KitFox: Multi-Physics Libraries for Integrated Power, Thermal, and Reliability Simulation of Multicore Microarchitecture

William Song[†], Saibal Mukhopadhyay[†], Sudhakar Yalamanchili[†], and Arun Rodrigues[‡]

[†]School of Electrical & Computer Engineering, Georgia Tech, Atlanta, GA
[‡]Sandia National Labs, Albuquerque, NM

Apr. 2015

Contents

1 Introduction

This document describes the design methodology, implementation, and usage of KitFox. KitFox is a modeling framework of multi-physics libraries for integrated power, thermal, and reliability simulation of multicore microarchitectures. The goal of KitFox framework is to facilitate the microarchitectural explorations at the intersection of applications, microarchitectural performance, and various physical phenomena including energy, power, thermal, and lifetime reliability. KitFox is based on the integration of models that simulate different physical properties, including the popular tools in the architecture community such as HotSpot and McPAT. Each model in KitFox is encapsulated into a standard class called *library* (e.g., energy, thermal, or reliability library). Any new or updated models can be seamlessly integrated into KitFox by encapsulating them into the respective library. There are no hidden software dependencies between libraries, and all interactions are explicitly managed via the standard library interface that avoids modifications to the third-party tools. The interface to the multicore simulator is through an application programming interface (API) that relieves the microarchitecture modeler from orchestrating all multi-physics interactions. Figure 1 illustrates the concept of standard libraries and user API. Towards this goal, KitFox has the following features and contributions.

- *Standard Library*: Each model is encapsulated into standard class called *library* and integrated to KitFox. No cross-dependency in software integration is created between the models, and any new models can be added to the framework.
- *Interacting Library Models*: Interactions between the physical models are orchestrated inside KitFox via the standard library interface. It minimizes user involvement in data management to coordinate multiple libraries and subclass models.
- *Simple User API*: KitFox provides a set of user API functions to be used in microarchitecture simulations. The same function is used for calculating the same physical property via the standard library interface, regardless of which individual model is used inside the framework.
- *Configurable Processor Model*: KitFox provides a configuration method that users can define the microarchitectural and physical hierarchy of a processor package and associate individual physical models with constituent processor components to be simulated.
- *Parallel Simulation via MPI Interface*: KitFox supports parallel simulation of physical models via the message passing interface (MPI). A microarchitecture simulator and KitFox can execute in parallel in separate MPI processes, and KitFox itself can also be parallelized into multiple MPI ranks.

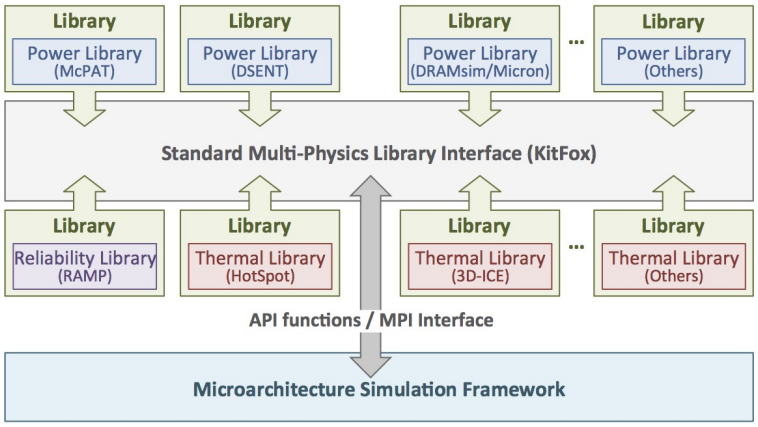

Figure 1. Multiple physical models are integrated into KitFox as *libraries*. It standardizes the integration and use of models, and provides microarchitecture simulators with a set of API functions.

2 Installation

2.1 Prerequisite

The following packages are required to be installed in the system in prior to building KitFox and supported modeling tools:

- g++ v4.2 or later
- open-mpi v1.4 or later
- libconfig++
- bison v2.4.1 and flex for 3D-ICE

2.2 Download

The latest stable version of KitFox can be obtained from the following website:

```
http://manifold.gatech.edu/projects/kitfox/
```

The latest development copy in progress can be obtained from the repository:

```
$ svn co https://svn.ece.gatech.edu/repos/Manifold/trunk/code/models/kitfox/
```

2.3 Files and Directories

KitFox is organized with the following files and directories:

```
kitfox
    | ... component.h
    | ... component.cc
    | ... configuration.h
    | ... configuration.cc
    | ... kitfox-defs.h
    | ... kitfox.h
    | ... kitfox.cc
    | ... library.h
    | ... library.cc
    | ... queue.h
    | ... queue.cc
    | ... communicator/
        | ... kitfox-client.h
        | ... kitfox-net.h
        | ... kitfox-server.h
        | ... kitfox-server.cc
    | ... config/
        | ... dramsim.config
        | ... dsent.config
        | ... intsim.config
        | ... mcpat.config
        | ... tsv.config
        | ... failure.config
        | ... 3dice.config
        | ... hotspot.config
        | ... microfluidics.config
    | ... library/
        | ... energylib_dramsim/
            | ... energylib_dramsim.h
            | ... energylib_dramsim.cc
            | ... patch/
```

```
        | ... tarball/
    | ... energylib_dsent/
        | ... energylib_dsent.h
        | ... energylib_dsent.cc
        | ... patch/
        | ... tarball/
    | ... energylib_intsim/
        | ... energylib_intsim.h
        | ... energylib_intsim.cc
        | ... patch/
        | ... tarball/
    | ... energylib_mcpat/
        | ... energylib_mcpat.h
        | ... energylib_mcpat.cc
        | ... patch/
        | ... tarball
    | ... energylib_tsv/
        | ... energylib_tsv.h
        | ... energylib_tsv.cc
        | ... patch/
        | ... tarball
    | ... reliabilitylib_failure/
        | ... reliabilitylib_failure.h
        | ... reliabilitylib_failure.cc
        | ... patch/
        | ... tarball/
| ... thermallib_3dice/
        | ... thermallib_3dice.h
        | ... thermallib_3dice.cc
        | ... patch/
        | ... tarball/
| ... thermallib_hotspot/
        | ... thermallib_hotspot.h
        | ... thermallib_hotspot.cc
        | ... patch/
        | ... tarball/
| ... thermallib_microfluidics/
        | ... thermallib_microfluidics.h
        | ... thermallib_microfluidics.cc
        | ... patch/
        | ... tarball/
```

2.4 Building Models

2.4.1 Downloading Models

NOTE: KitFox does NOT possess the copyrights of the following supported models. All terms and conditions are subject to what original developers present. KitFox does not guarantee the correctness or identical performance as the original distributions.
We strongly recommend to acquire the models through developers' distributions. Downloading the compatible version of each model is scripted in the Makefile via a wget command, but not all models are supported in this way due to developers' distribution methods. The paths to obtain the models are listed below:

- **DRAMSim2**
  - Version: v2.2.2
  - Source: `https://github.com/dramninjasUMD/DRAMSim2`
  - Download destination: library/energylib_dramsim/tarball/
- **DSENT**
  - Version: v0.91

- Source: `https://sites.google.com/site/mitdsent/download`
- Download destination: library/energylib_dsent/tarball/

- **McPAT**
  - Version: v0.8
  - Source: `http://www.hpl.hp.com/research/mcpat`
  - Directory destination: library/energylib_mcpat/tarball/
- **3D-ICE**
  - Version: v2.1
  - Source: `http://esl.epfl.ch/3d-ice/download.html`
  - Dependency (SuperLU-4.3): `http://crd-legacy.lbl.gov/~xiaoye/SuperLU`
  - Download destination: library/thermallib_3dice/tarball/
- **HotSpot**
  - Version: v5.02
  - Source: `http://lava.cs.virginia.edu/HotSpot/download_form2.html`
  - Download destination: library/thermallib_hotspot/tarball/

The tarball of each model has to be downloaded to the tarball directory of the corresponding model. For instance, download the compatible version of McPAT (mcpat0.8_r274.tar.gz) to library/energylib_mcpat/tarball/. Other models not listed above were developed with KitFox, and the developers agreed to distribute them along with KitFox. In each model, a patch file provides corrections to the model to solve compile errors due to the integration. These changes are not intended to alter the calculated results of the original version, but we do not guarantee the correctness or identical performance as original distributions.

2.4.2 Compiling Models

At the main directory of KitFox, each model can be compiled individually via the following command:

```
$ make <model name>
```

For instance, make mcpat compiles the McPAT source files and creates an archive file libmcpat.a in the main directory. model name has to be one of the following:

```
$ make dramsim
$ make dsent
$ make intsim
$ make mcpat
$ make tsv
$ make failure
$ make 3dice
$ make hotspot
$ make microfluidics
```

Alternatively, all models included in KitFox can be built by using the following command:

```
$ make models
```

2.5 Build KitFox

After the models that are to be used are compiled, KitFox can be built using the following command:

```
$ make kitfox
```

This command creates an archive file libKitFox.a that has to be linked to the architecture simulator along with the archive files of built models (e.g., libmcpat.a). Note that only the compiled models create archive files and are compiled with KitFox.

There are two different ways to clean the KitFox files. The first command cleans the object and archive files of KitFox. Re-compiling KitFox requires only make kitfox command and does not need to re-build the models.

```
$ make clean
```

The second command completely cleans all the objects and executables of the built models and KitFox. Re-compiling KitFox requires to repeat all the steps from building the models.

```
$ make distclean
```

2.6 Linking to Architecture Simulator

When compiling the user architecture simulator with KitFox, the archive files of the models and KitFox have to be linked. If some models are not build in the previous steps, they should not be linked.

```
CPPFLAGS += -I$(KITFOX_DIRECTORY) -I$(KITFOX_DIRECTORY)/communicator
LDFLAGS += -lKitFox -ldramsim -ldsent -lintsim -lmcpat -ltsv -lfailure -l3dice -lhotspot
           -lmicrofluidics -L$(KITFOX_DIRECTORY)
```

3 Libraries

KitFox provides a standardized method to integrate various implementations of physical modeling tools. In KitFox, standard classes called *libraries* are created, where each library hosts different type of model; *energy*, *thermal*, *reliability libraries*:

- **Energy Library**
  - Estimating per-access energy of different access types (e.g., logical switching, read, write, etc.)
  - Area calculation based on circuit-level models
  - Runtime update of variables (e.g., voltage, clock frequency, etc.)
- **Thermal Library**
  - Floor-planning and power-grid mapping
  - Calculation of steady-state or transient temperatures
  - Runtime update of variables (e.g., ambient temperature, coolant flow rate, etc.)
- **Reliability Library**
  - Calculation of transient failure rates based on operating conditions
  - Runtime update of variables (e.g., temperature, voltage, etc.)

The following is the baseline model library class that derives different library classes:

```
/* Base Library Class defined in library.h */
class model_library_t {
public:
    model_library_t(pseudo_component_t *PseudoComponent, int Type);
    virtual ~model_library_t();

    /* A model library must support runtime update of parameters.
       This function is also used as a callback function when new data are inserted
       into pseudo component queues. */
    virtual void update_library_variable(int type, void *Value, bool isLibraryVariable=false)=0;

    /* Initialize a subclass model. */
    virtual void initialize(void)=0;

    /* Type: energy, thermal, or reliability model */
    int type;

protected:
    /* The pseudo component that this model library is linked at. */
    pseudo_component_t *pseudo_component;
};
```

3.1 Energy Library

Power modeling tools are encapsulated as the subclass of the energy library. Power (or energy) is characterized at architecture-level components whose circuit-level behaviors are estimated from supported tools, based on technology parameters and architectural component configurations. For each modeled component in the simulator, a model of the energy library estimates per-access dynamic energies of distinct access types (e.g., read/write, logical switching, etc) and leakage power. Total dynamic energy is calculated by multiplying estimated per-access energies with corresponding types of access counters that can be collected from microarchitecture simulations. Leakage power is exponentially dependent on temperature states, and it requires thermal analysis for accurate power modeling. The following is the definition of the energy library class that hosts power modeling tools.

```
/* Energy Library defined in library.h */
class energy_library_t : public model_library_t {
public:
```

```
    energy_library_t(pseudo_component_t *PseudoComponent, int Type);
    virtual ~energy_library_t();

    /* Calculate unit energy per access. */
    virtual unit_energy_t get_unit_energy(void)=0;

    /* Calculate TDP power. */
    virtual power_t get_tdp_power(Kelvin MaxTemperature=373.0)=0;

    /* Calculate runtime power. */
    virtual power_t get_runtime_power(Second Time, Second Period, counter_t Counter)=0;

    /* Calculate area (if available). */
    virtual MeterSquare get_area(void)=0;

    /* Type: specific energy library model (e.g., McPAT) */
    int type;
};
```

For each model being integrated into KitFox, a *wrapper* class is created. The wrapper class is defined as the subclass of one of the library classes. It includes header/source files of the tool to be integrated and re-defines the usage of the model according to the virtual functions of the corresponding library class. Thus, the models of the same library type can be used in the identical way. The following is the example of McPAT wrapper class being defined as the subclass of the energy library.

```
/* McPAT Wrapper Class */
class energylib_mcpat : public energy_library_t {
public:
    energylib_mcpat(pseudo_component_t *PseudoComponent);
    ~energylib_mcpat();

    /* Virtual functions of base classes */
    virtual void initialize();
    virtual unit_energy_t get_unit_energy();
    virtual power_t get_tdp_power(Kelvin MaxTemperature=373.0);
    virtual power_t get_runtime_power(Second Time, Second Period, counter_t Counter);
    virtual MeterSquare get_area();
    virtual bool update_library_variable(int Type, void *Value, bool IsLibraryVariable=false);

private:
    /* McPAT input classes */
    ParseXML XML_interface;
    InputParameter input_p;
    ...

    /* McPAT library models */
    ArrayST *McPAT_ArrayST;
    dep_resource_conflict_check *McPAT_dep_resource_conflict_check;
    ...

    /* Tuning parameters used in the wrapper class */
    int energy_model, energy_submodel;
    double clock_frequency;
    double energy_scaling, area_scaling, scaling;
    ...
};
```

3.2 Thermal Library

Temperature modeling tools are encapsulated as the subclasses of the thermal library. Temperature is characterized at the package level. A processor package is represented as the stack of layers with thermal grids. Each thermal grid cell is expressed as a thermal RC connection. Calculating thermal field over a processor package is equivalent to solving the differential equations of this thermal RC network. Components representing a processor on the die are organized and represented by *floorplans*. Each floorplan component represents a set of microarchitectural units, where the power dissipation of each unit is estimated from the linked energy library model and microarchitectural timing simulation. The power estimates of floorplans are supplied to a gridding process that calculates the power at each cell in a finer resolution (configurable). This fine-grained power grid is the input to the thermal model that computes the temperature field over this grid. Changes in temperature are coupled to leakage power creating a feedback loop between the energy and thermal libraries.

```
/* Thermal Library defined in library.h */
class thermal_library_t : public model_library_t {
public:
    thermal_library_t(pseudo_component_t *PseudoComponent, int Type);
    virtual ~thermal_library_t();

    /* Calculate transient or steady-state temperature. */
    virtual void calculate_temperature(Second Time, Second Period)=0;

    /* Returns 3D temperature grid stack of power source layers. */
    virtual grid_t<Kelvin> get_thermal_grid(void)=0;

    /* Returns representative floorplan temperature.
    virtual Kelvin get_floorplan_temperature(Comp_ID ComponentID,
                                             int Type=KITFOX_TEMPERATURE_MAPPING_UNKNOWN)=0;

    /* Returns point temperature. */
    virtual Kelvin get_point_temperature(Meter X, Meter Y, Index Layer)=0;

    /* Map floorplan power. */
    virtual void map_floorplan_power(Comp_ID ComponentID, power_t FloorplanPower)=0;

    /* Add floorplan. */
    void add_floorplan(std::string ComponentName, Comp_ID CompnentID);

    /* Returns total number of floorplans. */
    const Index get_floorplan_counts(void);

    /* Returns a pseudo component ID of ComponentName. */
    const CompID get_floorplan_id(std::string ComponentName);
    const char* get_floorplan_name(Comp_ID ComponentID);

    /* Returns ith pseudo component in the map. */
    const Comp_ID get_floorplan_id(unsigned i);

    /* Type: specific thermal library model (e.g., HotSpot) */
    int type;

private:
    std::map<std::string, Comp_ID> floorplan_name_map;
    std::map<Comp_ID, std::string> floorplan_id_map;
};
```

3.3 Reliability Library

Gradual device wear leads to permanent failures. Phenomena that lead to device wear include negative bias temperature instability (NBTI), hot carrier injection (HCI), time-dependent dielectric breakdown (TDDB), etc. KitFox utilizes detailed interactive physics behaviors to calculate failure rates and predict lifetime reliability. Transient failure rates are calculated with respect to time-varying stress conditions including voltage and thermal stresses.

```
/* Reliability Library defined in library.h */
class reliability_library_t : public model_library_t {
public:
    reliability_library_t(pseudo_component_t *PseudoComponent, int Type);
    virtual ~reliability_library_t();

    /* Calculate failure rate. */
    virtual Unitless get_failure_rate(Second Time, Second Period, Kelvin Temperature,
                                      Volt Vdd, Hertz ClockFrequency)=0;

    /* Type: specific thermal library model */
    int type;
};
```

3.4 Physical Interactions

In KitFox framework, multiple physical models are concurrently simulated, and their interactions are captured at user-defined sampling rate, typically invoked by cycle-level microarchitecture simulation models. Figure 2 depicts an architecture-level abstraction of interactions between multiple physical properties and associated models. Execution of workloads through microarchitecture simulation generates switching activities of functional components. Architectural activities are represented with access counters and used to calculate dynamic power dissipation of modeled components. Leakage power is estimated by assuming a constant temperature during the sampling interval. Power results are mapped onto user-created thermal floorplans, and temperature is calculated based on the spatial distribution of input power and thermal grid states. Changes in temperature incur thermal-leakage power feedback, and recalculated leakage power is used for temperature calculation in the next sampling period. Reliability characteristics (i.e., failure rates) of modeled components are calculated with respect to time-varying operating conditions (i.e., voltage and thermal stresses). The chain of interactions create a loop and is repeated at every sampling interval. Execution controls such as voltage and frequency scaling may intervene in this chain of events and dynamically change operating conditions and resulting behaviors of physical phenomena.

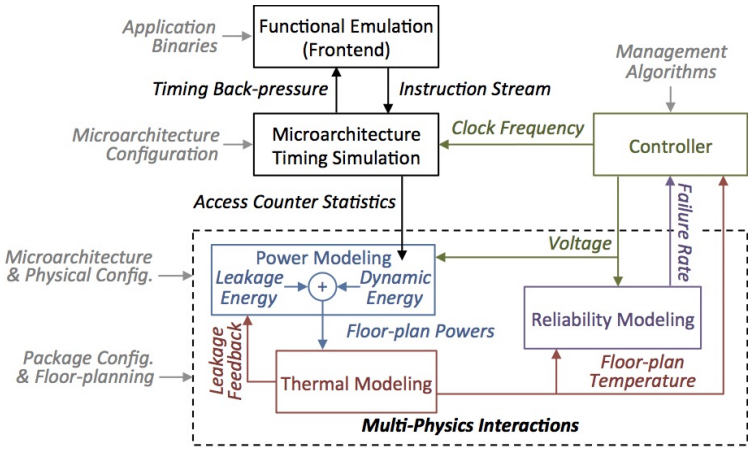

Figure 2. Architecture-level abstraction of interactions between multiple physical properties and microarchitecture. These interactions are captured concurrently during runtime simulation.

4 Configuration

In KitFox, a processor is represented as the hierarchy of *pseudo components*. Pseudo components are abstract units for which library models estimate physical phenomena. Different physical properties are characterized at different levels of processor abstraction. For instance, power is characterized at microarchitecture or circuit-level components using architectural activity counts. Temperature is calculated at the package level, based on power distribution on the die. Thus, pseudo components can represent different levels of processor abstraction, depending on which library models they are associated with and which physical properties are characterized. A pseudo component may represent a microarchitectural component when an energy library model is associated with it to calculate power or energy dissipation. Or it may be a processor package if a thermal library model is attached. The pseudo component hierarchy can be flexibly composed to simulation different processor designs. There is no inherent restriction on the number of levels in the pseudo component tree, and each pseudo component can have as many sub-components as necessary. Figure 4 illustrates an example of how KitFox framework serves to interface pseudo components and libraries to simulate a processor design. The simulated microarchitecture is decomposed into basic components (shown as "sources" in the figure), where power is estimated with energy library models. Each energy library may derive a different sub-class tool, and it enables choosing the most appropriate model for different microarchitectural components. Pseudo components can be grouped into another upper-level pseudo component (shown as "floorplans" in the figure) depending on their microarchitectural and technological similarities (e.g., core, memory). Higher-level components may represent larger processor units such as cores or regions on the die. The root component in the example represents the package and is linked to a thermal library model. It can designate any descendant components in the tree as its constituent floorplans. Some intermediate components without linked libraries can also be created for the convenience of processor description or data collection. Every pseudo component includes data queues to store computed results of library models and shared data (e.g., voltage, frequency) for cross-referencing between pseudo components.

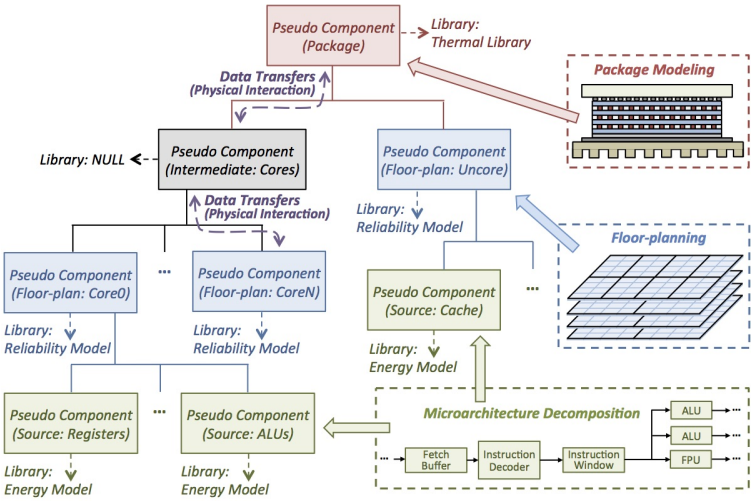

Figure 3. A processor is represented as the *pseudo component hierarchy* in KitFox framework. Pseudo components are physically defined units where associated libraries estimate physical phenomena. Physical interactions are emulated by transferring data between the data queues of pseudo components.

4.1 Single Component Example

The following shows an example of defining a pseudo component in the input configuration file. To define a pseudo component, it starts with the setting component. In this example, a pseudo component package is created. It has temperature and power data queues, each with a single entry; detailed discussion about the queue management will be covered later in the document (Section 7). library defines the parameters of the physical model associated with the package. 3D-ICE is used for package thermal model in this example,

and transient simulation mode is used. An example configuration to define 3D-ICE model can be found in config/3dice.config″ with 3D-stacked layers and a micro-channel model. Similarly, refer to config directory or Appendix to find how to use individual models integrated in KitFox.

```
component: {
    package: { // package component
        data_queue = ["KITFOX_DATA_TEMPERATURE", "KITFOX_DATA_POWER"];
        queue_size = 1;
        /* Parameters of the physical model */
        library: {
            model = "3d-ice"; // Use 3D-ICE for package thermal model
            thermal_analysis_type = "transient";
            grid_rows = 64;
            grid_columns = 64;
            ... // And more 3D-ICE parameters to be defined
        };
    };
};
```

4.2 Multi-Level Components, Multiple Models, and Shared Parameters

Pseudo components can be recursively created to construct multiple levels of pseudo component hierarchy. component within the package lists the sub-components whose name paths are delimited by dots, package.cores.core0 for example. In the example below, package.cores is a dummy component that is not associated with any physical models. Dummy components can be created for better hierarchical processor description or data management purpose that will be described later in the document (Section 7). package.cores.core0 and package.cores.core1 are the floorplans of the thermal library associated with the root component package. The floorplans are associated with the failure models for reliability modeling. In package.cores.core0, voltage variable is defined as 0.9V, but package.cores.core1 does not define the voltage that is necessary for the failure model. If a required variable is not defined, KitFox recursively searches the parameter in the components along the path to the root when parsing the input configuration file. In this example, voltage is defined in the root component package, and therefore package.cores.core1 uses 1.0V as the initial voltage input. The recursive search of parameters avoids defining the same parameter of the same value multiple times across pseudo components and simplifies the configuration by defining the common parameters in the parent components. Hence, the parameters common to multiple, distinct physical models can be defined only once and shared.

```
component: {
    package: { // package component
        /* Parameters of the physical model */
        library: {
            model = "3d-ice";
            ... // And more 3D-ICE parameters to be defined

            /* Common library parameters under the package */
            clock_frequency = 2.0e9; // 2GHz clock frequency
            voltage = 1.0; // 1V supply voltage

            floorplan = [
                "package.cores.core0",
                "package.cores.core1",
                "package.uncore.l2cache"
            ];
        };
        component: {
           cores: { // package.cores component
                library: {
                    model = "none"; // No physical model is associated.
```

```
                };
                component: {
                   core0: { // package.cores.core0 component
                        library: {
                            model = "failure"; // Use Failure model for reliability
                            voltage = 0.9; // Overwritten supply voltage
                            ... // And more Failure model parameters to be defined
                        };
                    };
                core1: { // package.cores.core1 component
                        library: {
                            model = "failure"; // Use Failure model for reliability
                            ... // And more Failure model parameters to be defined
                        };
                    };
                };
            };
            uncore: { // package.uncore component
                library: {
                    model = "failure"; // Use Failure model for reliability
                    ... // And more Failure model parameters to be defined
                };
                component: {
                   l2cache: { // package.uncore.l2cache component
                        library: {
                            model = "mcpat"; // Use McPAT for L2$ power model
                            energy_model = "cache"; // Use cache model in McPAT
                            line_size = 64; // Cache line size
                            ... // And more McPAT parameters to be defined
                        };
                    };
                };
            };
        };
    };
};
```

The pseudo components created in KitFox can be identified in two ways; string *name* and integer *ID*. Name is the component path such as package.cores.core0, and ID is a sequentially assigned integer number by KitFox when creating the pseudo components. KitFox provides functions to acquire the pseudo component ID with the name, or the name can be retrieved with the ID. Detailed will be covered in the document (Section 6 or 7).

5 Data Types

The same physical quantity is expressed by different notations, units, or even data formats by integrating multiple models, especially the third-party tools. For instance, temperature in Celsius, Fahrenheit, or Kelvin in data types of int, unsigned, float, double. KitFox defines the data types of shared physical quantities used across multiple physical models.

```
typedef int Index;            // KITFOX_DATA_INDEX
typedef uint64_t Comp_ID;     // Pseudo Component ID
typedef uint64_t Count;       // KITFOX_DATA_COUNT
typedef double Meter;         // KITFOX_DATA_LENGTH
typedef double MeterSquare;   // KITFOX_DATA_AREA
typedef double MeterCube;     // KITFOX_DATA_VOLUME
typedef double Second;        // KITFOX_DATA_TIME, KITFOX_DATA_PERIOD
typedef double Hertz;         // KITFOX_DATA_CLOCK_FREQUENCY
typedef double Volt;          // KITFOX_DATA_VOLTAGE, KITFOX_DATA_THRESHOLD_VOLTAGE
typedef double Joule;         // KITFOX_DATA_ENERGY
typedef double Watt;          // KITFOX_DATA_POWER, KITFOX_DATA_TDP
typedef double Kelvin;        // KITFOX_DATA_TEMPERATURE, KITFOX_DATA_AMBIENT_TEMPERATURE
typedef double Unitless;      // KITFOX_DATA_FAILURE_RATE
```

5.1 Dimension

Dimension data (KITFOX_DATA_DIMENSION) are used to define the geometry of the component, e.g., floorplans. Dimension uses the left-bottom corner for the orientation. Note that the area can be calculated by a physical model without known width and height information, so area is not necessarily the same as width × height. If the area is non-zero, get_area function returns area, or otherwise this function returns width × height.

```
class dimension_t {
public:
    dimension_t();

    Index die_index; // Index of the processor die in 3D package (0 by default and 2D)
    char die_name[MAX_KITFOX_COMP_NAME_LENGTH]; // Name of the processor die
    Meter left, bottom; // Left and bottom orientation of the component
    Meter width, height; // x, y-dimensional length
    MeterSquare area; // Area

    const MeterSquare get_area(); // Returns area (if non-zero) or width*height
    const Meter get_center_x() const; // Returns left+width/2
    const Meter get_center_y() const; // Returns bottom+height/2
    const Meter get_left() const; // Returns left
    const Meter get_right() const; // Returns left+width
    const Meter get_bottom() const; // Returns bottom
    const Meter get_top() const; // Returns bottom+height
    void clear(); // Reset function
};
```

5.2 Grid

Thermal grid (KITFOX_DATA_THERMAL_GRID) stores 3D mesh data of thermal grid cells. Note that this data cannot be transmitted across MPI processes.

```
template <typename T>
class grid_t {
public:
    grid_t();
    grid_t(Meter CellWidth, Meter CellHeight,\
```

```
            unsigned Columns, unsigned Rows, unsigned Dies);

    unsigned cols(); // Returns x
    unsigned rows(); // Returns y
    unsigned dies(); // Returns z
    void operator=(grid_t<T> &g); // Assignment
    T& operator[](unsigned i) // Indexing operator
    T& operator()(unsigned Columns, unsigned Rows, unsigned Dies);
    void reserve(unsigned Columns, unsigned Rows, unsigned Dies);
    void clear(); // Reset function
    Meter grid_cell_width, grid_cell_height; // Grid cell geometry
private:
    unsigned x, y, z; // x, y, z dimensional size
};
```

5.3 Counter

Access counters (KITFOX_DATA_COUNTER) represent microarchitectural activities of a component. The dynamic power is calculated based on the counter numbers. switching counts represent logical switching activities of the component such as combinational logic, functional unit, wire, etc. read and write counts mean typical and read/write behaviors of storage units including cache, memory, array, buffer, queue, etc. For caches, read and write counts implicitly include the accesses to tag arrays, so tag access counts must not be counted in addition read/write counters. read_tag and write_tag are used to count the read/write accesses to tag arrays. Note that both read and write cache misses are counted by read_tag, and write_tag is not the counter for write misses but used to represent tag write/update behaviors such as the tag update of reservation station. In fact, a cache miss incurs multiple increments of counters. First, it reads the tag array (read_tag++) and finds a miss. MSHR is a separate unit than the main cache array, so the MSHR has to be modeled with another pseudo component. The MSHR counters must not be mixed with the cache counters. This is the same for other scheduling buffers in the cache including prefetch and writeback buffers. When the cache miss is served from the next-level cache or memory, the cache array is updated (write++). For a fully-associative cache, a cache access requires the search of multiple entries to determine hit or miss, which is different from the access behavior or typical set-associative cache. To account for this case, search counter is used to represent the entry-searching behavior of the cache array. In this situation, read, write, or tag accesses indicate the direct access to a cache entry, not including the search behavior.

```
class counter_t {
public:
    counter_t();

    Count switching; // Logical switching activities (e.g., logics, wires, ALUs, etc.)
    Count read; // Read accesses of a storage unit (e.g., caches, arrays, queues, etc.)
    Count write; // Write accesses of a storage unit
    Count search; // Tag search of a CAM structure (fully-associative cache).
    Count read_tag; // Tag read accesses (e.g., cache miss)
    Count write_tag; // Tag write accesses (e.g., tag update)
    /* Additional counters for DRAM */
    Count precharge; // Precharge counts
    Count background_open_page_high;
    Count background_open_page_low;
    Count background_closed_page_high;
    Count background_closed_page_low;

    void operator=(const counter_t &c);
    const counter_t operator+(const counter_t &c);
    const counter_t operator-(const counter_t &c);
    const counter_t operator*(const Count &c);
    const counter_t operator/(const Count &c);
    void clear(); // Reset function
```

```
};
```

5.4 Unit Energy

Unit energy (KITFOX_DATA_UNIT_ENERGY) defines the per-access energies of distinct access behaviors defined in the counter_t class.

```
class unit_energy_t {
public:
    unit_energy_t();

    Joule baseline; // Activity-independent dynamic energy consumption per clock cycle
    Joule switching;
    Joule read;
    Joule write;
    Joule read_tag;
    Joule write_tag;
    Joule search;
    Joule precharge;
    Joule background_open_page_high;
    Joule background_open_page_low;
    Joule background_closed_page_high;
    Joule background_closed_page_low;
    Joule leakage; // Leakage energy dissipation per clock cycle

    void operator=(const unit_energy_t &e);
    const unit_energy_t operator+(const unit_energy_t &e);
    const unit_energy_t operator-(const unit_energy_t &e);
    void clear(); // Reset function
};
```

5.5 Energy

Energy class (KITFPX_DATA_ENERGY) represents the generic energy definition comprised of dynamic and leakage portions.

```
class energy_t {
public:
    energy_t();

    Joule total;
    Joule dynamic;
    Joule leakage;

    Joule get_total(); // Returns total (if non-zero) or dynamic+leakage
    void operator(const energy_t &e);
    const energy_t operator+(const energy_t &e);
    const energy_t operator-(const energy_t &e);
    void clear(); // Reset function
};
```

5.6 Power

Power (KITFOX_DATA_POWER or KITFOX_DATA_TDP) represent generic power (or TDP) definition comprised of dynamic and leakage portions.

```
class power_t {
public:
    power_t();
```

```
    Watt total;
    Watt dynamic;
    Watt leakage;

    Watt get_total(); // Returns total (if non-zero) or dynamic+leakage
    void operator(const power_t &e);
    const power_t operator+(const power_t &e);
    const power_t operator-(const power_t &e);
    void clear(); // Reset function
};
```

5.7 Data Conversion

Several operators are defined in ei-defs.h to simplify the common data conversions, such as energy from/to power.

```
inline power_t operator*(const energy_t &e, const Hertz &f); // Power=Energy*Freq
inline power_t operator/(const energy_t &e, const Second &t); // Power=Energy/Time
inline energy_t operator/(const power_t &p, const Hertz &f); // Energy=Power/Freq
inline energy_t operator*(const power_t &p, const Second &t); // Energy=Power*Time
```

6 List of API Functions

This section lists the API functions of KitFox. For the examples of using these API functions, refer to the later sections of this document.

6.1 KitFox Constructor and Configuration Functions

The KitFox constructor needs one input argument for serial simulation and two arguments for parallel simulation. KitFoxComm is an MPI intra-communicator between KitFox processes in parallel simulations, and InterComm is an MPI inter-communicator between KitFox servers and clients (i.e., user simulator components). Detailed implementation about the MPI communicators will be explained in the later sections of this document (Section 9).

```
kitfox_t(MPI_Comm *KitFoxComm = NULL, MPI_Comm *InterComm = NULL);
```

The following function configures the pseudo component hierarchy, based on the input configuration file and initializes physical models.

```
void configure(const char *ConfigFile);
```

This function finds the pseudo component with ComponentName and returns the integer ID assigned by KitFox.

```
const Comp_ID get_component_id(std::string ComponentName);
```

This function finds the pseudo component with ComponentID and returns the string name (i.e., component path in the input configuration).

```
std::string get_component_name(Comp_ID ComponentID);
```

6.2 Computational Functions

This function triggers the power calculation of the energy library model associated with a pseudo component, based on the access counters. Calculated power is stored in the queue of the pseudo component.

```
void calculate_power(Comp_ID ComponentID, Second Time, Second Period, counter_t Counter);
```

This function triggers the temperature calculation of the pseudo component. Before calculating temperature, this function internally synchronizes the power data across pseudo components. It recursively aggregates the power numbers from the leaf components up to the pseudo component that is associated with the thermal library. After power data synchronization, the pseudo components that are designated as the floorplans of the thermal library store the up-to-date power data. Then, the temperature calculation function of the thermal library is invoked. After calculating the thermal field, the floorplan components are updated with the new temperature values stored in the data queues. When inserting the thermal data, it also synchronizes the pseudo components by recursively inserting the temperature value of the floorplan to the data queue of every descendent component. This assumes that no placement information is necessarily provided with the descendant components of the floorplan, and hence uniform thermal field is assumed within the floorplan. If descendent components are associated with energy library models, inserting new temperature data invokes the library-update functions that handle leakage-feedback interactions.

```
void calculate_temperature(Comp_ID ComponentID, Second Time, Second Period);
```

This function triggers the calculation of failure rate λt. Calculated failure rates are stored in the data queue of pseudo components. Temperature, voltage, and clock frequency data of the pseudo component calculating the failure rate must be up to date before calling this function. Otherwise, the simulation terminates by detecting the synchronization error.

```
void calculate_failure_rate(Comp_ID ComponentID, Second Time, Second Period);
```

6.3 Data Queue Functions

This function inserts the data into the queue of pseudo components. Return value is an error_code that must be equal to KITFOX_QUEUE_ERROR_NONE if the operation is successful.

```
template <typename T>
const int push_data(Comp_ID ComponentID, Second Time, Second Period, int DataType, T *Data);

template <typename T>
const int push_data(Comp_ID ComponentID, Second Time, int DataType, T *Data);
```

This function retrieves the data from the queue of pseudo components. Return value is also an error_code that must be equal to KITFOX_QUEUE_ERROR_NONE if the operation is successful.

```
template <typename T>
const int pull_data(Comp_ID ComponentID, Second Time, Second Period, int DataType, T *Data);

template <typename T>
const int pull_data(Comp_ID ComponentID, Second Time, int DataType, T *Data);
```

This function overwrites the data in the queue of pseudo component. Return value is an error_code that must be equal to KITFOX_QUEUE_ERROR_NONE if the operation is successful.

```
template <typename T>
const int overwrite_data(Comp_ID ComponentID, Second Time, Second Period, int DataType, T *Data);

template <typename T>
const int overwrite_data(Comp_ID ComponentID, Second Time, int DataType, T *Data);
```

These functions synchronize the queue data of pseudo components. An error_code is returned, which has to be KITFOX_QUEUE_ERROR_NONE if the operation is successful.

```
const in synchronize_data(Comp_ID ComponentID, Second Time, Second Period, int DataType);

template <typename T>
const int synchronize_and_pull_data(Comp_ID ComponentID, Second Time, Second Period,\
                                    int DataType, T *Data);

template <typename T>
const int push_and_synchronize_data(Comp_ID ComponentID, Second Time, Second Period,\
                                    int DataType, T *Data);

template <typename T>
const int overwrite_and_synchronize_data(Comp_ID ComponentID, Second Time, Second Period,\
                                         int DataType, T *Data);
```

7 Data Queues

7.1 Data Queue Types

Data queues are used to store measurement data from multicore timing simulation and associated physical models. Organization and operation of these data queues are central to correct modeling of multi-physics interactions. In KitFox, computed results from each physical model are stored in *data queues* to handle cross-reference between libraries and runtime update of models. Each data queue stores a single type of data (e.g., power) and is identifiable with type information that indicates which type of data is stored within. Each element in the queue is time-stamped with 1) simulation time at which it was created and 2) sampling interval at which it was recorded. Since the computed results are stored in separate structures rather than overwriting the variables of models, user or other libraries can refer to the correct results with time times while runtime updates of the models can independently proceed without corrupting the results. There are two types of queues as follows.

7.1.1 Closed Queue (queue for periodic data)

Discrete-time data such as power and temperature are calculated and stored at the end of sampling interval, time $= t$ based on observed statistics during period p. These data are regarded as valid during $(tp, t]$ duration, and the queue returns the data for any access requests between tp and t (sec) including t (sec).

7.1.2 Open Queue (queue for aperiodic data)

Some types of data are aperiodically collected during runtime simulation. For example, clock frequency remains constant until a control mechanism (e.g., dynamic frequency scaling) decides to change the clock frequency. In such a case, datum stored at time $= t$ is valid for $[t, \infty)$ or until a new value is inserted at $t + \delta$ (sec). The queue returns the data for access requests between t and $t + \delta$ (or ∞) including t (sec).

7.2 Creating Data Queues

When pseudo components are created, several queues are created by default. The followings are the list of default data queues created with pseudo components.

- Close queues (KITFOX_QUEUE_DISCRETE):
  power (KITFOX_DATA_POWER), temperature (KITFOX_DATA_TEMPERATURE)
- Open queues (KITFOX_QUEUE_OPEN):
  TDP(KITFOX_DATA_TDP), voltage (KITFOX_DATA_VOLTAGE), dimension (KITFOX_DATA_DIMENSION),
  threshold voltage (KITFOX_DATA_THRESHOLD_VOLTAGE),
  clock frequency (KITFOX_DATA_CLOCK_FREQUENCY)

Depending on the library type, additional default queues are created in the pseudo components.

- Power models: counters (KITFOX_DATA_COUNTERS)
- Thermal models: thermal grid (KITFOX_DATA_THERMAL_GRID)
- Reliability models: failure rate (KITFOX_DATA_FAILURE_RATE)

Data queues can also be explicitly created by defining the options in the input configuration file. If the queue is already created by default, no duplicated queues will be created.

```
component: {
    source: { // Component  source
        data_queue = ["KITFOX_DATA_TEMPERATURE", "KITFOX_DATA_POWER", "KITFOX_DATA_COUNTER"];
        queue_size = 2; // All data queues have 2 entries.
        library: { // Physical model parameters are defined here. };
    };
};
```

Or alternatively, the queue size can be independently defined.

```
component: {
    source: { // Component  source
        data_queue: {
            KITFOX_DATA_TEMPERATURE: { size = 2; }; // 2-entry temperature queue
            KITFOX_DATA_POWER: { size = 2; }; // 2-entry power queue
            KITFOX_DATA_COUNTER: { size = 1 }; // 1-entry counter queue
        };
        library: { // Physical model parameters are defined here. };
    };
};
```

7.3 Accessing Data Queues

There are three queue operations defined: push_data (data insertion), pull_data (data collection), and overwrite_data (data replacement). push_data and pull_data are basic write and read operations of the data queue, respectively. pull operation is not a dequeue operation, and dequeuing is implicitly handled when the queue becomes full. overwrite_data function replaces the existing entry with the new value. All these queue operations require data type (e.g., KITFOX_DATA_POWER) and time tag information (i.e.,g time t and period p). A queue operation first finds the data queue with the data type identifier, where each data queue stores a single type of data. Time tag is checked at every queue operation, and error detection is provided. For push_data operation, data interval must be contiguous. If the period p is not provided (i.e., p = UNSPECIFIED_TIME), the queue operation implicitly derives the period value by comparing with the last entry in the queue. For pull_data operation, a data request must match the time tag (t and p pair) of an existing entry. If no matching entry is found, an error code is set. If the period p is not provided, it tries to find the interval that the time t falls into and returns the value of that interval. overwrite_data is combined operation of pull-and-push. It first performs the same process as the pull operation and then replaces the data value if a matching entry is found. The following lists the queue operations functions provided with KitFox.

```
template <typename T> const int push_data(Comp_ID ComponentID, Second Time,\
                      Second Period, int DataType, T *Data);
template <typename T> const int pull_data(Comp_ID ComponentID, Second Time,\
                      Second Period, int DataType, T *Data);
template <typename T> const int overwrite_data(Comp_ID ComponentID, Second Time,\
                      Second Period, int DataType, T *Data);
```

The example below shows how to insert the power data using the push function and retrieving the value via the pull operation.

```
/* Pseudo component ID */
// Get the component ID of a pseudo component package.cores.core0
Comp_ID core0_id = kitfox->get_component_id("package.cores.core0");

/* Timing information */
Second current_time = 120.0; // 120 seconds
Second sampling_period = 10.0 // 10-second interval

/* Data (power for example) */
power_t core0_power;
core0_power.dynamic = 23.8; // 23.8W dynamic power
core0_power.leakage = 5.7; // 6.2W leakage power

/* Push operation */
// Insert the data into the queue of pseudo component package.cores.core0.
int error_code = kitfox->push_data(core0_id, current_time, sampling_period,\
                         KITFOX_DATA_POWER, &core0_power);
assert(error_code == KITFOX_QUEUE_ERROR_NONE); // Check queue operation status.

/* Pull operation */
```

```
// Acquire the data from the queue of pseudo component package.cores.core0.
// Find the matching entry with the given time t since period p is undefined.
int error_code = kitfox->pull_data(core0_id, current_time, UNSPECIFIED_TIME,\
                        KITFOX_DATA_POWER, &core0_power);
assert(error_code == KITFOX_QUEUE_ERROR_NONE); // Check queue operation status.

std::cout << "core0.power = " << core0_power.get_total() << std::endl;
```

When there is an error with queue operation, a non-zero error code is returned. The following lists the error codes of the data queue. overwrite_data is a combined operation of push and pull, and it produces one of the error codes that push or pull may generate.

- KITFOX_QUEUE_ERROR_NONE: Successful queue operation
- KITFOX_QUEUE_ERROR_TIME_DISCONTINUOUS: push_data with a gap between data intervals
- KITFOX_QUEUE_ERROR_OUTORDER: push_data with older time tag than existing entries
- KITFOX_QUEUE_ERROR_OVERLAP: push_data with overlapping data intervals
- KITFOX_QUEUE_ERROR_INVALID: pull_data with no matching time tag
- KITFOX_QUEUE_ERROR_DATA_DUPLICATED: Duplicated data queue is created; the second queue creation is ignored.
- KITFOX_QUEUE_ERROR_DATA_INVALID: No data queue exists for given data type.
- KITFOX_QUEUE_ERROR_DATA_TYPE_INVALID: Data queue exists, but data type does not match.

7.4 Synchronizing Data Queues between Pseudo Components

The data queues of pseudo components can be synchronized via synchronize_data function. When this function is called at a pseudo component, data are coalesced or distributed, depending on the data type, across the pseudo component hierarchy between the given pseudo component and its descent components. synchronize_data function is recursively called inside KitFox until the pseudo component and all of its descendent components are synchronized.

```
const int synchronize_data(Comp_ID ComponentID, Second Time, Second Period, int DataType);
```

The following describes the synchronization method of different data types.

- Counter (KITFOX_DATA_COUNTER): counter_t values are added from leaf components to the component of synchronization.
- Power (KITFOX_DATA_POWER or KITFOX_DATA_TDP): power_t values are aggregated from leaf components to the component of synchronization.
- Dimension (KITFOX_DATA_DIMENSION): area is added up from leaf components to the component of synchronization. Geometry information (i.e., left, bottom, width, height) is not altered.
- Temperature (KITFOX_DATA_TEMPERATURE): Kelvin values are equally applied to sub-components if they do not contain valid temperature values at the given data interval. If a sub-component has a valid temperature, its temperature value is used from that point to its sub-components.
- Voltage (KITFOX_DATA_VOLTAGE or KITFOX_DATA_THRESHOLD_VOLTAGE): Volt values are equally applied to sub-components if they do not contain valid voltage values at the given data interval. If a sub-component has a valid voltage, its voltage value is used from that point to its sub-components.
- Clock frequency (KITFOX_DATA_CLOCK_FREQUENCY): Hertz values are equally applied to sub-components if they do not contain valid clock frequency at the given data interval. If a sub-component has a valid clock frequency, its value is used from that point to its sub-components.
- Other data types: There is no synchronization method defined for other data types.

In the following example, synchronize_data is called for KITFOX_DATA_VOLTAGE at at the pseudo component package.cores.core0. Inside KitFox, all sub-components of package.cores.core0 are updated with the new voltage value by assuming they do not explicitly store different different voltage values at the given data interval.

```
Volt new_voltage = 1.28; // 1.28V voltage scaling is to be applied.
Second current_time = 210.0; // Current time is 210 seconds.

/* Push operation for new voltage */
// Voltage is stored in an open queue, where period p is unknown.
int error_code = kitfox->push_data(core0_id, current_time, UNSPECIFIED_TIME,\
                                   KITFOX_DATA_VOLTAGE, &new_voltage);
assert(error_code == KITFOX_QUEUE_ERROR_NONE);

/* Synchronize all sub-components of package.cores.core0  */
int error_code = kitfox->synchronize_data(core0_id, current_time, UNSPECIFIED_TIME,\
                                          KITFOX_DATA_VOLTAGE);
assert(error_code == KITFOX_QUEUE_ERROR_NONE);
```

Or alternatively, the example above can be performed by a single function.

```
/* Push and synchronize for new voltage */
int error_code = kitfox->push_and_synchronize_data(core0_id, current_time, UNSPECIFIED_TIME,\
                                                   KITFOX_DATA_VOLTAGE, &new_voltage);
assert(error_code == KITFOX_QUEUE_ERROR_NONE);
```

7.5 Data Queues and Library Updates

If a pseudo component is associated with a physical model, update_library_variable function of the physical model is registered with the data queue when KitFox is configured. When new data values are inserted into the queues (i.e., push_data or overwrite_data), queue operations internally call update_library_variable functions. This function defines how to update dependent variables or states in reaction to the new data. Since the library updates in the data queues are internally handled, users do not need to control this situation but should be aware that data insertion into the queue incurs the automated update of the variables of physical models. The advantage of using library updates is that the physical interactions can be simply implemented by transferring data between the queues of pseudo components. The following is the internal implementation of data queue insertion.

```
template <typename T>
void queue_t::create(const Second Time, const Second Period, const int DataType, T Data) {
    // Error code is set if data queue does not exist.
    data_queue_t<T> *data_queue = get_data_queue<T>(DataType);
    if(data_queue) {
        data_queue->push_back(Time, Period, Data); // Data insertion
        callback(DataType, &Data); // Library update
        error_code = KITFOX_QUEUE_ERROR_NONE; // No error
    }
}
```

8 Calculation of Physical Properties

KitFox provides several API functions to invoke the calculation of physical models.

8.1 Power Calculation

Dynamic energy is calculated with respect to access counters that have to be collected from microarchitecture simulators (refer to Section 5.3 for counter definition). Per-access energies of distinct access types are estimated by the physical models and multiplied with the counters of corresponding access types (refer to Section 5.4 for unit energy definition). The calculated power result is stored in the data queue of the pseudo component that can be probed by a queue operation functions pull_data.

```
void calculate_power(Comp_ID ComponentID, Second Time, Second Period, counter_t Counter);
```

This function can be called with the pseudo components linked to one of the following physical models.

- DRAMSim2
- DSENT
- IntSim
- McPAT

calculate_power function performs the following operations.

```
energy.dynamic = unit_energy.baseline * period * clock_frequency;
energy.dynamic += unit_energy.switching * counter.switching;
energy.dynamic += unit_energy.read * counter.read;
energy.dynamic += unit_energy.write * counter.write;
energy.dynamic += unit_energy.read_tag * counter.read_tag;
energy.dynamic += unit_energy.write_tag * counter.write_tag;
energy.dynamic += unit_energy.search * counter.search;
energy.dynamic += unit_energy.precharge * counter.precharge;
energy.dynamic += unit_energy.background_open_page_high * counter.background_open_page_high;
energy.dynamic += unit_energy.background_open_page_low * counter.background_open_page_low;
energy.dynamic += unit_energy.background_closed_page_high * counter.background_closed_page_high;
energy.dynamic += unit_energy.background_closed_page_low * counter.background_closed_page_low;
energy.leakage = unit_energy.leakage * period * clock_frequency;
energy.total = energy.dynamic + energy.leakage;
power = energy / period;
```

8.2 Temperature Calculation

Temperature is calculated based on the power distribution of thermal blocks (i.e., floorplans, thermal grid cells) on the die. This function first synchronizes the power data of pseudo components. When DoPowerSync = false, KitFox assumes that the power data are already synchronized and skips this process. If pseudo components have asynchronous power data, the simulation terminates with an error message. After calculating the temperature, pseudo components that are designated as floorplans have updated temperature values in the data queues. Temperature data are then synchronized across pseudo components. It is assumed that the temperature is uniform within the floorplan by assuming that no further placement information is available. All the descendent components of the floorplan are updated with the same temperature.

```
void calculate_temperature(Comp_ID ComponentID, Second Time, Second Period, bool DoPowerSync);
```

This function can be called at pseudo components associated with one of the following physical models.

- 3D-ICE
- HotSpot
- Microfluidics

8.3 Failure Rate (Reliability) Calculation

Failure rates are calculated based on the temperature, voltage, and clock frequency of modeled blocks. These variables must be updated before calling the reliability calculation function. When DoDataSync = false, KitFox skips the synchronization process. Note that temperature (KITFOX_DATA_TEMPERATURE) is stored in the closed queue that requires data insertion at every sampling point, which means that calculate_temperature function must be called prior to this function. On the other hand, voltage (KITFOX_DATA_VOLTAGE) and clock frequency(KITFOX_DATA_CLOCK_FREQUENCY) are stored in open queues that return the data for the requests between $[t, \infty)$, where t is the time that the last values were inserted. If voltage and frequency are not updated before calling this function but are supposed to be scaled, then this function can inadvertently calculate the failure rate based on the old voltage and frequency values. However, it is rare for this case to occur since most control decision (e.g., DVFS) are made after physical calculations are all completed. The following function is called at the pseudo component associated with the failure physical model.

```
void calculate_failure_rate(Comp_ID ComponentID, Second Time, Second Period, bool DoDataSync);
```

When this function is invoked, what is actually stored in the data queue is λt value, where $\lambda t = \sum_{i=1}^{N} \lambda_i (t_i - t_{i-1})$ based on the exponential distribution model. i is the index of N sampling points; t_N is the current time, and $t_N - t_{N-1}$ is the last sampling period. λ_i is the transient failure rate of i^{th} sampling point. The mean-time-to-failure (MTTF) can be calculated as $\frac{t}{\lambda t}$ indicating the expected lifetime based on the cumulative calculation of transient failure rates.

8.4 Calculation Sequence

Processor physical properties are dynamically interacting with each other, creating dependencies as illustrated in Figure 2. It is critical to follow the exact sequence of calculations in order to correctly model the physical interactions chain as well as not to break the simulations.

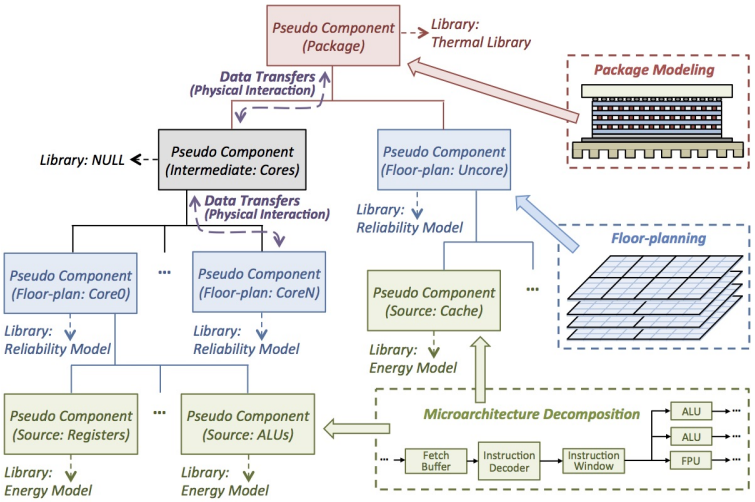

Figure 4. A processor is represented as the *pseudo component hierarchy* in KitFox framework. Pseudo components are physically defined units where associated libraries estimate physical phenomena. Physical interactions are emulated by transferring data between the data queues of pseudo components.

The following shows an example of stepwise function calls to perform the physics chain shown in Figure 2.

```
while (program runs) {
    clock_cycle++;
    Do {architecture timing simulation and counters collection}

    if (@sampling point) {
        Second time = get_current_time();
        Second period = get_sampling_interval();
```

```
        /* Calculate power for all source components with power models */
        for (all source components with power models) {
            kitfox->calculate_power(src_comp_id, time, period, src_comp_counters);
        }

        /* Calculate temperature at the component with thermal model.
        Power-temperature dependency is internally captured here. */
        kitfox->calculate_temperature(package_cmp_id, time, period, /*DoPowerSync*/true);

        /* Calculate failure rate for all components with reliability models.
        Temperature-voltage-frequency-reliability dependency is captured here. */
        for (all components (e.g., floor-plans) with reliability models) {
            kitfox->calculate_failure_rate(flp_comp_id, time, period, /*DoDataSync*/true);
        }

        /* Probe any components of interest and retrieve the data */
        Kelvin core0_temperature;
        int error_code = kitfox->pull_data(core0_id, time, period,\
                         KITFOX_DATA_TEMPERATURE, &core0_temperature);
        assert(error_code == KITFOX_QUEUE_ERROR_NONE);

        /* Apply execution control (e.g., frequency scaling) */
        if(core0_temperature > thermal_threshold) {
            Hertz P4_state_freq = 1.0e9; // 1GHz
            error_code = kitfox->push_and_synchronize_data(core0_id, time, period,\
                         KITFO_DATA_CLOCK_FREQUENCY, &P4_state_freq);
            assert(error_code == KITFOX_QUEUE_ERROR_NONE);
        }
    }
}
```

9 Examples

9.1 Serial Simulation

The following example shows initializing the microarchitecture simulator and connecting the processor model to KitFox. When running the microarchitecture timing simulation, stepwise KitFox functions can be called as shown in Section 8.4.

```
#include <libconfig.h++>
#include "kitfox.h"

using namespace libKitFox;
using namespace libconfig;
using namespace std;

int main(int argc, char** argv) {
    /* Create and configure KitFox */
    kitfox_t *kitfox = new kitfox_t();
    kitfox->configure("input.config");

    /* Create user microarchitecture simulator */
    user_processor_t *processor = new processor("processor.config");

    /* Connect the processor to kitfox by passing the pointer. */
    processor->connect_kitfox(kitfox);

    /* Run processor simulation: Within the run function, microarchitecture
    simulation is performed, and stepwise KitFox functions are called at
    user-defined sampling intervals. */
    processor->run();

    delete kitfox;
    delete processor;

    return 0;
}
```

9.2 Parallel Simulation

In parallel simulations, microarchitecture and KitFox simulations can be split into multiple processes. KitFox itself can also be split into different processes. In this case, each KitFox creates only the pseudo components that are to be modeled in its process, and the KitFox objects communicate across processes via the MPI interface to transfer the data between pseudo components. Instead of passing the pointer to KitFox as shown in the serial simulation example, the processor model has to create an KitFox client to delegate the messages to the KitFox object in different process. The following shows an example of creating and initializing parallel KitFox via the MPI interface.

```
#include <mpi.h>
#include <libconfig.h++>
#include "kitfox.h"

using namespace libKitFox;
using namespace libconfig;
using namespace std;

int main(int argc, char **argv) {
    int N_LPs; // Number of processes
    int Rank; // Rank of this process
    int N_ArchLPs; // Number of architecture simulation processes
    assert(N_ArchLPs < N_LPs);
```

```
    /* Get MPI information. */
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &NPs);
    MPI_Comm_rank(MPI_COMM_WORLD, &Rank);

    /* Split MPI communicator to architecture simulators and KitFox servers. */
    MPI_Comm LOCAL_COMM, INTER_COMM;
    MPI_Comm_split(MPI_COMM_WORLD, IsArchLP(), Rank, &LOCAL_COMM);
    MPI_Intercomm_create(LOCAL_COMM, 0, MPI_COMM_WORLD,
                         IsArchLP()?N_ArchLPs:0, 1234, &INTER_COMM);

    /* Get the MPI information within the intra-communicator. */
    int N_LocalNPs; // Number of processes within the intra-communicator
    int LocalRank; // Rank of this process within the intra-communicator
    MPI_Comm_size(LOCAL_COMM, &N_LocalNPs);
    MPI_Comm_rank(LOCAL_COMM, &LocalRank);

    if(!IsArchLP() {
        /* Create and initialize the KitFox of this LP */
        kitfox_t *kitfox = new kitfox_t(/*Intra-communicator*/ &LOCAL_COMM,
                                        /*Inter-communicator*/ &INTER_COMM);

        /* Cross-connect KitFox processes. This function is blocking until all KitFox
        processes are created and connected. */
        kitfox->connect();

        /* Initialize the KitFox of this process. */
        kitfox->configure(argv[LocalRank]);
    }
    else {
        /* Create user architecture simulator */
        user_processor_t *processor = new processor("processor.config", &LOCAL_COMM);

        /* Create KitFox client and connect to the server */
        int ConnectinKitFoxRank = get_which_KitFox_process_to_connect();
        processor->connect_kitfox(&INTER_COMM, ConnectingKitFoxRank);

        /* Run processor simulation: Within the run function, architecture simulation
        is performed, and stepwise KitFox functions are called via KitFox clients. */
        processor->run();
    }
}
```

The following shows using the KitFox client within the user processor simulator.

```
#include "kitfox-defs.h"
#include "kitfox-client.h"

void user_processor_t::connect_kitfox(MPI_Comm *InterComm, const int ConnectingKitFoxRank) {
    /* Create KitFox Client. InterComm is MPI inter-communicator, and
    ConnectingKitFoxRank is the rank of the KitFox (within the KitFox intra-communicator)
    that this simulator wants to connect.
    kitfox_client = new kitfox_client_t(InterComm, ConnectingKitFoxRank);

    /* Connect to KitFox in different process. This function is blocking until
    KitFox finishes initialization and accepts the connection. */
    kitfox_client->connect_server(ConnectingKitFoxRank);
}
```

```
/* Once connected, the KitFox client provides the same set of functions as KitFox. */
void user_processor_t::run() {
    while (program runs) {
        clock_cycle++;
        Do {architecture timing simulation and counters collection}

        if (@sampling point) {
            Second time = get_current_time();
            Second period = get_sampling_interval();

            /* Calculate power for all source components with power models */
            for (all source components with power models) {
                /* Functions without return values are non-blocking. */
                kitfox_client->calculate_power(src_comp_id, time, period, src_comp_counters);
            }

            /* Calculate temperature at the component with thermal model.
            Power-temperature dependency is internally captured here. */
            kitfox_client->calculate_temperature(package_cmp_id, time, period, true);

            /* Calculate failure rate for all components with reliability models.
            Temperature-voltage-frequency-reliability dependency is captured here. */
            for (all components (e.g., floor-plans) with reliability models) {
                kitfox_client->calculate_failure_rate(flp_comp_id, time, period, true);
            }

            /* Probe any components of interest and retrieve the data.
            Functions with return values are blocking. */
            Kelvin core0_temperature;
            int error_code = kitfox_client->pull_data(core0_id, time, period,\
                             KITFOX_DATA_TEMPERATURE, &core0_temperature);
            assert(error_code == KITFOX_QUEUE_ERROR_NONE);

            /* Apply execution control (e.g., frequency scaling) */
            if(core0_temperature > thermal_threshold) {
                Hertz P4_state_freq = 1.0e9; // 1GHz
                error_code = kitfox_client->push_and_synchronize_data(core0_id, time, period,\
                             KITFOX_DATA_CLOCK_FREQUENCY, &P4_state_freq);
                assert(error_code == KITFOX_QUEUE_ERROR_NONE);
            }
        }
    }
}
```

10 Integration of Models (for Advanced Users)

This section is intended for advanced users who plan to integrate new models into KitFox framework.

10.1 Library

All models integrated in KitFox are categorized into classes called *library*. model_library_t is the base class for all physical models, which is defined as follows. The constructor take the pointer to the pseudo component and model library type (KITFOX_LIBRARY_ENERGY_MODEL, KITFOX_LIBRARY_THERMAL_MODEL, or KITFOX_LIBRARY_RELIABILITY_MODEL). update_library_variable is the runtime variable update function that is registered as a callback function of the data queue (refer to Section 7).

```
class model_library_t {
public:
    model_library_t(pseudo_component_t *PseudoComponent, int Type);
    virtual ~model_library_t();

    int type; // Library type

    /* Callback function associated with the data queue to update runtime variables
    and states of subclass model */
    virtual bool update_library_variable(int Type, void *Value,
                                         bool IsLibraryVariable = false) = 0;

    /* Initialization function to be defined by subclass model */
    virtual void initialize() = 0;

protected:
    pseudo_component_t *pseudo_component; // Pointer to pseudo component
};
```

There are three different derivative classes defined; *energy*, *thermal*, *reliability* libraries.

10.1.1 Energy Library

Energy library is one of the subclasses of model_library_t and the base class for energy/power models. Possible models (Type) are KITFOX_LIBRARY_DRAMSIM, KITFOX_LIBRARY_DSENT, KITFOX_LIBRARY_INTSIM, and KITFOX_LIBRARY_MCPAT. get_unit_energy function returns the unit_energy class that includes per-access energies of different access types (refer to Section 5). get_runtime_power is called inside calculate_power function of KitFox (refer to Section 8) and returns the runtime power including leakage. If the model support area estimation, get_area can be used, otherwise a trivial value is returned. get_tdp_power calculates and returns the maximum power based on the assumption of peak access counts and temperature.

```
class energy_library_t : public model_library_t {
public:
    energy_library_t(pseudo_component_t *PseudoComponent, int Type);
    virtual ~energy_library_t();

    int type; // Model type

    /* Returns per-access energies of different access types. */
    virtual unit_energy_t get_unit_energy() = 0;

    /* Calculates TDP power w.r.t (user defined) max duty cycle
    and peak temperature. */
    virtual power_t get_tdp_power(Kelvin MaxTemperature = 373.0) = 0;

    /* Calculates runtime power with respect to counters. */
    virtual power_t get_runtime_power(Second Time, Second Period, counter_t Counter) = 0;
```

```
    /* Calculates area (if possible) */
    virtual MeterSquare get_area(void) = 0;
};
```

10.1.2 Thermal Library

Thermal library is one of the subclasses of model_library_t and the base class for thermal models. Possible models (Type) are KITFOX_LIBRARY_3DICE, KITFOX_LIBRARY_HOTSPOT, and KITFOX_LIBRARY_MICROFLUIDICS. push_floorplan_power function maps the power input to the floorplan power structure of the thermal model, and calculate_temperature can only be called after push_floorplan_power is called for all floorplans to update the power inputs. This sequence is implemented in the calculate_temperature of KitFox, so the thermal library only has to implement the details of power mapping and temperature calculation. Note that calculate_temperature does not have a return value. After this function is called, KitFox calls get_thermal_grid, get_floorplan_temperature, or get_point_temperature to update the data queues of pseudo components, so calculate_temperature only has to update the internal thermal states and be ready for subsequent data queries. Other floorplan mapping functions provide useful information about the floorplans.

```
class thermal_library_t : public model_library_t {
public:
    thermal_library_t(pseudo_component_t *PseudoComponent, int Type);
    virtual ~thermal_library_t();

    int Type; // Model type

    /* Calculates transient or steady-state temperature. */
    virtual void calculate_temperature(Second Time, Second Period) = 0;

    /* Returns 3D thermal grid stack of source layers. */
    virtual grid_t<Kelvin> get_thermal_grid(void) = 0;

    /* Returns representative floorplan temperature. */
    virtual Kelvin get_floorplan_temperature(Comp_ID ComponentID,
                    int Type = KITFOX_TEMPERATURE_MAPPING_UNKNOWN) = 0;

    /* Returns point temperature on the die. */
    virtual Kelvin get_point_temperature(Meter X, Meter Y, Index Layer) = 0;

    /* Maps the power to the internal floor-plan structure or array. */
    virtual void push_floorplan_power(Comp_ID ComponentID, power_t PartitionPower) = 0;

    /* Other floorplan mapping functions */
    void add_pseudo_floorplan(std::string ComponentName, Comp_ID ComponentID);
    const Index get_pseudo_floorplan_counts(void);
    const Comp_ID get_pseudo_floorplan_id(std::string ComponentName);
    const char* get_pseudo_floorplan_name(Comp_ID ComponentID);
    const Comp_ID get_pseudo_floorplan_id(unsigned i);
};
```

10.1.3 Reliability Library

Thermal library is one of the subclasses of model_library_t and the base class for reliability models. Possible model (Type) is KITFOX_LIBRARY_FAILURE. get_failure_rate calculates the instantaneous failure rate λ for given temperature, voltage, frequency, and duration (i.e., period). Cumulative calculation of the failure rate (refer to Section 8) is handled in KitFox, and this function only needs to calculate the failure rate without considering the failure rate history.

```
class reliability_library_t : public model_library_t {
public:
    reliability_library_t(pseudo_component_t *PseudoComponent, int Type);
```

```
    virtual ~reliability_library_t();

    int Type; // Model type

    virtual Unitless get_failure_rate(Second Time, Second Period, Kelvin Temperature,\
    Volt Vdd, Hertz Frequency) = 0;
};
```

10.2 Wrapper Class of Integrated Models

For each model being integrated into the library, a wrapper class is created to define the details of virtual functions of base classes. For instance, HotSpot is a thermal model and takes thermal_library_t as the base class that again takes the base class model_library_t. The following shows an example of HotSpot wrapper class.

```
#include "library.h"

class thermallib_hotspot : public thermal_library_t {
    thermallib_hotspot(pseudo_component_t *PseudoComponent);
    ~thermallib_hotspot();

    virtual void initialize() {
        /* Define the initialization of HotSpot variables */
    }

    virtual void calculate_temperature(Second Time, Second Period) {
        /* Define the temperature calculation method of HotSpot */
    }

    virtual grid_t<Kelvin> get_thermal_grid() {
        /* Returns the 3D (or 2D) thermal grid for only source layers */
    }

    virtual Kelvin get_point_temperature(Meter X, Meter Y, Index Layer) {
        /* Returns the temperature of the given point on the die */
    }

    virtual void push_floorplan_temperature(Comp_ID ComponentID, power_t PartitionPower) {
        /* Map the power input to the power structure of floor-plans. If the thermal model
        uses string name to identify the floor-plan, the base class provides
        get_pseudo_floorplan_name to retrieve the floor-plan name based on ComponentID.
        This name-ID mapping is created in KitFox, so the thermal library can readily use it.*/
        std::string floorplan_name = get_floorplan_name(ComponentID);
    }

    virtual void update_library_variable(int Type, void *Value, bool IsLibraryVariable) {
        /* This function implements the update of variable. Note that this function is called
        when there is data insertion into the queue of pseudo components. Type denotes
        data type. If data type is not the KitFox-defined data such as KITFOX_DATA_TEMPERATURE,
        then IsLibraryVariable must be false.
        */
    }
private:
    /* Other HotSpot variables or structures */
};
```

10.2.1 Parsing Input Configuration

With integrated input configuration, the wrapper class must be aware of parsing the input to initialize the variable of the model in the initialize function. The pseudo_component_t provides methods to read

parameters from the input configuration file. The following is the list of libconfig-related functions that pseudo_component_t provides.

```
libconfig::Setting& get_setting() const;
libconfig::Setting& get_library_setting() const;
const bool exists_library(void);
const bool exists(std::string &Var, bool InPathLookup = false);
const bool exists(const char *Var, bool InPathLookup = false);
const bool exists_in_library(std::string &Var, bool InPathLookup = false);
const bool exists_in_library(const char *Var, bool InPathLookup = false);
libconfig::Setting& lookup(std::string &Var, bool InPathLookup = false);
libconfig::Setting& lookup(const char *Var, bool InPathLookup = false);
libconfig::Setting& lookup_in_library(std::string &Var, bool InPathLookup = false);
libconfig::Setting& lookup_in_library(const char *Var, bool InPathLookup = false);
```

The following shows an example of retrieving input parameters via pseudo_component_t. Note that the base class model_library_t has the pointer to the pseudo component.

```
void thermalllib_hotspot::initialize() {
    /* This function looks up "grid_rows" parameters only in the library of
    this pseudo component. If not defined, it throws an exception error. */
    unsigned grid_rows = pseudo_component->lookup_in_library("grid_rows");

    /* This function does recursive lookup of parameter "voltage" in the library of
    this pseudo component and all of its ancestor components. */
    unsigned voltage = pseudo_component->lookup_in_library("voltage", true);

    /* It can be tested if a parameter is defined in the input configuration to do
    conditional initialization of a variable. */
    double clock_frequency;
    if(pseudo_component->exists_in_library("clock_frequency")) {
        clock_frequency = pseudo_component->lookup_in_library("clock_frequency");
    }
    else {
        clock_frequency = 1e9;
    }

    /* libconfig setting can be returned as well, which provides useful functionality
    for arrays, groups, etc. Refer to libconfig documentation. */
    Setting &libconfig_setting = pseudo_component->lookup_in_library("layers");
    for(unsigned i =0; i < libconfig_setting.getLength(); i++) {
        /* Initialize the parameters of each layer of the package. */
    }
}
```

References

[1] Huang et al., "HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design," *Trans. on VLSI*, Nov. 2006.

[2] Huang et al., "Accurate, Pre-RTL Temperature-Aware Design Using a Parameterized, Geometric Thermal Model," *Trans. on Computers*, Sept. 2008.

[3] Kahng et al., "Orion 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration," *DATE*, Apr. 2009.

[4] Li et al., "McPAT: Integrated Power, Area, Timing Modeling Framework for Multicore Architectures," *MICRO*, Dec. 2009.

[5] Rosenfeld et al., "DRAMsim2: A Cycle Accurate Memory System Simulator," *Computer Architecture Letters*, Jan. 2011.

[6] Sridhar et al., "3D-ICE: Fast Compact Transient Thermal Modeling for 3D ICs with Inter-Tier Liquid Cooling," *ICCAD*, Nov. 2010.

[7] Srinivasan et al., "The Case for Lifetime Reliability-Aware Microprocessors," *ISCA*, June 2004.

[8] Srinivasan et al., "Lifetime Reliability: Toward An Architectural Solution," *IEEE Micro*, May 2005.

[9] Sun et al., "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," *NOCS*, May 2012.

[10] Thoziyoor et al., "A Comprehensive Memory Modeling Tool and Its Application to The Design and Analysis of Future Memory Hierarchies," *ISCA*, June 2008.

[11] White et al., "Microelectronics Reliability: Physics-of-Failure Based Modeling and Lifetime Evaluation," *NASA JPL Publication*, 2008.

A Configuration Examples

This section is mainly comprised of input configuration code. It is intended for users that need examples to find which parameters are necessary for different physical models. Models are listed in the alphabetical order.

A.1 DRAMSim2

Below is an example of configuring DRAMSim2 parameters.

```
component: {
    /* DRAMSim2 (Micron) memory power model */
    memory: {
        library: {
            model = "dramsim";

            /* DDR2 micron 16M 8b x 8 sg3E model */
            IDD0 = 85;
            IDD1 = 100;
            IDD2P = 7;
            IDD2Q = 40;
            IDD2N = 40;
            IDD3Pf = 30;
            IDD3Ps = 10;
            IDD3N = 55;
            IDD4W = 135;
            IDD4R = 135;
            IDD5 = 215;
            IDD6 = 7;
            IDD6L = 5;
            IDD7 = 280;

            BL = 4;
            DEVICE_WIDTH = 8;
            DATA_BUS_BITS = 64;

            tRC = 18;
            tRAS = 14;
            tRFC = 43;

            voltage = 1.8;
            clock_frequency = 0.333e9; // 333MHz

            page_policy = "open_page";

            /* Other DRAMSim2 parameters are for functional simulation,
            not necessarily for power. */
        };
    };
};
```

A.2 DSENT

The following shows an example of configuring DSENT components. In this example, DSENT components are defined as sub-components of the root component package (refer to Section 4.2 for multi-level components and configuration) that defines common variables for different types of DSENT models.

```
component: {
    package: {
        library: {
            model = "none";

            /* Common parameters */
            temperature = 340.0; // 340K initial temperature
            voltage = 0.8; // 0.8V supply voltage
            clock_frequency = 1.0e9; // 1GHz clock frequency

            /* Bulk22LVT - DSENT technology model */
            Gate: {
                PitchContacted = 0.120e-6;
                MinWidth = 0.100e-6;
                CapPerWidth = 0.900e-9;
            };

            Drain: {
                CapPerWidth = 0.620e-9;
            };

            Nmos: {
                CharacterizedTemperature = 300.0;
                EffResWidth = 0.700e-3;
                EffResStackRatio = 0.800;
                OffCurrent = 100e-3;
                MinOffCurrent = 60e-9;
                SubthresholdSwing = 0.100;
                DIBL = 0.150;
                SubthresholdTempSwing = 100.0;
            };

            Pmos: {
                CharacterizedTemperature = 300.0;
                EffResWidth = 0.930e-3;
                EffResStackRatio = 0.680;
                OffCurrent = 100e-3;
                MinOffCurrent = 60e-9;
                SubthresholdSwing = 0.100;
                DIBL = 0.150;
                SubthresholdTempSwing = 100.0;
            };

            Wire: {
                available_layers = ["Metal1",
                                    "Local",
                                    "Intermediate",
                                    "Semiglobal",
                                    "Global"];
                Metal1: {
                    MinWidth = 32e-9;
                    MinSpacing = 32e-9;
                    Resistivity = 5.00e-8;
                    MetalThickness = 60.0e-9;
```

```
        DielectricThickness = 60.0e-9;
        DielectricConstant = 3.00;
    };

    Local: {
        MinWidth = 32e-9;
        MinSpacing = 32e-9;
        Resistivity = 5.00e-8;
        MetalThickness = 60.0e-9;
        DielectricThickness = 60.0e-9;
        DielectricConstant = 3.00;
    };

    Intermediate: {
        MinWidth = 55e-9;
        MinSpacing = 55e-9;
        Resistivity = 4.00e-8;
        MetalThickness = 100.0e-9;
        DielectricThickness = 100.0e-9;
        DielectricConstant = 2.80;
    };

    Semiglobal: {
        MinWidth = 110e-9;
        MinSpacing = 110e-9;
        Resistivity = 2.60e-8;
        MetalThickness = 200.0e-9;
        DielectricThickness = 170.0e-9;
        DielectricConstant = 2.80;
    };

    Global: {
        MinWidth = 160e-9;
        MinSpacing = 160e-9;
        Resistivity = 2.30e-8;
        MetalThickness = 280.0e-9;
        DielectricThickness = 250.0e-9;
        DielectricConstant = 2.60;
    };
};

StdCell: {
    Tracks = 11;
    HeightOverheadFactor = 1.400;
    AvailableSizes = [1.0, 1.4, 2.0, 3.0, 4.0, 6.0, 8.0, 10.0, 12.0, 16.0];
};

/* Photonics */
Waveguide: {
    LossPerMeter = 100.0;
    Pitch = 4.0e-6;
};

Splitter: { Loss = 1.00; }
Coupler: { Loss = 1.00; };

Laser: {
    CW: {
        Efficiency = 0.25;
```

```
        LaserDiodeLoss = 1.0;
        Area = 0.0;
    };
    GatedCW: {
        Efficiency = 0.25;
        LaserDiodeLoss = 1.0;
        Area = 200e-12;
    };
};

Modulator: {
    Ring: {
        SupplyBoostRatio = 1.2;
        ParasiticRes = 100.0;
        ParasiticCap = 5e-15;
        FCPDEffect = 3e-27;
        Tn = 0.01;
        NA = 3e24;
        ND = 1e24;
        ni = 1e16;
        JunctionRatio = 0.8;
        Height = 500e-9;
        Width = 500e-9;
        ConfinementFactor = 0.3;
    };
};

Ring: {
    Area = 100e-12;
    Lambda = 1300e-9;
    GroupIndex = 4;
    Radius = 3e-6;
    ConfinementFactor = 0.3;
    ThroughLoss = 0.01;
    DropLoss = 1.0;
    MaxQualityFactor = 150e3;
    HeatingEfficiency = 100000;
    TuningEfficiency = 10e9;
    LocalVariationSigma = 200e9;
    TemperatureMax = 380;
    TemperatureMin = 280;
    MaxElectricallyTunableFreq = 50e9;
};

Photodetector: {
    Reponsivity = 1.1;
    Area = 10e-12;
    Cap = 0.0;
    ParasiticCap = 5.0e-15;
    Loss = 1.0;
    MinExtinctionRatio = 3.0;
    AvalancheGain = 1.0;
};

SenseAmp: {
    BER = 1e-15;
    CMRR = 5.0;
    OffsetCompensationBits = 5;
    OffsetRatio = 0.04;
```

```
            SupplyNoiseRandRatio = 0.01;
            SupplyNoiseDetRatio = 0.05;
            NoiseMargin = 0.02;
            JitterRatio = 0.01;
        };

        Receiver: {
            Int: { IntegrationTimeRatio = 0.7; };
        };
    };

    component: {
        /* DSENT router model */
        router: { // package.router
            library: {
                model = "dsent";
                energy_model = "Router";
                tech_model = "Bulk22LVT";
                IsPerformTimingOptimization = true;
                TimingOptimization: { StartNetNames = []; };
                TDPInjectionRate = 0.3;
                NumberInputPorts = 5;
                NumberOutputPorts = 5;
                NumberBitsPerFlit = 64;
                NumberVirtualNetworks = 2;
                NumberVirtualChannelsPerVirtualNetwork = [2, 3];
                NumberBuffersPerVirtualChannel = [4, 5];
                InputPort: { BufferModel = "DFFRAM"; };
                CrossbarModel = "MultiplexerCrossbar";
                SwitchAllocator: { ArbiterModel = "MatrixArbiter"; };
                ClockTreeModel = "BroadCastHTree";
                ClockTree: {
                    NumberLevels = 5;
                    WireLayer = "Global";
                    WireWidthMultiplier = 1.0;
                };
            };
        };

        /* DSENT electrical link model */
        electrical_link: { // package.electrical_link
            library: {
                model = "dsent";
                energy_model = "RepeatedLink";
                tech_model = "Bulk22LVT";
                IsPerformTimingOptimization = false;
                TimingOptimization: { StartNetNames = []; };
                TDPInjectionRate = 0.3;
                NumberBits = 64;
                WireLayer = "Global";
                WireWidthMultiplier = 1.0;
                WireSpacingMultiplier = 1.0;
                WireLength = 1e-3;
                Delay = 1e-9;
            };
        };
    };
};
```

A.3 IntSim

Below is a configuration example to use IntSim. IntSim characterizes the power at the chip-level based on transistor density, wire length, etc, which means that this power model is not characterized with respect to microarchitectural configurations.

```
component: {
    /* IntSim chip-level power model */
    chip: {
        library: {
            model = "intsim";
            energy_model = "default";

            feature_size = 22e-9; // 22nm technology
            clock_frequency = 2e9; // 2GHz
            voltage = 0.8; // 0.8V Vdd
            temperature = 358.0; // Avg temperature
            threshold_voltage = 0.14;

            area = 35e-6; // 35mm^2
            num_gates = 58e6; // 58M transistors
            critical_path_depth = 10;
            rents_constant_k = 4;
            rents_constant_p = 0.6;
            activity_factor = 0.1; // Avg switching activity factor
            tdp_activity_factor = 0.2;
            Vdd_spec = 0.8;
            Vt_spec = 0.14;
            oxide_thickness = 1.1e-9;
            alpha_power_law = 1.3;
            pn_ratio = 0.5;
            subvtslope_spec = 100e-3;
            dielectric_permitivity = 2.0;
            copper_resistivity = 1.95e-8;
            wiring_aspect_ratio = 2.0;
            reflectivity_coefficient = 0.5;
            specularity = 0.5;
            num_power_pads = 600;
            power_pad_distance = 300e-6;
            power_pad_length = 50e-6;
            ir_drop_limit = 0.02;
            router_efficiency = 0.5;
            repeater_efficiency = 0.5;
            average_fanouts = 3;
            clock_margin = 0.2;
            HTree_max_span = 3e-3;
            latches_per_buffer = 20;
            clock_factor = 1.0;
            clock_gating_factor = 0.4;
            max_tier = 40;
        };
    };
};
```

A.4 McPAT

McPAT is comprised of multiple models that can be individually instantiated in KitFox. Below is the list of McPAT models and their sub-models (cache of array for example).

- array: Storage unit model
  - memory: Memory model
  - cache: Cache model
  - ram: SRAM model
  - cam: CAM model
- dep_resource_conflict_check: Dependency check logic model (instruction dependency)
- flash_controller: Flash controller model
- functional_unit: Functional unit model
  - alu: ALU model
  - mul: MUL/DIV model
  - fpu: FPU model
- inst_decoder: Instruction decoder model
- interconnect: Interconnect wire model
- memory_controller: Memory controller model including Frontend, Backend, and PHY
- noc: Network model
  - noc_bus: Network bus model
  - noc_router: Network router model
- niu_controller: NIU controller model
- pice_controller: PCIe controller model
- pipeline: Latches of core pipeline stages
- selection_logic: Selection logic model (reservation station)
- undiff_core: Undifferentiated core model that is not functionally categorized such as some custom logics

The following shows an example of configuring the McPAT models listed above. In the example, McPAT components are defined as sub-components of the root component package (refer to Section 4.2 for multi-level components and configuration) that defines common parameters for different types of McPAT models.

```
component: {
    package: {
        library: {
            model = "none";

            /* Common parameters */
            clock_frequency = 3.0e9; // 3GHz clock frequency
            voltage = 1.2; // 1.2V
            features_size = 22nm; // 22nm technology
            temperature = 350; // Avg temperature

            /* Optimization parameters */
            opt_for_clk = true;
            obj_func_dyn_energy = 0;
            obj_func_dyn_power = 0;
            obj_func_leak_power = 0;
            obj_func_cycle_t = 1;
            delay_wt = 100;
            area_wt = 0;
            dynamic_power_wt = 100;
            leakage_power_wt = 0;
            cycle_time_wt = 0;
            delay_dev = 10000;
```

```
    area_dev = 10000;
    dynamic_power_dev = 10000;
    leakage_power_dev = 10000;
    cycle_time_dev = 10000;
    ed =  2;
    nuca = 0;
    nuca_bank_count = 0;
    delay_wt_nuca = 0;
    area_wt_nuca = 0;
    dynamic_power_wt_nuca = 0;
    leakage_power_wt_nuca = 0;
    cycle_time_wt_nuca = 0;
    delay_dev_nuca = 10000;
    area_dev_nuca = 10000;
    dynamic_power_dev_nuca = 10;
    leakage_power_dev_nuca = 10000;
    cycle_time_dev_nuca = 10000;
    force_wiretype = false;
    rpters_in_htree = true;
    with_clock_grid = false;
    force_cache_config = false;

    wire_type = "global"; /* "global", "global_5", "global_10", "global_20",
                "global_30", "low_swing", "semi_global", "transmission", or "optical" */
    wiring_type = "global"; // "local", "semi_global", "global", or "dram"
    device_type = "hp"; // "hp", "lstp", "lop", "lp_dram", or "comm_dram"
    interconnect_projection = "aggressive"; // "aggressive" or "conservative"
    embedded = false;
    longer_channel_device = false;
};

component: {
    core: { // package.core
        library: {
            component_type = "core"; // "core", "llc", or "uncore"
            core_type = "ooo"; // "ooo" or "inorder"
            opt_local = false;
        };
        component: {
            array_cache: { // package.core.array_cache
                library: {
                    model = "mcpat";
                    energy_model = "array";
                    energy_submodel = "cache";

                    line_size = 64; // 64B line size
                    out_width = 512; // line_size*8 bits of output width
                    assoc = 4; // 4-way assoc
                    size = 32768; // 32KB size
                    tag_width = 44; // Tag bits
                    num_rw_ports = 1; // R/W ports
                    num_rd_ports = 0;
                    num_wr_ports = 0;
                    num_se_rd_ports = 0;
                    num_search_ports = 0; // Only for fully-associative cache
                    nbanks = 1; // 1 bank
                    cycle_time = 1; // Cycle time
                    access_time = 1; // Access time
                    access_mode = "normal"; // "normal", "sequential", or "fast"
```

```
        add_ecc_b = true; // Model ECC bits?
        adjust_area = false;
    };
};

dep_resource_conflict_check: { // package.core.dep_resource_conflict_check
    library: {
        model = "mcpat";
        energy_model = "dep_resource_conflict_check";

        compare_bits = 4;
        decode_width = 4;
    };
};

functional_unit_alu: { // package.core.functional_unit_alu
    library: {
        model = "mcpat";
        energy_model = "functional_unit";
        energy_submodel = "alu";

        scaling = 4; // Model 4 ALUs with this component.
    };
};

functional_unit_mul: { // package.core.functional_unit_mul
    library: {
        model = "mcpat";
        energy_model = "functional_unit";
        energy_submodel = "mul";
    };
};

functional_unit_fpu: { // package.core.functional_unit_fpu
    library: {
        model = "mcpat";
        energy_model = "functional_unit";
        energy_submodel = "fpu";
    };
};

inst_decoder: { // package.core.inst_decoder
    library: {
        model = "mcpat";
        energy_model = "inst_decoder";

        x86 = true; // x86 instruction decoder?
        decode_length = 16; // opcode length
        scaling = 4; // Model 4 decoders with this component
    };
};

interconnect: { // package.core.interconnect
    library: {
        model = "mcpat";
        energy_model = "interconnect";

        wire_length = 1e-4; // 100um
        pipelinable = false;
```

```
            opt_local = false;
            data_width = 64; // 64-bit data width
            wire_type = "global";
        };
    };

    pipeline: { // package.core.pipeline
        library: {
            model = "mcpat";
            energy_model = "pipeline";

            pipeline_stages = 32;
            x86 = true;
            micro_opcode_length = 16;
            pc_width = 64;
            fetch_width = 4;
            decode_width = 4;
            issue_width = 6;
            commit_width = 4;
            instruction_length = 128;
            int_data_width = 64;
            num_hthreads = 1; // Hyperthreading
            thread_states = 32; // Num of per-thread states
            arch_int_regs = 32;
            phy_int_regs = 128;
            virtual_address_width = 64;
        };
    };

    selection_logic: { // package.core.selection_logic
        library: {
            model = "mcpat";
            energy_model = "selection_logic";

            selection_input = 36;
            selection_output = 6;
        };
    };

    undiff_core: { // package.core.undiff_core
        library: {
            model = "mcpat";
            energy_model = "undiff_core";

            pipeline_stages = 32;
            issue_width = 6;
            num_hthreads = 1;
            opt_clockrate = false;
        };
    };
};
}; // End of package.core

uncore: {
    library: {
        component_type = "uncore"; // "core", "llc", or "uncore"
    };
    component: {
        flash_controller: { // package.core.flash_controller
```

```
    library: {
        model = "mcpat";
        energy_model = "flash_controller";

        controller_type = "low_power"; // "high_performance" or "low_power"
        total_load_percentage = 0.7;
        peak_transfer_rate = 200; // 200MB/S
        withPHY = true;
    };
};

memory_controller: { // package.core.memory_controller
    library: {
        model = "mcpat";
        energy_model = "memory_controller";

        controller_type = "low_power"; // "high_performance" or "low_power"
        line_size = 64; // Buffer line size
        databus_width = 64;
        addressbus_width = 64;
        num_channels = 2;
        transfer_rate = 3200; // 3200MB/S peak transfer rate
        num_ranks = 2;
        lvds = true;
        withPHY = false;
        request_window_size = 128;
        IO_buffer_size = 64;
        physical_address_width = 51;
    };
};

niu_controller: { // package.core.niu_controller
    library: {
        model = "mcpat";
        energy_model = "niu_controller";

        controller_type = "low_power"; // "high_performance" or "low_power"
        total_load_percentage = 0.7;
    };
};

noc_router: { // package.core.noc_router
    library: {
        model = "mcpat";
        energy_model = "noc";
        energy_submodel = "noc_router";

        input_ports = 5;
        output_ports = 5;
        virtual_channels = 4;
        input_buffer_entries = 8;
        flit_bits = 128;
        duty_cycle = 0.3;
        traffic_pattern = 1.0;

    };
};

noc_bus: { // package.core.noc_bus
```

```
                        library: {
                            model = "mcpat";
                            energy_model = "noc";
                            energy_submodel = "noc_bus";

                            link_throughput = 1;
                            link_latency = 1;
                            chip_coverage = 1.0;
                            route_over_percentage = 0.5;
                            link_length = 10e-3; // or define "chip_area = 100e-6;"
                        };
                    };

                    pcie_controller: { // package.core.pcie_controller
                        library: {
                            model = "mcpat";
                            energy_model = "pcie_controller";

                            controller_type = "low_power"; // "high_performance" or "low_power"
                            total_load_percentage = 0.7;
                            withPHY = true;
                            num_channels = 8;
                        };
                    };
                };
            }; // End of package.uncore
        };
    }; // End of package
};
```

A.5 Failure

Failure includes the models of various failure mechanisms. Below is an example of using Failure model.

```
component: {
    failure: {
        library: {
            model = "failure";
            feature_size = 22e-9; // 22nm technology
            voltage = 1.0; // 1.0V
            clock_frequency = 2.0e9; // 2GHz

            failure_model: {
                hci: { // HCI model
                    n = 3.0;
                    Ea = -0.1;
                };

                em: { // Electromigration model
                    n = 2.0;
                    Ea = 0.9;
                };

                nbti: { // NBTI model
                    n = 5.0;
                    Ea = 0.4;
                };

                sm: { // Stress migration model
                    T0 = 500.0;
                    n = 2.5;
                    Ea = 0.9;
                };

                tddb: { // TDDB model
                    a = 78.0;
                    b = -0.081;
                    c = 0.1;
                    x = -0.759;
                    y = 66.8;
                    z = 8.37e-4;
                    Ea = 0.7;
                };
            };

            /* Target conditions that the failure models are tuned to meet the MTTF */
            target_temperature = 338;
            target_voltage = 1.0;
            target_frequency = 2.0e9;
            target_MTTF = 3.15532800e8; // 10 years of MTTF (in seconds)
        };
    };
};
```

A.6 3D-ICE

3D-ICE supports liquid cooling model in the 3D package. In the example below, 3D-ICE is associated with the root-level component package, and its sub-components are designated as floorplans. Each partition component specifies the geometry of the floorplan.

```
component: {
    package: {
        library: {
            model = "3d-ice";
            thermal_analysis = "transient";
            ambient_temperature = 300.0;
            temperature = 320.0; // Initial temperature
            chip_width = 10e-3; // 10mm
            chip_height = 10e-3; // 10mm
            grid_rows = 100;
            grid_cols = 100;
            grid_map_mode = "avg"; // "avg", "center", "min", or "max"

            material: {
                SILICON: {
                    thermal_conductivity = 1.30e-4;
                    volumetric_heat_capacity = 1.638e-12;
                };

                BEOL: {
                    thermal_conductivity = 2.25e-6;
                    volumetric_heat_capacity = 2.175e-12;
                };
            };

            conventional_heat_sink: {
                heat_transfer_coefficient = 1.0e-7;
            };

            microchannel: {
                type = "2rm";
                height = 100e-6;
                channel_length = 50e-6;
                wall_length = 50e-6;
                wall_material = "SILICON";
                coolant_flow_rate = 42;
                coolant_heat_transfer_coefficient_top = 5.7132e-8;
                coolant_heat_transfer_coefficient_bottom = 4.7132e-8;
                coolant_heat_volumetric_heat_capacity = 4.172e-12;
                coolant_incoming_temperature = 300.0;
            };

            die: {
                TOP_IC: {
                    layer: {
                        /* Layers must be listed sequentially from the top to bottom. */
                        SOURCE_LAYER: {
                            height = 2e-6;
                            material = "SILICON";
                            is_source_layer = true;
                        };
                        BOTTOM_LAYER: {
                            height = 50e-6;
                            material = "SILICON";
```

```
                is_source_layer = false;
            };
        };
    };

    BOTTOM_IC: {
        layer: {
            TOP_LAYER: {
                height = 10e-6;
                material = "BEOL";
                is_source_layer = false;
            };
            SOURCE_LAYER: {
                height = 2e-6;
                material = "SILICON";
                is_source_layer = true;
            };
            BOTTOM_LAYER: {
                height = 50e-6;
                material = "SILICON";
                is_source_layer = false;
            };
        };
   };
};

stack: {
    // IMPORTANT: Stacks must be listed sequentially from the top to bottom.
    MEMORY_DIE: {
        type = "die";
        die = "TOP_IC";
    };
    TOP_CHANNEL: {
        type = "channel";
    };
    CORE_DIE: {
        type = "die";
        die = "BOTTOM_IC";
    };
    BOTTOM_CHANNEL: {
        type = "channel";
    };
    BOTTOM_MOST: {
        type = "layer";
        height = 10e-6;
        material = "BEOL";
        is_source_layer = false;
    };
};

partition = ["package.MEMORY_DIE-Background1",
             "package.MEMORY_DIE-Background2",
             "package.MEMORY_DIE-HotSpot1",
             "package.MEMORY_DIE-HotSpot2",
             "package.CORE_DIE-Core0",
             "package.CORE_DIE-Core1",
             "package.CORE_DIE-Core2",
             "package.CORE_DIE-Core3",
             "package.CORE_DIE-Core4",
```

```
                "package.CORE_DIE-Core5",
                "package.CORE_DIE-Core6",
                "package.CORE_DIE-Core7",
                "package.CORE_DIE-Cache0",
                "package.CORE_DIE-Cache1",
                "package.CORE_DIE-CLK",
                "package.CORE_DIE-FPU",
                "package.CORE_DIE-CrossBar"];
};

/* Components that are designated as floorplans */
component: {
    MEMORY_DIE-Background1: { // package.MEMORY_DIE-Background1
        library: {
            model = "none";
            dimension: {
                left = 0.0;
                bottom = 0.0;
                width = 5e-3;
                height = 5e-3;
                die_name = "MEMORY_DIE";
                die_index = 1;
            };
        };
    };

    MEMORY_DIE-Background2: { // package.MEMORY_DIE-Background2
        library: {
            model = "none";
            dimension: {
                left = 5e-3;
                bottom = 5e-3;
                width = 5e-3;
                height = 5e-3;
                die_name = "MEMORY_DIE";
                die_index = 1;
            };
        };
    };

    MEMORY_DIE-HotSpot1: { // package.MEMORY_DIE-HotSpot1
        library: {
            model = "none";
            dimension: {
                left = 5e-3;
                bottom = 0.0;
                width = 5e-3;
                height = 5e-3;
                die_name = "MEMORY_DIE";
                die_index = 1;
            };
        };
    };

    MEMORY_DIE-HotSpot2: { // package.MEMORY_DIE-HotSpot2
        library: {
            model = "none";
            dimension: {
                left = 0.0;
```

```
            bottom = 5e-3;
            width = 5e-3;
            height = 5e-3;
            die_name = "MEMORY_DIE";
            die_index = 1;
        };
    };
};

CORE_DIE-Core0: { // package.CORE_DIE-Core0
    library: {
        model = "none";
        dimension: {
            left = 0.0;
            bottom = 0.0;
            width = 2.5e-3;
            height = 3.5e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-Core1: { // package.CORE_DIE-Core1
    library: {
        model = "none";
        dimension: {
            left = 2.5e-3;
            bottom = 0.0;
            width = 2.5e-3;
            height = 3.5e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-Core2: { // package.CORE_DIE-Core2
    library: {
        model = "none";
        dimension: {
            left = 5.0e-3;
            bottom = 0.0;
            width = 2.5e-3;
            height = 3.5e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-Core3: { // package.CORE_DIE-Core3
    library: {
        model = "none";
        dimension: {
            left = 7.5e-3;
            bottom = 0.0;
            width = 2.5e-3;
            height = 3.5e-3;
```

```
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-Core4: { // package.CORE_DIE-Core4
    library: {
        model = "none";
        dimension: {
            left = 0.0;
            bottom = 6.5e-3;
            width = 2.5e-3;
            height = 3.5e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-Core5: { // package.CORE_DIE-Core5
    library: {
        model = "none";
        dimension: {
            left = 2.5e-3;
            bottom = 6.5e-3;
            width = 2.5e-3;
            height = 3.5e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-Core6: { // package.CORE_DIE-Core6
    library: {
        model = "none";
        dimension: {
            left = 5.0e-3;
            bottom = 6.5e-3;
            width = 2.5e-3;
            height = 3.5e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-Core7: { // package.CORE_DIE-Core7
    library: {
        model = "none";
        dimension: {
            left = 7.5e-3;
            bottom = 6.5e-3;
            width = 2.5e-3;
            height = 3.5e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
```

```
    };
};

CORE_DIE-Cache0: { // package.CORE_DIE-Cache0
    library: {
        model = "none";
        dimension: {
            left = 0.0;
            bottom = 3.5e-3;
            width = 1.25e-3;
            height = 3.0e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-Cache1: { // package.CORE_DIE-Cache1
    library: {
        model = "none";
        dimension: {
            left = 8.75e-3;
            bottom = 3.5e-3;
            width = 1.25e-3;
            height = 3.0e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-CLK: { // package.CORE_DIE-CLK
    library: {
        model = "none";
        dimension: {
            left = 1.25e-3;
            bottom = 3.5e-3;
            width = 1.25e-3;
            height = 3.0e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};

CORE_DIE-FPU: { // package.CORE_DIE-FPU
    library: {
        model = "none";
        dimension: {
            left = 7.5e-3;
            bottom = 3.5e-3;
            width = 1.25e-3;
            height = 3.0e-3;
            die_name = "CORE_DIE";
            die_index = 0;
        };
    };
};
```

```
            CORE_DIE-CrossBar: { // package.CORE_DIE-Crossbar
                library: {
                    model = "none";
                    dimension: {
                        left = 2.5e-3;
                        bottom = 3.5e-3;
                        width = 5.0e-3;
                        height = 3.0e-3;
                        die_name = "CORE_DIE";
                        die_index = 0;
                    };
                };
            };
        };
    }; // End of package
};
```

A.7 HotSpot

HotSpot is a 2D-package thermal model. Although it can be tweaked to simulate a 3D package, missing TSV models would not correctly model the thermal characteristics of the 3D package.

```
component: {
    package: {
        library: {
            model = "hotspot";
            thermal_analysis = "transient"; // "transient" or "steady_state";
            model_type = "grid"; // "grid" or "block";
            grid_map_mode = "avg"; // "avg", "center", "min", or "max"

            thermal_threshold = 345.95;
            ambient_temperature = 318.15;
            temperature = 333.15;

            chip_thickness = 0.15e-3;
            chip_thermal_conductivity = 100.0;
            chip_heat = 1.75e6;

            heatsink_convection_capacitance = 140.4;
            heatsink_convection_resistance = 0.1;
            heatsink_side = 60e-3;
            heatsink_thickness = 6.9e-3;
            heatsink_thermal_conductivity = 400.0;
            heatsink_heat = 3.55e6;

            spreader_side = 30e-3;
            spreader_thickness = 1e-3;
            spreader_thermal_conductivity = 400.0;
            spreader_heat = 3.55e6;

            interface_thickness = 20e-6;
            interface_thermal_conductivity = 4.0;
            interface_heat = 4.0e6;

            secondary_model = false;
            secondary_convection_resistance = 50.0;
            secondary_convection_capacitance = 40.0;
            metal_layers = 8;
            metal_layer_thickness = 100e-6;

            c4_thickness = 0.0001;
            c4_side = 20e-6;
            c4_pads = 400;

            substrate_side = 0.021;
            substrate_thickness = 1e-3;

            solder_side = 0.021;
            solder_thickness = 0.00094;

            pcb_side = 0.1;
            pcb_thickness = 0.002;

            dtm_used = false;
            leakage_used = false;
            package_model_used = false;
```

```
    grid_rows = 64; // Must be power of 2
    grid_cols = 64; // Must be power of 2

    layer: {
        tim: {
            index = 1;
            is_source_layer = false;
            use_layer_partition = 0;
            lateral_heat_flow = true;
            heat = 4e6;
            resistance = 0.25;
            thickness = 20e-6;
        };

        silicon: {
            index = 0;
            is_source_layer = true;
            lateral_heat_flow = true;
            heat = 1.75e6;
            resistance = 0.01;
            thickness = 0.15e-3;
        };
    };

    partition = ["package.L2_left",
                 "package.L2",
                 "package.L2_right",
                 "package.Icache",
                 "package.Dcache",
                 "package.Bpred_0",
                 "package.Bpred_1",
                 "package.Bpred_2",
                 "package.DTB_0",
                 "package.DTB_1",
                 "package.DTB_2",
                 "package.FPAdd_0",
                 "package.FPAdd_1",
                 "package.FPReg_0",
                 "package.FPReg_1",
                 "package.FPReg_2",
                 "package.FPReg_3",
                 "package.FPMul_0",
                 "package.FPMul_1",
                 "package.FPMap_0",
                 "package.FPMap_1",
                 "package.IntMap",
                 "package.IntQ",
                 "package.IntReg_0",
                 "package.IntReg_1",
                 "package.IntExec",
                 "package.FPQ",
                 "package.LdStQ",
                 "package.ITB_0",
                 "package.ITB_1"];
};

component: {
    L2_left: { // package.L2_left
        library: {
```

```
        model = "none";
        dimension: {
            left = 0.0e-3;
            bottom = 9.8e-3;
            width = 4.9e-3;
            height = 6.2e-3;
            die_index = 0;
        };
    };
};

L2: { // package.L2
    library: {
        model = "none";
        dimension: {
            left = 0.0e-3;
            bottom = 0.0e-3;
            width = 16.0e-3;
            height = 9.8e-3;
            die_index = 0;
        };
    };
};

L2_right: { // package.L2_right
    library: {
        model = "none";
        dimension: {
            left = 11.1e-3;
            bottom = 9.8e-3;
            width = 4.9e-3;
            height = 6.2e-3;
            die_index = 0;
        };
    };
};

Icache: { // package.Icache
    library: {
        model = "none";
        dimension: {
            left = 4.9e-3;
            bottom = 9.8e-3;
            width = 3.1e-3;
            height = 2.6e-3;
            die_index = 0;
        };
    };
};

Dcache: { // package.Dcache
    library: {
        model = "none";
        dimension: {
            left = 8.0e-3;
            bottom = 9.8e-3;
            width = 3.1e-3;
            height = 2.6e-3;
            die_index = 0;
```

```
        };
    };
};

Bpred_0: { // package.Bpred_0
    library: {
        model = "none";
        dimension: {
            left = 4.9e-3;
            bottom = 12.4e-3;
            width = 1.033e-3;
            height = 0.7e-3;
            die_index = 0;
        };
    }:
};

Bpred_1: { // package.Bpred_1
    library: {
        model = "none";
        dimension: {
            left = 5.933e-3;
            bottom = 12.4e-3;
            width = 1.033e-3;
            height = 0.7e-3;
            die_index = 0;
        };
    };
};

Bpred_2: { // package.Bpred_2
    library: {
        model = "none";
        dimension: {
            left = 6.967e-3;
            bottom = 12.4e-3;
            width = 1.033e-3;
            height = 0.7e-3;
            die_index = 0;
        };
    };
};

DTB_0: { // package.DTB_0
    library: {
        model = "none";
        dimension: {
            left = 8.0e-3;
            bottom = 12.4e-3;
            width = 1.033e-3;
            height = 0.7e-3;
            die_index = 0;
        };
    };
};

DTB_1: { // package.DTB_0
    library: {
        model = "none";
```

```
        dimension: {
            left = 9.033e-3;
            bottom = 12.4e-3;
            width = 1.033e-3;
            height = 0.7e-3;
            die_index = 0;
        };
    };
};

DTB_2: { // package.DTB_2
    library: {
        model = "none";
        dimension: {
            left = 10.067e-3;
            bottom = 12.4e-3;
            width = 1.033e-3;
            height = 0.7e-3;
            die_index = 0;
        };
    };
};

FPAdd_0: { // package.FPAdd_0
    library: {
        model = "none";
        dimension: {
            left = 4.9e-3;
            bottom = 13.1e-3;
            width = 1.1e-3;
            height = 0.9e-3;
            die_index = 0;
        };
    };
};

FPAdd_1: { // package.FPAdd_1
    library: {
        model = "none";
        dimension: {
            left = 6.0e-3;
            bottom = 13.1e-3;
            width = 1.1e-3;
            height = 0.9e-3;
            die_index = 0;
        };
    };
};

FPReg_0: { // package.FPReg_0
    library: {
        model = "none";
        dimension: {
            left = 4.9e-3;
            bottom = 14.0e-3;
            width = 0.55e-3;
            height = 0.38e-3;
            die_index = 0;
        };
```

```
    };
};

FPReg_1: { // package.FPReg_1
    library: {
        model = "none";
        dimension: {
            left = 5.4e-3;
            bottom = 14.0e-3;
            width = 0.55e-3;
            height = 0.38e-3;
            die_index = 0;
        };
    };
};

FPReg_2: { // package.FPReg_2
    library: {
        model = "none";
        dimension: {
            left = 6.0e-3;
            bottom = 14.0e-3;
            width = 0.55e-3;
            height = 0.38e-3;
            die_index = 0;
        };
    };
};

FPReg_3: { // package.FPReg_3
    library: {
        model = "none";
        dimension: {
            left = 6.55e-3;
            bottom = 14.0e-3;
            width = 0.55e-3;
            height = 0.38e-3;
            die_index = 0;
        };
    };
};

FPMul_0: { // package.FPMul_0
    library: {
        model = "none";
        dimension: {
            left = 4.9e-3;
            bottom = 14.38e-3;
            width = 1.1e-3;
            height = 0.95e-3;
            die_index = 0;
        };
    };
};

FPMul_1: { // package.FPMul_1
    library: {
        model = "none";
        dimension: {
```

```
            left = 6.0e-3;
            bottom = 14.38e-3;
            width = 1.1e-3;
            height = 0.95e-3;
            die_index = 0;
        };
    };
};

FPMap_0: { // package.FPMap_0
    library: {
        model = "none";
        dimension: {
            left = 4.9e-3;
            bottom = 15.33e-3;
            width = 1.1e-3;
            height = 0.67e-3;
            die_index = 0;
        };
    };
};

FPMap_1: { // package.FPMap_1
    library: {
        model = "none";
        dimension: {
            left = 6.0e-3;
            bottom = 15.33e-3;
            width = 1.1e-3;
            height = 0.67e-3;
            die_index = 0;
        };
    };
};

IntMap: { // package.IntMap
    library: {
        model = "none";
        dimension: {
            left = 7.1e-3;
            bottom = 14.65e-3;
            width = 0.9e-3;
            height = 1.35e-3;
            die_index = 0;
        };
    };
};

IntQ: { // package.IntQ
    library: {
        model = "none";
        dimension: {
            left = 8.0e-3;
            bottom = 14.65e-3;
            width = 1.3e-3;
            height = 1.35e-3;
            die_index = 0;
        };
    };
```

```
};

IntReg_0: { // package.IntReg_0
    library: {
        model = "none";
        dimension: {
            left = 9.3e-3;
            bottom = 15.33e-3;
            width = 0.9e-3;
            height = 0.67e-3;
            die_index = 0;
        };
    };
};

IntReg_1: { // package.IntReg_1
    library: {
        model = "none";
        dimension: {
            left = 10.2e-3;
            bottom = 15.33e-3;
            width = 0.9e-3;
            height = 0.67e-3;
            die_index = 0;
        };
    };
};

IntExec: { // package.IntExec
    library: {
        model = "none";
        dimension: {
            left = 9.3e-3;
            bottom = 13.1e-3;
            width = 1.8e-3;
            height = 2.23e-3;
            die_index = 0;
        };
    };
};

FPQ: { // package.FPQ
    library: {
        model = "none";
        dimension: {
            left = 7.1e-3;
            bottom = 13.1e-3;
            width = 0.9e-3;
            height = 1.55e-3;
            die_index = 0;
        };
    };
};

LdStQ: { // package.LdStQ
    library: {
        model = "none";
        dimension: {
            left = 8.0e-3;
```

```
                    bottom = 13.7e-3;
                    width = 1.3e-3;
                    height = 0.95e-3;
                    die_index = 0;
                };
            };
        };

        ITB_0: { // package.ITB_0
            library: {
                model = "none";
                dimension: {
                    left = 8.0e-3;
                    bottom = 13.1e-3;
                    width = 0.65e-3;
                    height = 0.6e-3;
                    die_index = 0;
                };
            };
        };

        ITB_1: { // package.ITB_1
            library: {
                model = "none";
                dimension: {
                    left = 8.65e-3;
                    bottom = 13.1e-3;
                    width = 0.65e-3;
                    height = 0.6e-3;
                    die_index = 0;
                };
            };
        };

    };
}; // End of package
};
```

A.8 Microfluidics

Microfluidics is a compact 3D package thermal model with liquid cooling.

```
component: {
    package: {
        library: {
            model = "microfluidics";
            thermal_analysis = "steady_state";
            grid_map_mode = "average";

            /* Pin Fin dimension */
            HC = 200e-6;
            Dp = 100e-6;
            NT = 34;
            NL = 34;

            /* Chip dimension */
            ThSi = 100e-6;
            ThO = 10e-6;
            ThB = 10e-6;

            /* Pumping power */
            PPower = 0.03;

            heff = 562.28;
            hamb = 10.0;
            Tfin = 293.15;
            temperature = 293.15; // Initial temperature
            ambient_temperature = 293.15;

            rho_Si = 2330.0;
            cp_Si = 707.0;
            k_Si = 149.0;
            rho_O = 220.0;
            k_O = 1.4;
            cp_B = 1000.0;
            k_B = 1.4;
            rho_f = 997.0;
            cp_f = 4183.0;
            k_f = 0.5945;
            mu_f = 0.0008936;

            partition = ["package.proc.left_bottom",
                         "package.proc.right_bottom",
                         "package.proc.left_top",
                         "package.proc.right_top",
                         "package.mem.left_bottom",
                         "package.mem.right_bottom",
                         "package.mem.left_top",
                         "package.mem.right_top"];
        };

        component: {
            proc: { // package.proc
                component: {
                    left_bottom: { // package.proc.left_bottom
                        library: {
                            model = "none";
                            dimension: {
```

```
                left = 0.0e-3;
                bottom = 0.0e-3;
                width = 4.2e-3;
                height = 4.2e-3;
                die_index = 0;
            };
        };
    };

    right_bottom: { // package.proc.right_bottom
        library: {
            model = "none";
            dimension: {
                left = 4.2e-3;
                bottom = 0.0e-3;
                width = 4.2e-3;
                height = 4.2e-3;
                die_index = 0;
            };
        };
    };

    left_top: { // package.proc.left_top
        library: {
            model = "none";
            dimension: {
                left = 0.0e-3;
                bottom = 4.2e-3;
                width = 4.2e-3;
                height = 4.2e-3;
                die_index = 0;
            };
        };
    };

    right_top: { // package.proc.right_top
        library: {
            model = "none";
            dimension: {
                left = 4.2e-3;
                bottom = 4.2e-3;
                width = 4.2e-3;
                height = 4.2e-3;
                die_index = 0;
            };
        };
    };
  };
}; // End of package.proc

mem: { // package.mem
    component: {
        left_bottom: { // package.mem.left_bottom
            library: {
                model = "none";
                dimension: {
                    left = 0.0e-3;
                    bottom = 0.0e-3;
                    width = 4.2e-3;
```

```
                        height = 4.2e-3;
                        die_index = 1;
                    };
                };
            };

            right_bottom: { // package.mem.right_bottom
                library: {
                    model = "none";
                    dimension: {
                        left = 4.2e-3;
                        bottom = 0.0e-3;
                        width = 4.2e-3;
                        height = 4.2e-3;
                        die_index = 1;
                    };
                };
            };

            left_top: { // package.mem.left_top
                library: {
                    model = "none";
                    dimension: {
                        left = 0.0e-3;
                        bottom = 4.2e-3;
                        width = 4.2e-3;
                        height = 4.2e-3;
                        die_index = 1;
                    };
                };
            };

            right_top: { // package.mem.right_top
                library: {
                    model = "none";
                    dimension: {
                        left = 4.2e-3;
                        bottom = 4.2e-3;
                        width = 4.2e-3;
                        height = 4.2e-3;
                        die_index = 1;
                    };
                };
            };
        };
    }; // End of package.mem
  };
}; // End of package
};
```