December 7, 2011

3D-ICE 2.0

This document provides a brief summary of the usage of 3D-ICE 2.0. This includes illustrative examples of generating the input stack and floorplan files, and the various functions used for printing the results.

User Guide

Table of Contents

1. License and Copyright

This file is part of 3D-ICE 2.0.

3D-ICE is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version. 3D-ICE is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with 3D-ICE. If not, see http://www.gnu.org/licenses/.

Any usage of 3D-ICE for research, commercial or other purposes must be properly acknowledged in the resulting products or publications. Specifically, [1] and [2] must be cited in these cases.

Authors:

Arvind Sridhar[*]
Alessandro Vincenzi[*]
Giseong Bak[*]
Martino Ruggiero[*]
Thomas Brunschwiler[†]
David Atienza[*]

Contact Information:

EPFL-STI-IEL-ESL
Bâtiment ELG, ELG 130
Station 11
1015 Lausanne, Switzerland

Email: 3d-ice@listes.epfl.ch
(SUBSCRIPTION NECESSARY!)
URL: http://esl.epfl.ch/3d-ice.html

This research has been partially funded by the Nano-Tera RTD project CMOSAIC (ref.123618) - which is financed by the Swiss Confederation and scientifically evaluated by SNSF, and the PRO3D project- financed by the European Community 7th Framework Programme (ref.FP7-ICT-248776).

[*] Embedded Systems Laboratory, Department of Electrical Engineering, EPFL, Lausanne, Switzerland.
[†] Advanced Thermal Packaging Group, IBM Research Laboratory, Zurich, Switzerland.

2. What is new in 2.0?

3D-ICE 2.0 features several new additions and improvements over the first release of the simulator in September 2010. You would find the following main changes in this package:

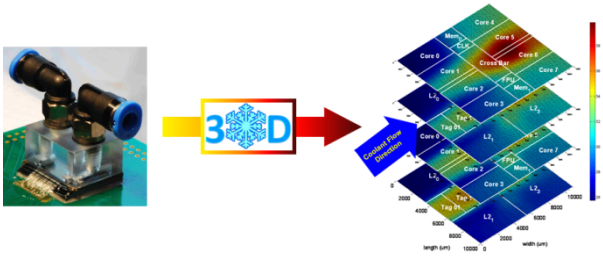
- A new porous medium model has been implemented for the simulation of convective heat transport in interlayer microchannel cavities
- Various new and enhanced heat transfer geometries have been introduced for the interlayer cavities.
- The user interface has been improved considerably. All the input-output information to the simulator has been transferred to the stack descriptor file, requiring little or no programming of the main C file by the user. This feature brings it closer to SPICE-like circuit simulators that use netlists for circuit descriptions. Hence, 3D-ICE 2.0 can now function as both a stand-alone thermal simulator as well as a software thermal library.
- The user now has the option of doing steady state thermal analysis in addition to transient analysis. This is useful for obtaining steady state temperatures for some average/corner case heat dissipation scenarios, without having to wait for the transients to settle; as well as for generating initial temperature states for certain types of transient simulations.
- A new network interface has been created for online thermal simulations of IC architectures, in tandem with functional emulation on other devices such as FPGAs. Under this system, 3D-ICE installed on a computer acts as the server, which communicates with a device such as an FPGA emulating the desired architecture via an Ethernet cable. This feature is particularly useful for those who wish to build and test online-thermal-performance-management schemes with Hardware-Software cosimulation of ICs with liquid cooling without actually building the device.
- Extensive new documentation has been provided with this release to help simplify the usage of the simulator– this includes this extended user guide, research publications and a new ***Doxygen*** source code documentation. Doxygen has been used to create an organized html-based visualization of the 3D-ICE software library files and can be accessed by running in the `make doc` command inside the 3D-ICE folder, and then opening `3d-ice/doc/html/index.html` on your web browser.

We ask the users of 3D-ICE 1.0 to take special note of some **major changes introduced in the writing of the stack description file in 3D-ICE, highlighted in bold text throughout this text**. These changes have been introduced to incorporate the above-mentioned new features, as well as to improve the robustness of the library.

3. Who needs 3D-ICE?

3D-ICE, or "**3D Interlayer Cooling Emulator**", is a Linux based *generic simulation platform* written in C to simulate the transient thermal behavior of 3D IC structures with inter-tier microchannel heat sinks. It is intended for various purposes including, but not limited to,

- Performing thermal analysis of 2D or 3D ICs during early stages of VLSI circuit/architecture design by electronic engineers,
- Simulating run-time thermal management strategies such as DVFS, dynamic work allocation, variable coolant flow rate etc.,
- Testing of microchannel heat-sink performances by microfabrication engineers and heat-sink designers,
- Evaluating accuracies of new and existing heat transfer correlations by experimental heat transfer engineers.



3D-ICE is based on the conventional compact modeling of heat transfer by conduction in solids, and advances a novel compact modeling methodology, called the Compact Transient Thermal Modeling (CTTM), for heat transfer by convection in microchannels. The user is free to use microchannel heat sinks of any dimension with the corresponding heat transfer performance data depending upon the accuracy/speed needs of the user. This simulator is ideal for situations where a quick estimate of chip temperatures is required, when the electronic designer is still iterating between various floorplanning and operating strategies in order to optimize for electronic performance and thermal safety/reliability of the final system.

In addition, the format of inputs, outputs and the problem construction/solving in 3D-ICE have been modeled on the popular compact modeling simulator **HotSpot**, making it easier for users who are familiar with this tool or with conventional circuit simulators. With numerous functions to access a variety of thermal data during the simulation, the user can reach deep into the heart of the thermal simulation, use the data to interface with other (popular or custom) tools and automate any kind of design/run-time optimization algorithms using 3D-ICE. More functionality will be added in the future versions of 3D-ICE to make this interfacing even more automatic and easy.

For more details on the theory and discussions about the accuracy/speed of the modeling technique used in 3D-ICE 2.0, please refer to the publications [1] and [2] available with this library.

4. Before you begin

The 3D-ICE library has been written and developed using:

- bison 2.4.1
- flex 2.5.35
- gcc 4.1.2

Make sure that these tools are installed on your system before compiling and that the corresponding variables in makefile.def point to the respective binary file. To use 3D-ICE, you must also download the SuperLU library [4], available at http://crd.lbl.gov/~xiaoye/SuperLU/ .

A. Compile SuperLU

Before compiling 3D-ICE, you must compile the SuperLU library by executing the following commands:

```
$ wget http://crd.lbl.gov/~xiaoye/SuperLU/superlu_4.0.tar.gz

$ tar xvfz superlu_4.0.tar.gz

$ cd SuperLU_4.0/

$ cp MAKE_INC/make.linux make.inc
```

Next, check and edit the `SuperLUroot` variable in `./make.inc` and select the `blas` library before compiling. You can either use a `blas` library installed on you system or the `blas` library supplied by the authors of SuperLU (see the README file). If you decide to use the former then the variable BLASDEF must be set and `BLASLIB` must point to your `blas` library. Then compile SuperLU with

```
$ make
```

For the latter case, `BLASDEF` must be unset and BLASLIB must point to the `./libblas.a` archive. Then compile SuperLU with

```
$ make blaslib

$ make
```

These are the operations that can be done when compiling SuperLU on a generic Linux platform. In case of a different architecture, please reference to the README file.

B. Compile 3D-ICE

Check and edit the `SLU_MAIN` variable in `./makefile.def` to make it point to the main folder of SuperLU. Next, select the value of `SLU_LIBS` according to the choice done above when compiling SuperLU. You can then compile 3D-ICE with:

```
$ tar xvfz 3d-ice-2.0.tar.gz

$ cd 3d-ice

$ make
```

Now you can find the executable `3D-ICE-Emulator` in the `3d-ice/bin/` folder. This will serve as the thermal simulator application for all your 3D-ICE projects.

Note to 3D-ICE 1.0 users: The locations of the executable has now been changed from the `examples/` folder to the `bin/` folder.

5. Overview of 3D-ICE

This chapter provides a general overview of 3D-ICE: its operating principle, the input files required and the conventions and terminology used in writing these input files. In the next chapter, these input files are discussed in detail.

A. Principle of thermal simulation

3D-ICE is based on the compact modeling of heat flow in solids and liquids applied to a 3D-IC structure with microchannel cooling. As quick recap, the structure is divided into cuboidal *thermal cells* based on the discretization parameters you provide. Next, thermal conductance for heat flow through each face of the cuboid is calculated and connected to the neighboring cells at these faces. Also, a capacitance representing the heat capacity of the cell is calculated. Hence, an equivalent electrical RC circuit is created where the temperatures are represented by voltages and the heat flow is represented by the currents in the circuit. A typical thermal cell representing transient conduction in solids is shown below:

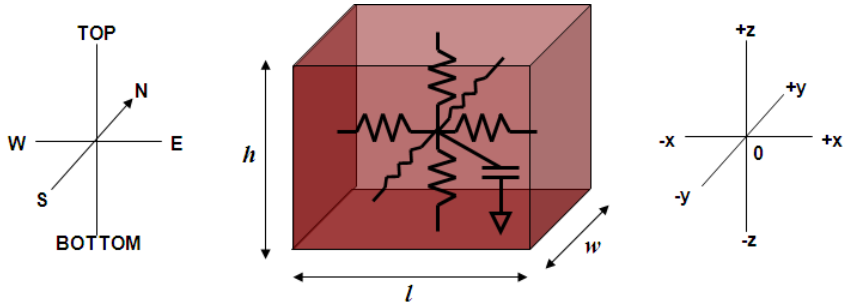


Figure 1: A typical solid thermal cell

In the case of thermal cells representing microchannels or fluidic cavities, while the heat transferred from the cavity walls into the fluid is represented using conventional thermal resistances (or conductances), the convective heat transport in the downstream direction due to mass flow is represented using voltage controlled current sources. There are four different interlayer cooling cavity models implemented in 3D-ICE 2.0.

i. Microchannel 4-resistor model

This is the model, which was available in 3D-ICE 1.0, is based on the 4RM-CTTM model described in [1]. Here, the thermal cells are constructed such that in the cavity layer, 2 faces of the thermal cells corresponding to the microchannel completely cover the entire cross-section of the microchannel as shown in Fig. 2. Hence, the discretization along the direction perpendicular to the channels is fixed by the channel geometry. There are four resistances one for each wall of the microchannel representing convective cooling of the wall surfaces. They can be coupled with solid thermal cells on all four directions as shown in Fig. 2. The convective resistance in the top, bottom and the sides can be provided independently into the simulator

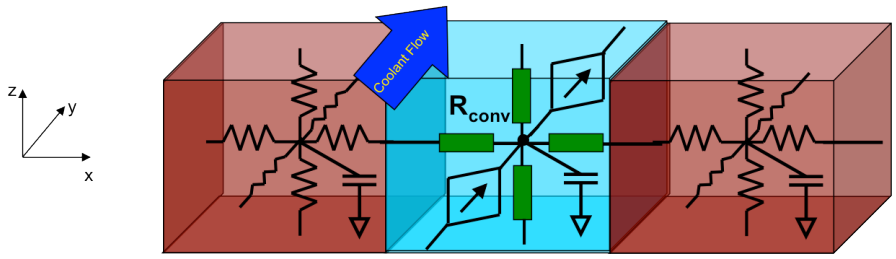


Figure 2: An mc4rm fluid thermal cell adjacent to two solid thermal cells

ii. Microchannel 2-resistor model

This new model for microchannel cavities is based on the 2RM-CTTM model presented in [2]. Here, the dependence of discretization on the channel geometry is completely removed, by homogenizing the entire cavity layer into a single "porous" material. Hence, each cell consists of circuit parameters corresponding to both the flowing coolant as well as the solid walls as shown in Fig. 3. The thermal cell here is two-dimensional because of the elimination of convective resistances on the sides. This model lets the user completely control the granularity of the model by making the thermal cell dimensions (in the x-direction) completely independent of the channel dimensions. Also, when using this model, the top and bottom heat transfer coefficients to be provided in the input files of 3D-ICE must be a projected average of the heat transfer coefficients between the side walls, and the top and bottom walls respectively. For more information about constructing this model, refer to [2].

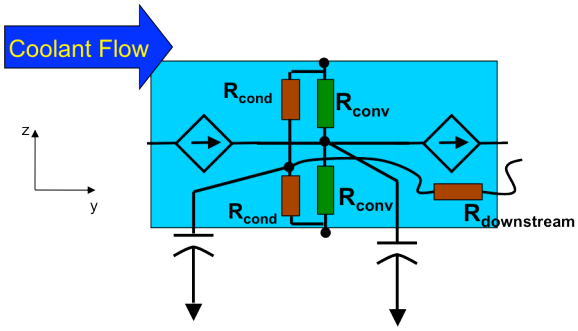


Figure 3: An mc2rm thermal cell

iii. Pin fins in-line

Enhanced heat transfer geometries, such as pin fins can be used in interlayer cooling to improve the heat transfer properties of the liquid-cooled heat sink, when compared to conventional microchannels. Pin fins are also attractive because the can serve as ideal paths for the fabrication of TSVs in 3D ICs. These advantages come at the cost of increased pressure drops. Inline pin fins are one of the two standard types of pin fin HTGs. The top view of such an HTG is shown in Fig. 4.

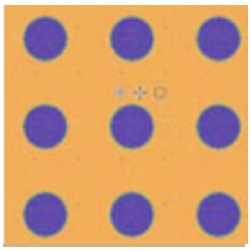


Figure 4: Pin fins inline heat transfer geometry

iv. Pinfins staggered

Staggered pin fins (Fig. 5) is another standard type of pin fin heat transfer geometry. For both these pin fin geometries the 2RM-CTTM modeling method is utilized. The corresponding "porous" thermal cell is illustrated in Fig. 6.

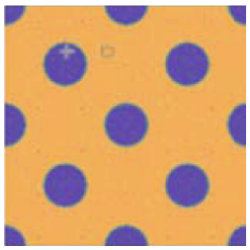


Figure 5: Pin fins staggered heat transfer geometry

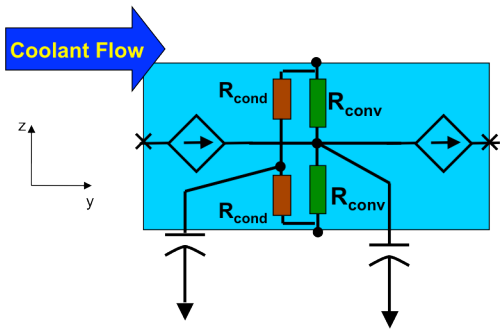


Figure 6: A pin fin thermal cell

In 3D-ICE 2.0, the heat transfer coefficients calculations for two specific test cases have been implemented that uses the darcy velocity of the coolant in the cavity. These computations for the convective resistance are based on empirical studies on these geometries performed in [3] and the resulting formulae (where v_{darcy} is the darcy velocity in m/s):

$$\mathrm{HTC}_{\mathrm{inline}} = \frac{2.526\times10^{-5}}{\left(\frac{v_{darcy}}{1\,m/s}+1.35\right)^{0.64}}+1.533\times10^{-6}$$

$$\mathrm{HTC}_{\mathrm{staggered}} = \frac{2.526\times10^{-5}}{\left(\frac{v_{darcy}}{1\,m/s}+1.35\right)^{1.52}}+1.533\times10^{-6}$$

B. Inputs to 3D-ICE

3D-ICE accesses all the information needed to emulate a 3D IC from two different types of input files and these constitute a 3D-ICE project:

- **Stack Description File**: The stack description file or the "stack descriptor" is the project file created by the user to describe the 3D IC thermal problem that will be solved. It contains information about the structure, material properties of the 3D Stack, the description of the various heat sinks in the system, the discretization parameters, analysis parameters, and finally, commands to 3D-ICE for printing out the desired outputs from the simulation. Please refer to Section 6.A when writing this file.
- **Floorplan File**: This file describes the architecture of a 3D IC design as is seen by the thermal model. It contains information about the location and the size of each major logic block in the IC, and the corresponding heat dissipation traces, as a function of time. Each die included in the stack descriptor must have a corresponding floorplan file. Multiple dies can use the same floorplan file. Please refer to Section 6.B for instructions on how to write this file.

C. Convention and Terminology

In 3D-ICE, both Cartesian coordinates and cardinal directions are used to describe the location of cells/nodes and direction of heat flow in the structure. The indices of cells/nodes along the *x* direction (WEST-EAST) are sometimes referred to as *columns*, the indices along the *y* direction (SOUTH-NORTH) are sometimes referred to as *rows*, and the indices along the *z* direction (BOTTOM-TOP) sometimes referred to as *layers*. Also, the word *length* is used primarily to refer to dimensions in the *x* direction, the term *width* is used for the *y* direction and the term *height* is used for the *z* direction unless otherwise specified. A typical solid thermal cell along with the coordinate systems used in the library is shown in Fig. 1.

The corresponding model for a liquid cell is shown in Fig. 2. Note that the current sources shown here correspond to the fluid flow in the microchannels. In this library, only flow direction NORTH or *+y* is supported, and hence, when microchannels used in a 3D-IC structure, they are always laid out facing SOUTH-NORTH (with the inlet being at the southern end and the outlet at the northern end of the channels). Hence, the northern edge of the IC is expected to be the hottest and the southern end of the IC is expected to be the coldest in any analysis. It is up to you to decide your floorplanning of the ICs accordingly.

The syntaxes used in this document for describing how these input files must be written are based on the following convention:

`[: … :]`	POSIX characters class
`…	…`	OR- either of the two elements must be used
`[ … ]?`	an optional element

`[ … ]+` one or more of this element must be used

`[ … ]*` zero or more of this element must be used

Within the files, keywords must be written in low case and all the white spaces belonging to

```
[:space:]
```

will be skipped during the parsing. Identifiers (referred to as ID) must match the following expression:

```
[:alpha:] [ _ | [:alnum:] ]*.
```

Floating points values (referred to as DVALUE) must belong to

```
[+|-]? [:digit:]+ [ \. [:digit:]+ [ [e|E] [+|-]? [:digit:]+ ]? ]? .
```

Please refer to the flex sources in `3D-ICE/flex` for more details. It is also possible to insert comments at the end of a line (`//`) or to comment an entire block (`/* ... */`) of the input files (similar to C or Java)

6. Creating a 3D-ICE project

As mentioned in Section 5.B, a 3D-ICE project consists of writing a stack descriptor file and one or more floorplan files. See the example Stack Description File and Floorplan files provided in the `./examples` folder for reference.

A. Stack Description File

The stack description file (*.stk) is a netlist that specifies all the physical and geometrical properties of the 3D-IC for the simulation. The extension of the file is not relevant- it will be parsed independent of its presence or content.

The stack description file contains EIGHT main sections (mandatory and optional) and they must be declared in this order:

1. Materials
2. Conventional heat sink
3. Liquid-cooled cavity
4. Dies
5. Dimensions
6. Stack
7. Analysis options
8. Output instructions

Note for 3D-ICE 1.0 users: The number of sections has increased from six to eight. The ordering of the sections has also been changed.

The following description of each of these sections is NOT in the above-mentioned order for ease of presentation and coherence.

i. Materials

The first section of the file contains the list of materials and their properties to be used in the simulation. At least one material must be declared. Materials are declared with the syntax,

```
material MATERIAL_ID :

        thermal conductivity     DVALUE ;

        volumetric heat capacity DVALUE ;
```

where

- `MATERIAL_ID` is a unique identifier to refer to this material,
- `thermal conductivity` is expressed in W/µm K ,
- `volumetric heat capacity` is expressed in J/µm^{3}K .

Materials declared here but not used in the following sections (channel, dies or stack) will be reported with a warning message (`stderr`).

Example

```
material SILICON :
        thermal conductivity     1.30e-4 ;
        volumetric heat capacity 1.628e-12 ;
```

ii. Dies

A die is a group of layers stacked together to form a single entity that is used when declaring the sequence of stacked elements of the 3D IC later in the file. This can represent an actual IC die in the stack. You can declare multiple dies, and use a single die multiple times during the stack description. The Dies section is a mandatory section and must contain at least one die element. A die must contain one *source* layer (the term *source* layer is used to denote those layers of the stack which contain active electronic components, and hence, provide the heat source for the simulation) and zero or more passive layers. The source layer can be placed at any location in the stack of layers in a die.

```
die DIE_ID :

        [ layer  IVALUE MATERIAL_ID ; ]*

          source IVALUE MATERIAL_ID ;

        [ layer  IVALUE MATERIAL_ID ; ]*
```

where

- `DIE_ID` is the unique identifier used to refer to the declared die,
- `IVALUE` is the height of the layer (in µm),
- `MATERIAL_ID` is the (previously declared) identifier of the material composing the layer.

The order of the layers within the die reflects their vertical disposition in the 3D IC, i.e., the first layer declared is the top most layer in the die (closer to the ambient) while the last one is the one at the bottom (closer to the PCB). Two examples of die declaration and their illustrations (not to scale) are shown below.

Example

```
die TOP_IC:
        source 2 SILICON;
        layer 50 SILICON;
```

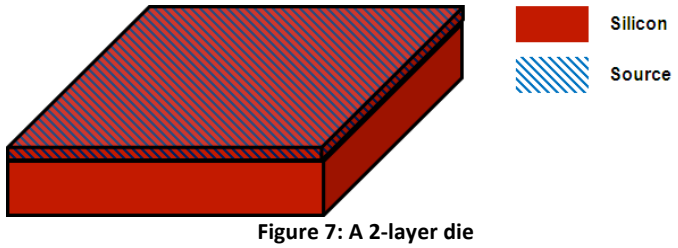


Figure 7: A 2-layer die

```
die BOTTOM_IC:
       layer 10 BEOL;
       source 2 SILICON;
       layer 50 SILICON;
```

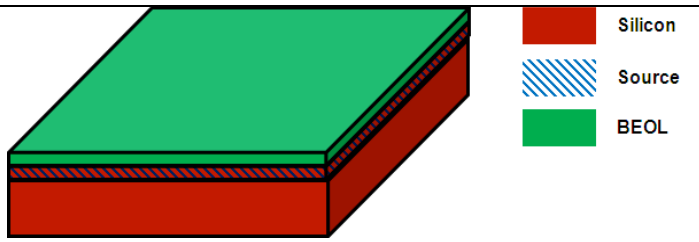


Figure 8: A 3-layer die

iii. Conventional Air-Cooled Heat Sink

This is an optional section which includes a conventional air-cooled heat sink in the 3D IC. All the faces of the 3D IC stack are modeled as adiabatic walls by default. When the Conventional Heat Sink is specified, the top surface of the stack is connected to the ambient via a thermal resistance.

```
conventional heat sink :

       heat transfer coefficient DVALUE ;

       ambient temperature DVALUE ;
```

where

- `heat transfer coefficient` of the heat sink is expressed in W/µm^{2}K,
- `ambient temperature` is the ambient temperature expressed in K.

Example

```
conventional heat sink:
       heat transfer coefficient 1.0e-7;
       ambient temperature 300;
```

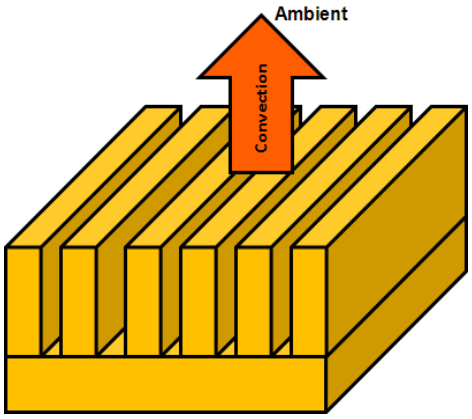


Figure 9: Air-cooled heat sink

iv. Liquid-cooled cavity

The Liquid-cooled cavity (henceforth referred to simply as “cavity”) section provides 3D-ICE information about the interlayer liquid-cooled heat sink. Since in a given 3D IC design, all interlayer cavities are designed identically, only one declaration of (and one type of) cavity is allowed. As mentioned earlier, there are four types of cavities in 3D-ICE 2.0 and the flow of coolant is always the SOUTH-NORTH or *+y* direction. This section can be omitted for simulating a 3D IC without liquid cooling (solid only).

Note to 3D-ICE 1.0 users: This entire section is new in 3D-ICE 2.0. While certain parameters under this section are common to all types of cavities and are the same as in 3D-ICE 1.0- these parameters are explained only once for brevity, some are specific to a particular geometry.

Microchannel 4-resistor model

```
microchannel_4rm :

        height DVALUE ;

        channel length DVALUE ;

        wall length DVALUE ;

        [ first wall length DVALUE ; ]?

        [ last  wall length DVALUE ; ]?

        wall material MATERIAL_ID  ;

        coolant flow rate DVALUE ;

        coolant heat transfer coefficient [ DVALUE |  side DVALUE ,

                                                      top  DVALUE ,

                                                      bottom DVALUE ] ;

        coolant volumetric heat capacity DVALUE ;

        coolant incoming temperature DVALUE ;
```

where

- `height` (in µm) corresponds to the height of the microchannel layer. This must exactly correspond to the microchannel height from the surface of the top wall to the surface of the bottom wall because of the 4RM-CTTM and 2RM-CTTM modeling requirements. Solid walls bounding the top and bottom faces of the microchannel would constitute new layers that should be declared separately,
- `channel length` and `wall length` are the cross sectional lengths of the channel and the wall (in the *x* direction, all in µm). During the discretization of the system (see the Dimensions section for more details), 3D-ICE automatically starts with a wall at the eastern most end of the layer, and alternate channels and walls along the *x* direction. But the user must ensure that dimensions are such that the layer always ends with a wall (in other words, the number of columns must always be an odd number),
- `first wall length` and `last wall length` (in µm) are optional properties that represents the length of the western-most (the first column) and eastern-most (the last column) walls. This option has been included since, during the fabrication of microchannels on the back of the substrate, although the etching mask pattern is predominantly regular, there are chances of irregularities at the ends, or there might be a deliberate use of different dimensions for the first and the last wall to preserve uniformity and symmetry of heat transfer coefficient. If one of

these two dimensions (or both) is not declared, then the `wall length` will be used at its corresponding location,

- `MATERIAL_ID` is the identifier of the material composing the walls (the ID must be previously declared in the materials section),
- `coolant flow rate` is expressed (in ml/min) and it refers to the volume of coolant flowing per unit time per channel layer (cavity) in the stack. If you have multiple layers of microchannels, the total flow rate must be divided by the number of channel layers and given as a single input,
- `coolant heat transfer coefficient` is the Heat Transfer Coefficient of convective heat removal from the walls into the coolant (in $W/\mu m^2K$). It is possible to specify a single value of HTC for all the wetted surfaces of the microchannel or specify three different values- one for the *side* wall surfaces, one for the *top* and one for the *bottom* wall surface of the microchannel.
- `coolant volumetric heat capacity` is expressed in $J/\mu m^3K$,
- `coolant incoming temperature` is the inlet coolant temperature expressed in K.

Example

```
microchannel_4rm:
       height 100 ;
       channel length 50;
       wall length 50;
       first wall length 25;
       last  wall length 25;
       wall material SILICON;
       coolant flow rate 42;
       coolant heat transfer coefficient
                           side 2.7132e-8,
                            top 5.7132e-8,
                         bottom 4.7132e-8;
       coolant volumetric heat capacity
                              4.172e-12;
       coolant incoming temperature
                                     300;
```

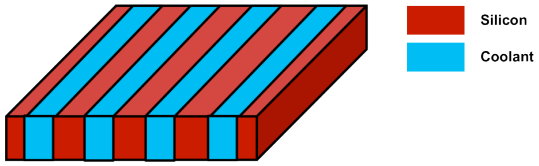


Figure 10: Microchannel cavity layer

Microchannel 2-resistor model

```
microchannel_2rm :

        height DVALUE ;

        channel length DVALUE ;

        wall length DVALUE ;

        wall material MATERIAL_ID  ;

        coolant flow rate DVALUE ;

        coolant heat transfer coefficient [ DVALUE |  top  DVALUE ,

                                                      bottom DVALUE ] ;

        coolant volumetric heat capacity DVALUE ;

        coolant incoming temperature DVALUE ;
```

Note that the only change from microchannel-4rm model is the omission of the first- and the last-wall widths and the side-way heat transfer coefficient, as was described in Section 5.A.ii. Also, remember that these heat transfer coefficients are effective "projections" of the heat transfer coefficients on the top and the bottom surfaces of the cavity. If you have the top, bottom and side heat transfer coefficients (HTCs) for a microchannel structure, you can compute these effective HTCs using the following formulae [2]:

$$\mathrm{HTC}_{\mathrm{top,eff}} = \frac{\mathrm{HTC}_{\mathrm{top}} \cdot \text{channel width} \quad + \quad \mathrm{HTC}_{\mathrm{side}} \cdot \text{channel height}}{\text{channel pitch}}$$

$$\mathrm{HTC}_{\mathrm{bottom,eff}} = \frac{\mathrm{HTC}_{\mathrm{bottom}} \cdot \text{channel width} \quad + \quad \mathrm{HTC}_{\mathrm{side}} \cdot \text{channel height}}{\text{channel pitch}}$$

Example

```
microchannel_2rm:
       height 100 ;
       channel length 50;
       wall length 50;
       wall material SILICON;
       coolant flow rate 42;
       coolant heat transfer coefficient
                           top 5.5698e-8,
                        bottom 5.0698e-8;
       coolant volumetric heat capacity
                               4.172e-12;
       coolant incoming temperature
                                    300;
```

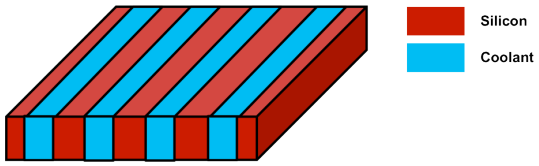


Figure 11: Microchannel cavity layer

Pinfins

```
       pinfin:

              height DVALUE ;

              pin diameter   DVALUE ;

              pin pitch     DVALUE ;

              pin distribution [inline | staggered] ;

              pin material MATERIAL_ID ;

              darcy velocity DAVLUE ;

              coolant volumetric heat capacity DVALUE;

              coolant incoming temperature  DVALUE;
```

where

- `height` (in µm) corresponds to the height of the pinfin layer. This must exactly correspond to the height of the pins in accordance with the 2RM-CTTM model,
- `pin diameter` and `pin pitch` are diameter and the pitch (all in µm) of the pins. 3D-ICE uses this information to compute the pin density and the porosity of the model,

- `pin distribution` defines the type of pinfin distribution- `inline` for inline (Fig. 4) and `staggered` for staggered (Fig. 5),
- `MATERIAL_ID` is the identifier of the material composing the pins (the ID must be previously declared in the materials section),
- `darcy velocity` is expressed (in µm/sec) and it refers to the darcy velocity in the cavity. This is used instead of the coolant flow rate because darcy velocities are more commonly measured quantities in experiments and CFD simulations due to their geometry independence, and more useful while expressing the heat transfer coefficient correlations in empirical studies.

Example (Pinfins inline)

```
pinfin :
       height 100 ;
       pin diameter   50 ;
       pin pitch     100 ;
       pin distribution inline ;
       pin material silicon ;
       darcy velocity 1.1066e+06 ;
       coolant volumetric heat capacity
                          4.172638e-12 ;
       coolant incoming temperature
                          300.0 ;
```

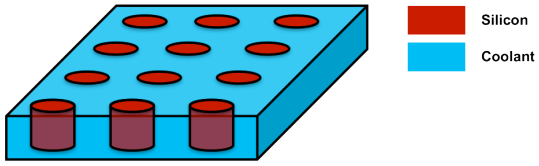


Figure 12: Pinfin inline cavity layer

Example (Pinfins staggered)

```
pinfin :
       height 100 ;
       pin diameter   50 ;
       pin pitch     100 ;
       pin distribution staggered ;
       pin material silicon ;
       darcy velocity 1.1066e+06 ;
       coolant volumetric heat capacity
                          4.172638e-12;
       coolant incoming temperature
                          300.0 ;
```

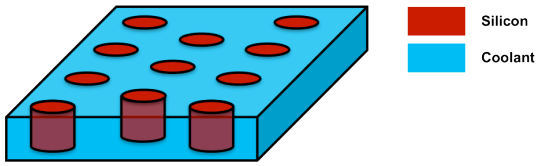


Figure 13: Channel layer

Note: If neither the conventional nor the liquid-cooled heat sink is declared in the Stack Description File, then the parsing will end with a warning message. Note that this would cause the temperatures to blow up unbounded with time in the presence of non-zero heat sources.

v. Stack

This section builds the vertical structure of the stack. The stack is composed of Dies (as previously declared) and layers and/or channels.

```
stack :

       [ layer   LL_ID  DVALUE MATERIAL_ID ; ]*

       [ channel CC_ID ; ]*

       [ die     DD_ID  DIE_ID floorplan "PATH" ; ]+
```

where

- `LL_ID`, `CC_ID` and `DD_ID` are identifiers used to name the stack elements and they can be used in the simulator code to refer to the corresponding element. They must be unique for each element,
- `MATERIAL_ID` is the identifier of the material (as previously declared) composing the declared layer,
- `DVALUE` is the height of the layer (in μm),
- `DIE_ID` is the identifier of a die (as previously declared) and `PATH` is the path to the floorplan file. This floorplan will be placed on the declared source layer in the definition of the die. The floorplan files contain information of the location and power dissipation activity of various floorplan components for the given die (see the description of Floorplan Files for more details). The same `DIE_ID` can be used multiple times (with different identifiers `DD_ID`) in a stack with the same or different floorplans, if identical/similar dies exist in a single IC.

Note that the keyword “channel” is used to specify the cavity in the stack- irrespective of which type of interlayer cavity (microchannels or pin fins) was declared in the stack descriptor.

The above grammar for the stack section is not the representative of a typical stack description and is only given for simplicity. Remember that your final stack sequence must satisfy the following:

- there must be at least one die
- it cannot begin or end with a cavity (i.e., liquid-cooled cavities can’t be the bottommost or the topmost layers in a stack)
- there cannot be two consecutive cavities
- cavities can be used only if previously declared
- layers are optional

Declaring a layer in a stack is not mandatory but we left this option to support stacks with irregular patterns of dies and channels or to build auxiliary layers (such as a bonding layer).

As in the case of defining dies, the final sequence of stack elements reflects their vertical disposition. The first stack element corresponds to the topmost element while the last element declared is closest to the PCB.

Example

```
material SILICON :
       thermal conductivity     1.30e-4;
       volumetric heat capacity 1.628e-12;
material BEOL :
       thermal conductivity     2.25e-6;
       volumetric heat capacity 2.175e-12;

conventional heat sink:
       heat transfer coefficient 1.0e-7;
       ambient temperature 300;

microchannel_4rm:
       height 100 ;
       channel length 50;
       wall length 50;
       first wall length 25;
       last  wall length 25;
       wall material SILICON;
       coolant flow rate 42;
       coolant heat transfer coefficient
                    top 5.7132e-8,
                    bottom 4.7132e-8;
       coolant volumetric heat capacity
                    4.172e-12;
       coolant incoming temperature
       300;

die TOP_IC:
       source 2 SILICON;
       layer 50 SILICON;
die BOTTOM_IC:
       layer 10 BEOL;
       source 2 SILICON;
       layer 50 SILICON;
dimension :
       chip length 10000, width 10000;
       cell length 50, width 50;

stack:
       die     MEMORY_DIE     TOP_IC    floorplan "./mem.flp";
       channel TOP_CHANNEL;
       die     CORE_DIE       BOTTOM_IC floorplan "./core.flp";
       channel BOTTOM_CHANNEL;
       layer   BOTTOM_MOST    10        BEOL;
```

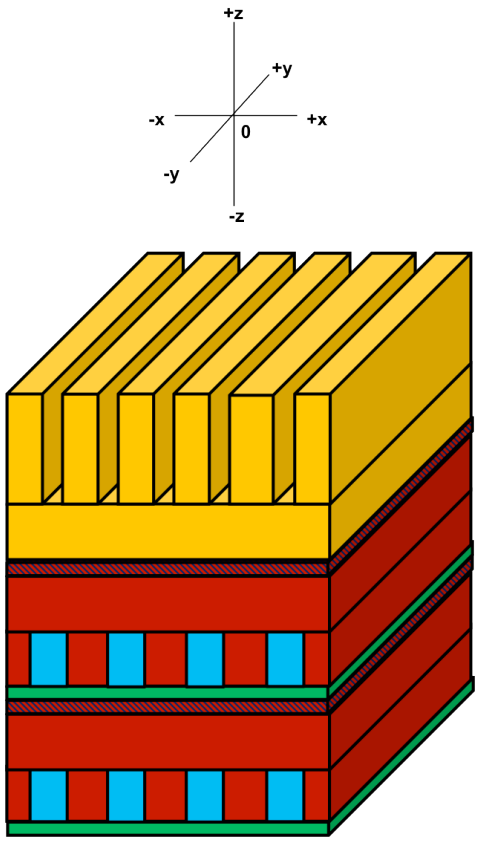


Figure 14: Complete stack

vi. Dimensions

This section of the Stack Description File declares the *xy* dimensions of the entire chip and the discretization sizes for the thermal cells (all in µm).

```
       dimension :

              chip length DVALUE , width DVALUE ;

              cell length DVALUE , width DVALUE ;
```

The entire chip is discretized based on the same cell length (along *x* direction) and width (along *y* direction) values. The discretization along the *z* direction is not specified, since the height of a thermal

cell is taken to be the same as the height of the layer in which it exists as shown for a layer in Fig. 15. However, if you want a finer discretization than that along the *z* direction, then you will have to split the layer into multiple layers of the same material stacked on the top of each other in the declaration of die/stack, as shown in Fig. 16 (here $h_1+h_2=h$).

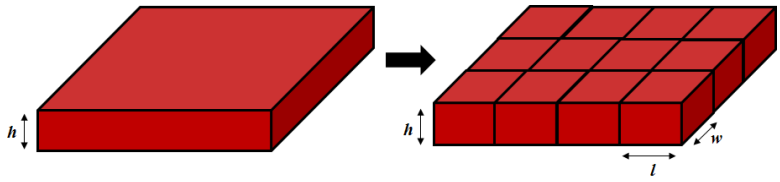


Figure 15: Discretization of a single layer

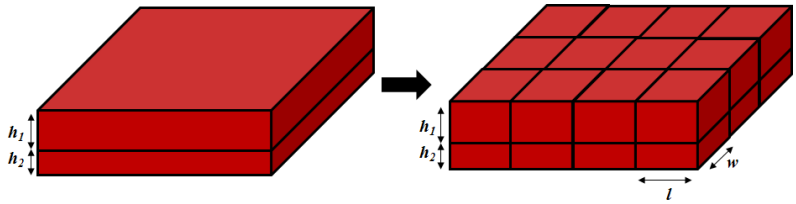


Figure 16: Discretization of a single layer split into 2 layers

Example

```
dimension :
        chip length 10000, width 10000;
        cell length 50, width 50;
```

If the microchannel 4-resistor model is used, the cell length is determined solely based on the cross sectional dimensions of channel and wall in the Channel section of the Stack Description File. This is because the 4RM-CTTM modeling used in 3D-ICE requires that the entire cross section of the microchannel be a part of the thermal cell. This means that the cell length at some position along the *x* direction in the model is dependent upon the channel or the wall cross sectional width at that point. Note that given different values of `channel width`, `channel pitch`, `first wall length` and `last wall length`, the discretization along this direction will be non-uniform in the simulator. For all other liquid-cooled cavity types, the cell length and cell width are necessary and sufficient to create a uniform discretization of the entire structure.

For the purpose of illustration, the discretized domain corresponding to the stacked structure built in Fig. 14 is shown in Fig. 17. Note that the `cell length` must still be declared in all cases, in spite of the fact that it will be ignored during the parsing when microchannel 4-resistor model cavity is used.

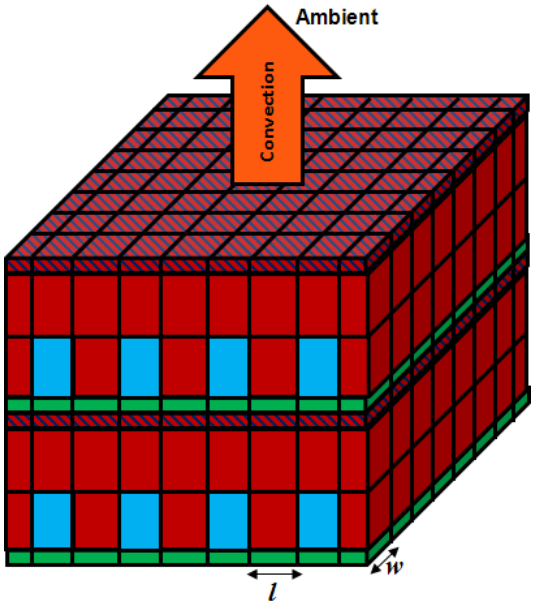


Figure 17: Discretized computational domain for the stack shown in Fig. 11

Problem Complexity

Remember that the dimensions of the chip, together with the dimensions of the thermal cells will directly affect the performance of the simulation. Indeed, the number of cells (nodes) influences both the amount of memory used by the library and the time needed to solve the linear system [1]. The main computational effort of the simulator is incurred during the execution of SuperLU and `blas` libraries. Specifically, the LU factorization of the system matrix is the most time/memory intensive. Hence, for large problem sizes, the availability of memory must be ensured to prevent this step from failing during the simulation.

vii. Analysis options

In 3D-ICE 2.0 either transient or steady state simulations can be performed.

Note to 3D-ICE 1.0 users: This entire section is new in 3D-ICE 2.0.

```
solver:

        [steady] | [transient step DVALUE, slot DVALUE] ;

        initial temperature DVALUE ;
```

where

- `steady` and `transient` indicate steady state and transient analysis respectively,
- `step` corresponds to the internal stepping time (in sec) to be used in the transient simulation in 3D-ICE,
- `slot` is the slot time (in sec) for which each power value in the floorplan file lasts for in the transient simulation (see Section 6.B). This value must be greater or equal to the stepping time value,

- `initial temperature` denotes the initial temperature (in K) for the simulation.

Example (steady state)

```
solver :
        steady;
        initial temperature 300.0;
```

Example (transient)

```
solver :
        transient step 0.02, slot 0.2;
        initial temperature 300.0;
```

viii. Output Instructions

A variety of thermal outputs can be obtained from 3D-ICE 2.0 simulations. In this last section of the stack descriptor, various output instructions can be provided for this purpose. The syntaxes for these instructions follow the model of conventional circuit simulators.

Note to 3D-ICE 1.0 users: This entire section is new in 3D-ICE 2.0.

```
output:

        [ T( VER_ID, DVALUE, DVALUE, "PATH" [, INSTANCE_ID]? ) ; ]*

        [ Tflp( DD_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ; ]*

        [ Tflpel( DD_ID.FLPEL_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ; ]*

        [ Tmap( VER_ID, "PATH" [, INSTANCE_ID]? ) ; ]*

        [ Tcoolantout( CC_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ; ]*
```

where

- `T` is an instruction to print the temperature of a particular thermal cell, identified by its three-dimensional coordinates in the thermal grid, in a text file. The output format consists of two columns- the time instance (in sec) against the corresponding temperature value (in K).

  `T( VER_ID, DVALUE, DVALUE, "PATH" [, INSTANCE_ID]? ) ;`

  - `VER_ID` specifies the "z" location of the thermal cell. This can assume a value of `LL_ID`, `DD_ID` or a `CC_ID` (see Section 6.A.v). If a die (`DD_ID`) is specified, then the thermal cell would be located in the source layer of that die,
  - `D_VALUE, D_VALUE` (in µm) specify the "x" and "y" locations of the thermal cell. An error message is printed if the location specified is outside of the computational domain,
  - `PATH` is the path of the text file in which the output must be written,
  - `INSTANCE_ID` is an optional parameter that can assume a value of `step`, `slot` or `final` and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions

that have the option `final` are executed. When no parameter is specified, `final` is assumed,

- `Tflp` is an instruction to print the temperature of a particular floorplan (the source layer of a particular die), identified by its die identifier (`DD_ID`). The output format consists of two columns- the time instance (in sec) against the corresponding temperature value (in K).

  `Tflp( DD_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ;`
  - `DD_ID` is the identifier of the die whose floorplan temperature must be printed.
  - `PATH` is the path of the text file in which the output must be written,
  - `OUTTYPE_ID` specifies the exact nature of the temperature to be reported, and can assume the values `maximum`, `minimum` or `average`, corresponding maximum, minimum and average temperature in the floorplan respectively,
  - `INSTANCE_ID` is an optional parameter that can assume a value of `step`, `slot` or `final` and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option `final` are executed. When no parameter is specified, `final` is assumed.

- `Tflpel` is an instruction to print the temperature of a particular floorplan element (the power block of the source layer of a particular die), uniquely identified by its die identifier (`DD_ID`) and floorplan element identifier (`FLPEL_ID`). The output format consists of two columns- the time instance (in sec) against the corresponding temperature value (in K).

  `Tflpel( DD_ID.FLPEL_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ;`
  - `DD_ID.FLPEL_ID` contains the die and the floorplan element identifier (see Section 6.B) which can uniquely identify a floorplan element in the entire stack,
  - `PATH` is the path of the text file in which the output must be written,
  - `OUTTYPE_ID` specifies the exact nature of the temperature to be reported, and can assume the values `maximum`, `minimum` or `average`, corresponding maximum, minimum and average temperature in the floorplan element area respectively,
  - `INSTANCE_ID` is an optional parameter that can assume a value of `step`, `slot` or `final` and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option `final` are executed. When no parameter is specified, `final` is assumed.
- `Tmap` is an instruction to print the temperature map of a particular layer in the stack, identified by its identifier. The output is printed in a matrix format in a text file, with each line representing a row of thermal cells in the thermal grid (counted in “y” direction), and every temperature value (in K) in each line corresponding to a thermal cell (columns, counted in the “x” direction).

Once the temperature map at a given time point is printed in this format, the temperature map at the next time point requested is printed from the next line in the same format. Hence, the total number of lines in the file is the product of the number of rows in the thermal grid and the number of instances the output is printed. In addition, whenever the instruction `Tmap` is used in a project, two other files, named "`x_axis.txt`" and "`y_axis.txt`" are printed which consist of the indices of the columns and rows (in µm), respectively, for these temperature maps. They can be used to visualize the data in the text files with the help of applications supporting graphical outputs such as Matlab.

```
Tmap( VER_ID, "PATH" [, INSTANCE_ID]? ) ;
```

- `VER_ID` identifies the layer for which the temperature map must be printed. This can assume a value of `LL_ID`, `DD_ID` or a `CC_ID` (see Section 6.A.v). If a die (`DD_ID`) is specified, then the source layer of that die is used,
- `PATH` is the path of the text file in which the output must be written,
- `INSTANCE_ID` is an optional parameter that can assume a value of `step`, `slot` or `final` and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option `final` are executed. When no parameter is specified, `final` is assumed.

- `Tcoolantout` is an instruction to print the temperature of the coolant at the outlet of a cavity identified by its channel identifier (`CC_ID`). The output format consists of two columns- the time instance (in sec) against the corresponding temperature value (in K).

```
Tcoolantout( CC_ID, "PATH", OUTTYPE_ID [, INTANCE_ID]? ) ;
```

- `CC_ID` is the identifier of the cavity whose outlet temperature must be printed,
- `PATH` is the path of the text file in which the output must be written,
- `OUTTYPE_ID` specifies the exact nature of the temperature to be reported, and can assume the values `maximum`, `minimum` or `average`, corresponding maximum, minimum and average outlet temperature respectively,
- `INSTANCE_ID` is an optional parameter that can assume a value of `step`, `slot` or `final` and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option `final` are executed. When no parameter is specified, `final` is assumed.

Examples

Two example stack descriptor files, one steady state and one transient, based on the above discussions are shown below

Example (steady state)

```
material SILICON :
       thermal conductivity     1.30e-4;
       volumetric heat capacity 1.628e-12;
material BEOL :
       thermal conductivity     2.25e-6;
       volumetric heat capacity 2.175e-12;

conventional heat sink:
       heat transfer coefficient 1.0e-7;
       ambient temperature 300;

microchannel_4rm:
       height 100 ;
       channel length 50;
       wall length 50;
       first wall length 25;
       last  wall length 25;
       wall material SILICON;
       coolant flow rate 42;
       coolant heat transfer coefficient
                     top 5.7132e-8,
                     bottom 4.7132e-8;
       coolant volumetric heat capacity
                     4.172e-12;
       coolant incoming temperature
       300;

die TOP_IC:
       source 2 SILICON;
       layer 50 SILICON;
die BOTTOM_IC:
       layer 10 BEOL;
       source 2 SILICON;
              layer 50 SILICON;
dimension :
       chip length 10000, width 10000;
       cell length 50, width 50;

stack:
       die     MEMORY_DIE     TOP_IC    floorplan "./mem.flp";
       channel TOP_CHANNEL;
       die     CORE_DIE       BOTTOM_IC floorplan "./core.flp";
       channel BOTTOM_CHANNEL;
       layer   BOTTOM_MOST    10        BEOL;

solver:
       steady ;
       initial temperature 300.0 ;

output:
       T( MEMORY_DIE, 5000, 3000, "output1.txt",final);
       Tmap( CORE_DIE, "output2.txt",  final );
       Tflpel(CORE_DIE.core1, "output3.txt", average);
       Tcoolantout(TOP_CHANNEL, "output4.txt", maximum);
```

Example (transient)

```
material SILICON :
       thermal conductivity     1.30e-4;
       volumetric heat capacity 1.628e-12;
material BEOL :
       thermal conductivity     2.25e-6;
       volumetric heat capacity 2.175e-12;

conventional heat sink:
       heat transfer coefficient 1.0e-7;
       ambient temperature 300;

microchannel_4rm:
       height 100 ;
       channel length 50;
       wall length 50;
       first wall length 25;
       last  wall length 25;
       wall material SILICON;
       coolant flow rate 42;
       coolant heat transfer coefficient
                    top 5.7132e-8,
                    bottom 4.7132e-8;
       coolant volumetric heat capacity
                    4.172e-12;
       coolant incoming temperature
       300;

die TOP_IC:
       source 2 SILICON;
       layer 50 SILICON;
die BOTTOM_IC:
       layer 10 BEOL;
       source 2 SILICON;
              layer 50 SILICON;
dimension :
       chip length 10000, width 10000;
       cell length 50, width 50;

stack:
       die     MEMORY_DIE     TOP_IC    floorplan "./mem.flp";
       channel TOP_CHANNEL;
       die     CORE_DIE       BOTTOM_IC floorplan "./core.flp";
       channel BOTTOM_CHANNEL;
       layer   BOTTOM_MOST    10        BEOL;

solver:
       transient step 0.02, slot 0.2 ;
       initial temperature 300.0 ;

output:
       T( MEMORY_DIE, 5000, 3000, "output1.txt",step);
       Tmap( CORE_DIE, "output2.txt",  slot );
       Tflp( MEMORY_DIE, "output3.txt", minimum, step);
       Tflpel(CORE_DIE.core1, "output3.txt", average, final);
       Tcoolantout(TOP_CHANNEL, "output4.txt", maximum);
```

B. Floorplan File

Every die in the stack must be related to a "Floorplan File" (*.flp), which essentially provides the power dissipation profile (or heat sources) for the simulation. Each Floorplan file must contain the list of functional blocks (cores, caches, memories, etc), their positions, and the power dissipation as a function of time.

Every functional block, here called *floorplan element*, is a rectangular area inside the die, laid out in the source layer. Each floorplan element has a unique identifier- the name it is assigned. In addition, the position and the dimensions of each floorplan element are given (in µm) based on the same Cartesian coordinates that was used for building the stack, with the origin at the SOUTH-WEST corner of the source layer. An example floorplan of a 1cmX1cm die with the reference coordinates is shown in Fig. 18. All the distances shown here are in µm.

A floorplan element in the Floorplan File is declared using the following syntax.

```
IDENTIFIER :

        position  DVALUE , DVALUE ;

        dimension DVALUE , DVALUE ;

        power values DVALUE [ , DVALUE ]* ;
```

where

- `IDENTIFIER` is the unique identifier used to name the floorplan element. This string must be unique within the floorplan file it belongs to but it can be used on a different file,
- `position`, is the (x,y) coordinate of the SOUTH-WEST corner of the floorplan element (in µm),
- `dimension` is the (length, width) dimensions of the floorplan element (in µm),
- The `DVALUE`(s) against the keyword `power values` are the list of power dissipation values (expressed in W) of the floorplan element for each *time slot* (scroll down for the explanation of time slots in 3D-ICE) separated by commas.

As an example, the declaration for the Cross Bar in Fig. 11 with 5 time slots would be as follows:

Example

```
Cross_Bar :
        position 2500, 3500;
        dimension 5000, 3000;
        power values 0.3, 0.3, 0.4, 0.2, 0.3;
```

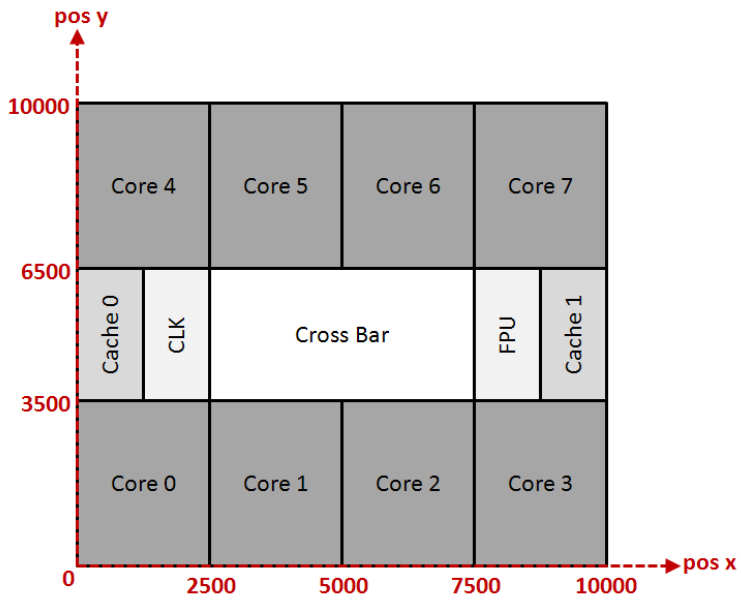


Figure 18: An example floorplan for a 1cmX1cm IC die

The following points must be kept in mind while writing the floorplan files:

- Floorplan elements must not overlap. During the parsing of the Floorplan File, the values describing the elements are checked to verify that all the elements are inside the chip and that they do not overlap.
- When the stack structure is discretized based on the given thermal cell dimensions, the power dissipated by a floorplan element is uniformly divided among the thermal cells contained *within its borders*. A thermal cell is considered to be *within the borders* of a given floorplan element if the CENTER of the thermal cell is inside the floorplan element. If the center of a thermal cell happens to be right on the border of 2 or more floorplan elements, then it is thrown into the floorplan on the NORTH/EAST by default. The distribution of thermal cells into different floorplan elements is illustrated in Fig. 19: the thermal cells highlighted in blue belong to the Cross Bar, the cells in green go to Core 1 and the cells in orange belong to Core 0. If you want to see exactly which thermal cells are covered by each floorplan element, you can use the `print_floorplan` or `print_all_floorplans` functions in the Library to print the range of rows (*y* indices of thermal cells) and columns (*x* indices of the thermal cells) covered by each floorplan element, after the input files have been parsed. For more details on these routines, please see Chapter 5.
- The same Floorplan File can be assigned to 2 or more dies in the Stack Description File if they happen to have identical structure and behavior in the design. Each floorplan element in the entire design, in that case, is uniquely identified by the `DD_ID` identifier of the corresponding die element and the `IDENTIFIER` of the floorplan element.
- If two dies in the stack have the same floorplan but during the simulation they have a different power dissipation activity, then 2 different Floorplan Files must be created for each die and assigned to the corresponding die element declarations in the Stack Description File. This is

because the power dissipations are directly linked to each floorplan element in the Floorplan File.

- It is possible to have gaps in the floorplan- regions where there is no floorplan element and hence, no power dissipation. The thermal cells in these regions will simply not be assigned any source value during the solving of system equations.

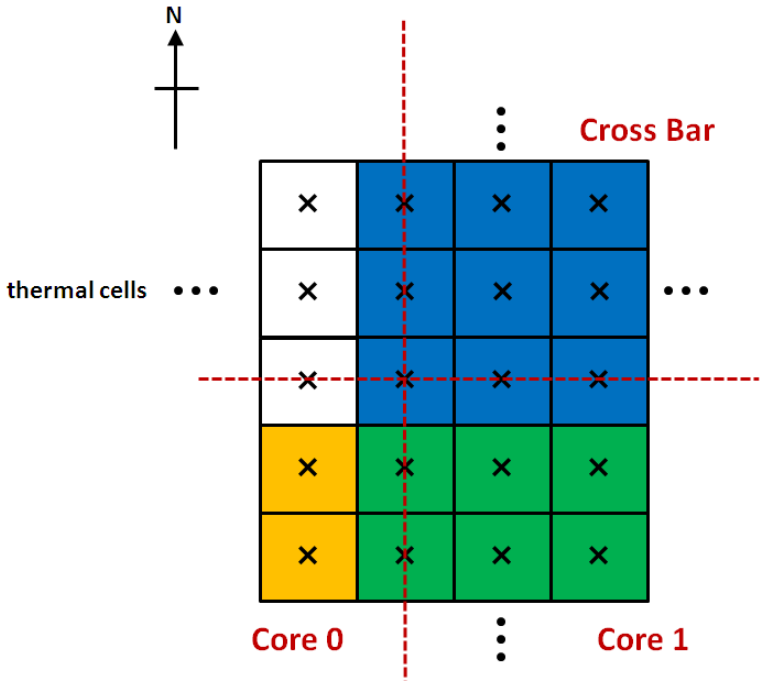


Figure 19: Distribution of thermal cells vis-à-vis floorplan elements

Time Slots

The entire time-interval of simulation (ToS) in 3D-ICE is divided into *time slots*- the minimum time duration for which the switching activity of the floorplan elements has been resolved. For example, if for a given design the switching activity (a measure of how much a floorplan component is active, directly related to its power dissipation) is sampled every 200ms during a 1 second ToS, then there are 5 time slots for the 3D-ICE simulation. And hence, there must be 5 values of power dissipation for each floorplan element declaration in the Floorplan File. Conversely, the number of power values for the floorplan element is interpreted as the number of time slots (NoTS) for the simulation. In all 3D-ICE simulations, it is assumed that the power dissipation in a particular thermal cell is CONSTANT during the period of a time slot- calculated based on the power value of the corresponding floorplan element, in which the thermal cell exists, for that time slot (see Fig. 13).

Note that time slot is different from *time step*, which is the discretization length of the time-domain for the numerical integration of the system of differential equations representing the entire thermal circuit created inside 3D-ICE. The exact durations of both time slot and time step are given when running the simulator. For further details, see Chapter 5.

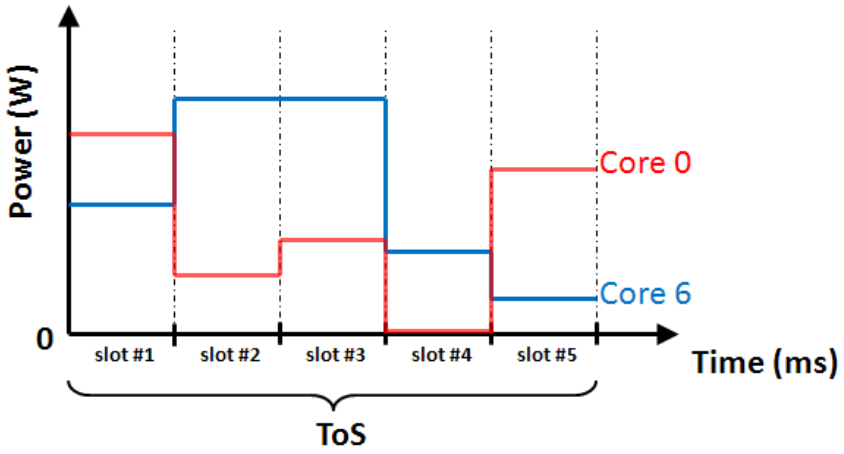


Figure 20: Power dissipation profile of Core 0 and Core 6 in Fig. 18

Since uniform sampling of the switching activity for all the components in an IC is assumed, the number of power values given for each floorplan element in the Floorplan File must be the same. Even when there is zero power dissipation for a particular floorplan element during some time slot (Core 0 at slot #4 in Fig. 20) or if a particular floorplan element has constant power dissipation during some or all time slots (Core 6 during slot #2 and slot #3 in Fig. 20), the corresponding power values must be mentioned explicitly for each time slot. The function `fill_stack_description` in 3D-ICE parses the Stack Description File first and then parses all the Floorplan File(s) starting from the bottom most die in the stack. Therefore, the number of power values (and hence NoTS) declared in the first floorplan element in that die will be used as a reference to check all the others elements and also, will eventually determine the ToS. Power values in excess of this reference number of slots will be discarded and reported with a warning message, while the lack of them will be considered as an error.

7. Running 3D-ICE

In the `3d-ice/bin/` folder, the main simulator file to be compiled is `3D-ICE-Emulator.c`. It has been written to parse and analyze any Stack Description File and the corresponding Floorplan Files placed in the same folder. Once compiled with the make command, the corresponding executable is acts as the thermal simulator application. To simulate a new thermal project, you must:

- create the Stack Description File and the corresponding Floorplan Files according to the instructions in Chapter 6
- run the following command in `3d-ice/bin/`
  `$./3D-ICE-Emulator PATH`

where `PATH` is the path to the Stack Descriptor File containing the description of the 3D-IC project.

Note to 3D-ICE 1.0 users: the information about step and slot time no longer appears in this command because it has been included in the stack descriptor file.

3D-ICE uses backward Euler method with constant time-stepping to solve the system equations. Hence, the solution is always numerically unconditionally stable. However, accuracy can be increased by reducing the time step value. The local truncation error of backward Euler method behaves as $O(h^2)$, where h is the time step. However, given a ToS, the number of time steps in the entire simulation is a $O(1/h)$ function. Hence, the upper bound of the total accumulated error at the final time point in the simulation behaves as

$$O(h^2).\ O(1/h)=O(h).$$

In other words, the total final error is approximately a linear function of the time step. For RC circuits, such as the thermal circuit that 3D-ICE solves, it is common practice to have at least 5 time steps for the duration of a rise time of the output temperature (defined as the time duration for the rise of temperature from 10% to 90% of its steady state value) to resolve the transients accurately.

A. Usage of the 3D-ICE 2.0 Software Thermal Library

3D-ICE, in addition to functioning as a stand-alone thermal simulator, can also serve as a software thermal library that can be used to build customized applications. Such requires knowledge about the organization and the use of various functions and data structures that are built into 3D-ICE. With the release of **3D-ICE 2.0, we are happy to announce that a new, useful tool for the visualization of the software library, called Doxygen [5], was used to build an online documentation of the 3D-ICE library**. It contains convenient access all information about the functions and the files in the library, including a hierarchical visualization of the various dependencies between them. This new documentation can be accessed by running in the `make doc` command inside the 3D-ICE folder, and then opening `3d-ice/doc/html/index.html` on your web browser. We urge users who are interested in employing 3D-ICE to build custom applications to refer to this documentation.

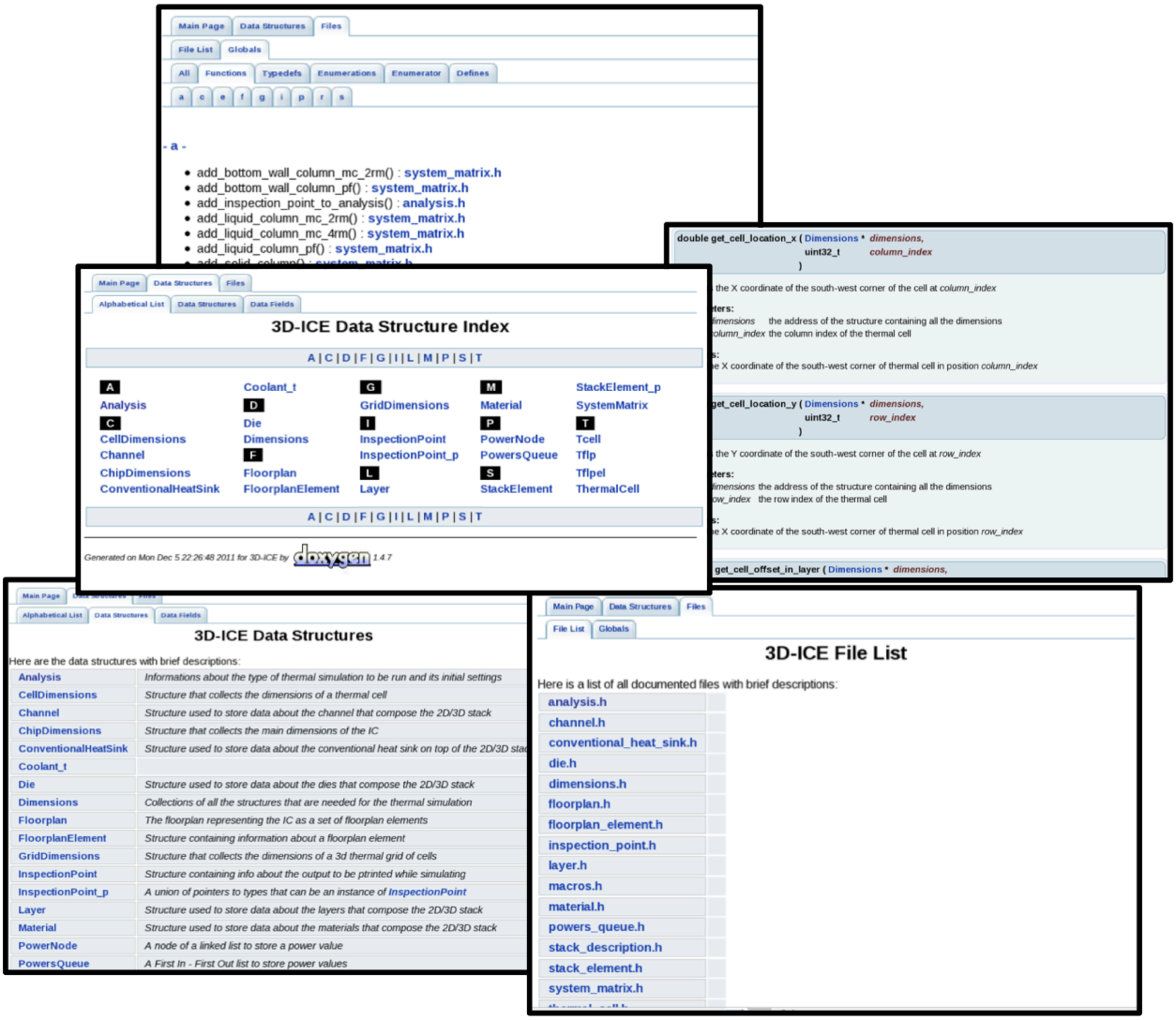


Figure 21: Doxygen documentation for 3D-ICE 2.0

Using 3D-ICE as a software library involves the following three steps.

Step 1: First, the problem must be initialized and the necessary data structures containing the 3D IC information must be filled. For this, the two main header files are to be included in the main file to access the homonym data structures.

- `stack_description.h`
- `analysis.h`
- `thermal_data.h`

For details about their content please refer to the description given in these files.

StackDescription

This data structure type collects all the data pertaining to the 3D IC structure and floorplans. An instance of this data type will then be related to the Stack Description File. The functions available to initialize, fill and empty an instance of a `StackDescription` variable are:

- `init_stack_description (StackDescription*)`
- `fill_stack_description ( StackDescription*, Analysis*, String_t)`
- `free_stack_description (StackDescription*)`

The functions to access the information in this data structure are in the doxygen documentation.

ThermalData

This data structure type collects all the data needed for the thermal simulation, such as the values of conductances in each cell, matrices representing the system equations, etc. The functions to initialize, fill and empty a `ThermalData` variable are

- `init_thermal_data (ThermalData*)`
- `fill_thermal_data ( ThermalData*, StackDescription*, Analysis*)`
- `free_thermal_data (ThermalData*)`

Just as an instance of `StackDescripiton` is tied to a Stack Description File, an instance of `ThermalData` depends upon an instance of `StackDescription` and `Analysis`. Hence it is necessary to initialize and fill the `StackDescription` and `Analysis` variables before filling the `ThermalData` variable. And for the rest of the simulation project, the three variables must be used in tandem since they refer to the same problem.

Step 2: Once the data structures have been filled, the next step is to execute the simulation. It can be done using the function.

- `emulate_step (ThermalData*, StackDescription*, Analysis*)`
- `emulate_steady (ThermalData*, StackDescription*, Analysis*)`

`emulate_step` increments the simulation time by a single time step and then terminates- this function can be called iteratively in a loop until the end of the ToS, controlled using the return variables of these functions which indicating whether or not the simulation has reached certain epochs (see the description in the files containing these functions for more details); while `emulate_steady` performs the steady simulation (one step simulation).

Step 3: Finally, outputs can be read from the thermal analysis during, and at the end of the simulation. There are functions in 3D-ICE that enable the extraction of the thermal state of a single floorplan element, an entire floorplan, a single channel outlet or all the coolant outlets in a given channel layer at any time during the simulation. It is also possible to read the temperature of a single thermal cell or print directly the thermal map (a matrix) for a specific layer in the stack. Some of these functions require you to refer to flooplan elements, layers etc. using the corresponding identifiers declared in the Stack Description File.

To learn more about all the functions and data structures descried above, please refer to the doxygen documentation of 3D-ICE in the `3D-ICE/doc/html/` folder.

B. Debugging of Thermal Simulation

For the purpose of debugging, several pre-processing options can be enabled to directly check the values computed during the construction of thermal data (thermal grid/circuit, system matrices, sources etc) before the simulation even starts. These options can be activated uncommenting the corresponding line in the file `3d-ice/sources/Makefile` and running the `make` command again (run the `make clean` command before building). As a consequence, messages will be redirected to `stderr`.

The debug options that are available are:

- `PRINT_THERMAL_CELLS` prints for every cell, the *cell ID* -defined as (layer, row, column, index)-, its dimensions, the 6 conductances (NORTH, SOUTH, EAST, WEST, TOP and BOTTOM), the c_{conv} constant (for liquid cells, see [1] for more details), its (heat capacity/time step) value, which is used in the construction of system matrices for backward Euler numerical integration and finaly, the parameters used to evaluate it (`volumetric heat capacity` and `delta_DVALUE`) are printed. The ID of the material of the cell is also printed at the beginning of a new layer of cells.
- `PRINT_SYSTEM_MATRIX` prints the content of the system matrix while it is filled. For every column in the system matrix ($\boldsymbol{G}+\boldsymbol{C}/h$, see [1] for more details), it prints the cell ID of the corresponding cell and the list of the nonzero coefficients indicating representing its neighbors. For every neighbor, it also prints the cell ID.
- `PRINT_SYSTEM_VECTOR` prints the content of the system vector (the Thermal state) at every time step.
- `PRINT_SOURCES` prints for every cell in a source layer, the cell ID, its dimensions, the source value for that cell, and also, the values used to evaluate that source value (floorplan `power values`, surface fraction of the floorplan element occupied by the cell and floorplan element `IDENTIFIER`). For every channel inlet cell (the southernmost cell along a channel), it prints the cell ID and the source value corresponding to the boundary condition, and also the parameters used to evaluate that source (`coolant incoming temperature` and c_{conv} constant). Remember that this debugging information will be printed every time the power values of the floorplan elements are read at the beginning of a time slot.

Note to 3D-ICE 1.0 users: you will find the debugging options described slightly changed from 3D-ICE 1.0.

8. References

[1] A Sridhar, A Vincenzi, M Ruggiero, T Brunschwiler, D Atienza, "3D-ICE: Fast compact transient thermal modeling for 3D-ICs with inter-tier liquid cooling", *Proceedings of the 2010 International Conference on Computer-Aided Design (ICCAD 2010)*, San Jose, CA, USA, November 7-11 2010.

[2] A Sridhar, A Vincenzi, M Ruggiero, T Brunschwiler, D Atienza, "Compact transient thermal model for 3D ICs with liquid cooling via enhanced heat transfer cavity geometries", *Proceedings of the 16th International Workshop on Thermal Investigations of ICs and Systems (THERMINIC'10)*, Barcelona, Spain, 6-8 October, 2010.

[3] T Brunschwiler, S Paredes, U Drechsler, B Michel, W Cesar, Y Leblebici, B Wunderle, H Reichl, "Heat-removal performance scaling of interlayer cooled chip stacks", *Proceedings of the 2010 IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems* (ITHERM '10), pp. 1-12, June 2010.

[4] T A Davis and E P Natarajan, "Algorithm 8xx: KLU, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software*, vol.5, no.1, pp.1–14, 2010.

[5] Doxygen, URL: http://www.stack.nl/~dimitri/doxygen/.