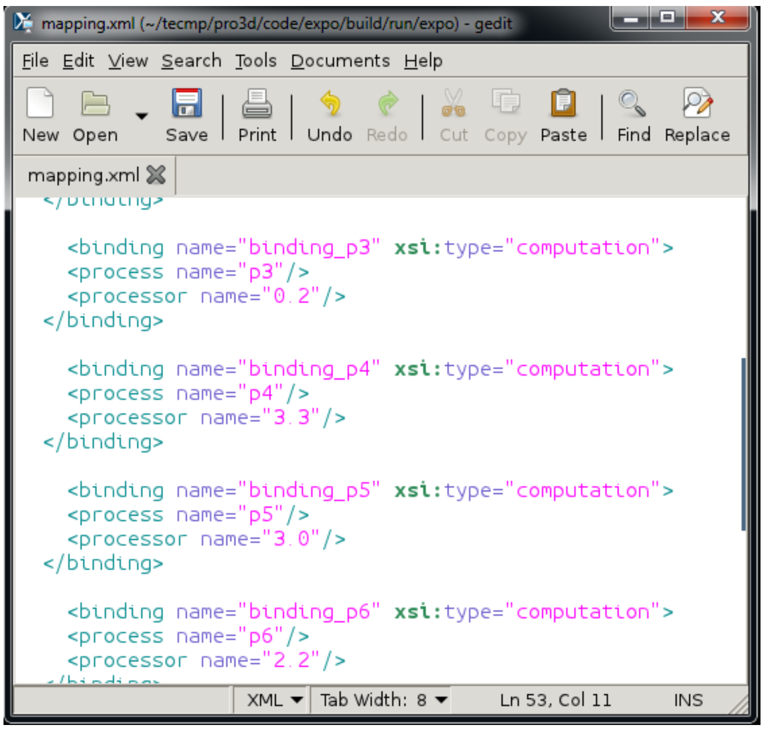


(a)



(b)

Figure 8: EXPO: (a) initial population, (b) a set of optimized mappings (each dot represents a mapping)


mapping.xml (~/tecmp/pro3d/code/expo/build/run/expo) - gedit
File Edit View Search Tools Documents Help
New
Open
Save
Print
Undo
Redo
Cut
Copy
Paste
Find
Replace
mapping.xml
XML
Tab Width: 8
Ln 53, Col 11
INS

```
</binding>

    <binding name="binding_p3" xsi:type="computation">
    <process name="p3"/>
    <processor name="0.2"/>
</binding>

    <binding name="binding_p4" xsi:type="computation">
    <process name="p4"/>
    <processor name="3.3"/>
</binding>

    <binding name="binding_p5" xsi:type="computation">
    <process name="p5"/>
    <processor name="3.0"/>
</binding>

    <binding name="binding_p6" xsi:type="computation">
    <process name="p6"/>
    <processor name="2.2"/>
</binding>
```

Figure 9: EXPO: one of the optimized mappings - XML description.

4 Theoretical investigations into dynamic mapping

4.1 Introduction

Thermal management is proposed at several levels in the PRO3D platform. First, the cooling infrastructure is designed, analyzed and verified to provide the necessary heat dissipation to the system [1]. Second, when exploring for different mappings, it is possible to have temperature-based objective functions that are optimized along with performance-based objective functions. As mentioned in the previous section, we may aim at minimizing the peak temperature along with minimizing the total execution time.

In spite of the above design-time choices, there may be the need to respond to run-time thermal emergencies. Such emergencies may arise due to incorrect models for heat dissipation, power consumption, or software execution times. The wide breadth of the associated parameters necessitates consideration for such dynamism. In the project, we propose to respond to such run-time effects primarily using the cooling infrastructure. The proposed PRO3D platform allows for the flow rate of the input coolant onto the micro-channels to be changed dynamically [2]. Thus, the coolant flow-rate can be increased in response to high on-chip temperatures. Such a solution has two advantages. First, it does not need any specific "transition", as the only change is in the cooling infrastructure. Second, we can expect that the performance provided by the architecture through such a transition is unaffected. For contrast, if the high temperatures were to be responded to by decreasing the frequency of the system, the performance of the system could be affected.

To complement the above dynamic approach, we have investigated two other approaches. In either approach the software is affected by the dynamism. In the first approach, we consider how we can "re-map" an application to reduce high on-chip temperatures. The prime consideration here is the computation of different mappings. In the second approach, we consider how control-theoretic approaches can be used to control the frequency of the system. The prime consideration in the this approach is to discuss the effect on the performance provided by such a system. We discuss these two interventions in the rest of the section.

4.2 Offline computation of multiple mappings

The main consideration for run-time re-mapping of the application is to distribute the computation amongst offline and online stages. One solution is to re-compute a revised mapping during runtime, considering the temperature of the system. However, such a runtime reconfiguration can be computationally intensive and perhaps not feasible during situations of high temperature. The alternative is to pre-compute certain mappings at design-time and to then intelligently switch between these mappings during run-time. This is the approach we adopt, whereby *we pre-compute several mappings at design-time amongst which run-time re-mappings can be switched.*

High temperatures and high performance are contradictory requirements. When the temperature of the system is high, the maximum performance that can be extracted from the system will be lower. In this sense, there exists a natural trade-off space between temperature and performance. It is our aim to express this trade-off space through offline computations. We do this by using evolutionary algorithms to simultaneously consider temperature and performance objective functions.

In the DOL3D tool-chain, in the EXPO phase we compute mappings with two objective functions: (a) lower time-to-completion, and (b) lower localized utilization demand. While the first objective function expresses the performance requirement of the system, the second represents the temperature objective function. Higher the maximum utilization of any core, higher the temperature. Considering these two objective functions, we can compute the Pareto front. An example of such a front is shown in Figure 8 (b). This Pareto front is characterized by several mappings where the mappings are said to be incomparable in the multi-objective sense. These mappings naturally generate the set of mappings amongst which we can switch at run-time. If the temperature of the system is too large, we can switch to a mapping with a lower temperature while compromising on the provided performance.

It is possible to refine the thermal objective function further. For instance, it is possible to compute the peak temperature using a thermal simulator, such as 3DICE [7], and incorporate that as the objective function in EXPO. Yet another approach could be to analytically compute peak temperature for steady-state conditions and use the objective function in EXPO. In this sense, the tool-chain allows for a modular approach, where different specific temperature objective functions can be incorporated, as illustrated in section 3.4 on an example.

4.3 Feedback-control-based speed actuation

As a second approach, we consider the theoretical investigation of a system where an adaptive approach to dynamic voltage and frequency scaling (DVFS) is used. We consider a system where frequency (or speed) of the system is changed as a feedback control based on the current temperature of the system. In other words, the frequency of the system is given by a feedback control law such as

$$f(t) = g(T(t)), \tag{2}$$

where $f(t)$ and $T(t)$ are the frequency and temperature of the system at time t, respectively. The function g is the feedback control law that captures the relationship between temperature and frequency.

Such a system provides some interesting advantages. Independent of the software that is to execute on the system and its timing properties, we can compute provably safe bounds on the temperature of the system. In other words, there we have the property of fault isolation: Even if some design-time software model is faulty, we can still guarantee that the temperature constraint is still satisfied. However, such a system has the disadvantage that the performance of such a system is difficult to analyze. Since the frequency is temperature dependent the performance provided by the system is a function on the past execution of the system.

The above challenge in analyzing the performance is of especial concern in real-time systems where tasks must be guaranteed to complete within timing deadlines. We studied the problem of analyzing the worst-case delay suffered by a stream of jobs when served by a system with feedback speed actuation. We characterized the jobs by standard real-time timing models such as arrival rate [18] denoted as $\alpha(\Delta)$. This model captures the upper-bound on the execution time of jobs arriving in any interval of length Δ. With arrival rate we can compactly represent variability in the arrival and execution times of a stream of jobs. The challenge then is to compute the worst-case delay under any permissible arrival pattern of jobs.

We study the problem for a *given* time horizon denoted as τ, i.e., we observe the system for the time-interval $[0, \tau]$. We proved that the worst-case delay suffered by any job in the system is simply given by the delay suffered by the last job of the specific trace of jobs:

$$R(t) = \alpha(\tau) - \alpha(\tau - t), \quad t \in [0, \tau]. \tag{3}$$

We also proved that the above bound is tight, i.e., there exists a stream of jobs where a job actually suffers from the worst-case delay. Thus, tight timing analysis of a processor with feedback speed actuation is straight-forward: we have to symbolically simulate a single job arrival pattern and observe the delay of the last job.

The above analysis considers a cold start, i.e., the temperature of the system at time 0 is equal to the ambient temperature. However, if this initial temperature is higher, the worst-case delay can be different. To consider this variation, we extended the results to consider any initial temperature. In summary, we showed that the above worst-case trace also gives the worst-case delay on a "modified" thermal model of the system, where the temperature of the processor is clipped to the initial temperature. The proofs of these results along with other details are available in [14].

5 Summary

This document describes the current status of DOL3D tool-chain and its role in PRO3D. The ultimate role of DOL3D is to optimize the mapping of PRO3D applications to 3D platforms, tackling the main challenges of 3D integration. Implementing the Y-chart design paradigm, DOL3D considers parallel streaming applications represented as synchronous dataflow graphs (SDFs) and specified independently from the architecture. An optimized mapping is found after the system-level exploration of different design alternatives, with respect to both timing and thermal parameters. The document summarizes the DOL3D specification format, the automatic generation of different simulation levels, and introduces our advances in performance and temperature analysis models, mapping optimization, and design space exploration.

References

[1] PRO3D consortium, D1.1: Definition of the 3D Integration Flow and Thermal Modeling with Through Silicon Vias. 2010.

[2] PRO3D consortium, D1.2: Report on Specification of 3D Stack Technology with Liquid Cooling. 2010.

[3] PRO3D consortium, D2.1: Report on Process-Level Mapping and Tool Support in DOL3D". 2010.

[4] PRO3D consortium, D3.1: Report on 3D Aware Specification of Parallel Applications. 2010.

[5] PRO3D consortium, D7.1: 3D Requirements Synchronisation for Technical Work Packages. 2010.

[6] PRO3D consortium, D5.2: Port an Application to PRO3D Tool Chain. 2011.

[7] 3D-ICE. 3D Interlayer Cooling Emulator, EPFL, `http://esl.epfl.ch/3D-ICE`.

[8] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. Pisa: A platform and programming language independent interface for search algorithms. In *EMO*, pages 494–508, 2003.

[9] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analyzing system properties in platform-based embedded system design. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 190–195, Munich, Germany, 2003.

[10] D. Chokshi, P. Kumar, I. Bacivarov, H. Yang, and L. Thiele. Functional simulation and validation of DOL3D application, `https://minalogic.net/svnroot/PRO3D/code/ETHZ/doc/functional_simulation_DOL3D.pdf`.

[11] K. Huang, W. Haid, I. Bacivarov, M. Keller, and L. Thiele. Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. *ACM Transactions in Embedded Computing Systems (TECS)*, 2011.

[12] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, 1987.

[13] P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang. Mapping of applications to mpsocs. CODES+ISSS11, October 9-14, 2011, Taipei, Taiwan., 2011.

[14] D. Rai, H. Yang, I. Bacivarov, J.-J. Chen, and L. Thiele. Worst-case temperature analysis for real-time systems. DATE11, Grenoble, France, 2011.

[15] K. Skadron et al. Temperature-aware microarchitecture: modeling and implementation. *ACM T. Arch. and Code Opt.*, 1(1):94–125, 2004.

[16] STMicroelectronics and CEA. Platform 2012: A Many-core Programmable Accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology, Nov. 2010. Whitepaper.

[17] L. Thiele. EXPO , `http://www.tik.ee.ethz.ch/pisa/variators/expo/documentation.html`.

[18] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. *ISCAS*, 4:101–104, 2000.

[19] L. Thiele, L. Schor, H. Yang, and I. Bacivarov. Thermal-aware system analysis and software synthesis for embedded multi-processors. In *Proc. Design Automation Conference (DAC)*, San Diego, California, USA, 2011. ACM.

[20] E. Zitzler, D. Brockhoff, and L. Thiele. The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration. In *EMO*, pages 862–876, 2006.

[21] E. Zitzler and L. Thiele. Multiobjective optimization using evolutionary algorithms - a comparative case study. In *PPSN*, pages 292–304, 1998.

[22] E. Zitzler and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization Evolutionary Methods for Design, Optimisation, and Control. In *CIMNE*, pages 95 – 100, 2002.