JADE Simulation Platform

User Manual

Version 5.0

OPTICS Lab

Big Data System Lab

Hong Kong University of Science and Technology

https://eexu.home.ece.ust.hk

December 2020

CONTENTS

I Motivation 4

II Introduction 5

II-A New features in JADE Version 5.0 6

III JADE organization 7

III-A Root directory 7

III-B Workspace 7

IV Installation guideline 9

IV-A Requirements 9

IV-B Quick compilation 9

IV-C Advanced compilation 9

V Recommended workflow 11

VI Running simulation 13

VI-A Quick guideline 13

VII Benchmark applications 15

VII-A Realistic statistical application model 15

VII-B Directory organization and naming of application models 15

VII-C Pipeline stage partitioning 16

VII-D Mapping and scheduling 17

VII-E Synthetic traffic 18

VIII Architectures 19

VIII-A Network topology file 19

VIII-B Custom network topologies 20

VIII-C Optical switch fabrics 21

VIII-D Memory hierarchy and cache coherence protocol 22

VIII-E External memory simulation 22

VIII-F Components configuration file 24

VIII-G Power models 24

IX Output items 27

Agreement and License 28

Revision History 29

References 30

Appendix-A: Basic Command Line Options 31

Appendix-B: Advanced Command Line Options 35

Appendix-C: List of Applications in COSMIC Benchmark Suite 37

I. MOTIVATION

Recent advances in the computing industry towards multiprocessor technologies shifted the dominant method of performance increase from frequency scaling to parallelism. Due to its huge design space, evaluating candidate multicore processors and tens or hundreds of such processor architectures in early design stages, when the number of variables is at its maximum, is challenging. Simulation plays an important role in estimating architecture performance, and evaluating how the system would perform on average, as well as boundary cases, would require many iterations to cover various cases in the application input domain. Since the simulation of heterogeneous systems with enough details are naturally slow, exhaustively evaluating the system for all possible inputs requires a tremendous amount of time and resources. While there exist quite a few multiprocessor simulators available, they often rely on individual input specification, demanding extensive input enumeration and simulation runs, diminishing their effectiveness for complex systems evaluation. Aiming to fulfill this gap, we publicly release a heterogeneous multiprocessor system simulation platform called JADE, targeting fast initial architecture explorations. Opposing to most simulators, JADE uses statistical models that follow distributions extracted from internal structures of the application, providing a more convenient and systematic exploration approach to evaluate systems performance. JADE simulation features include detailed electrical and optical interconnections for both single-chip and multi-chip systems, detailed memory hierarchy infrastructure, and built-in energy analysis allowing studies of a broad spectrum of systems.

II. Introduction

JADE is a cycle-accurate event-driven simulator, implemented using C++ and most of the configurations are templates in text form. Figure 1 shows an overview of JADE features. It receives as input the detailed description of the hardware architecture such as network topology, memory hierarchy, cache coherence protocol, and specific processor parameters. The statistical models of realistic applications are available from the heterogeneous COSMIC benchmark [1]. JADE integrates power model libraries to provide holistic power analysis for various configurations of architecture, and technology nodes. JADE also includes power management functions: Dynamic Voltage and Frequency Scaling (DVFS), Energy Consumption Calculation, and Power Delivery Systems (PDS). Along with memory access, holistic performance and power analysis, it is also possible to extract customized system behavior to monitor specific components. The following sections detail selected features implemented in JADE.

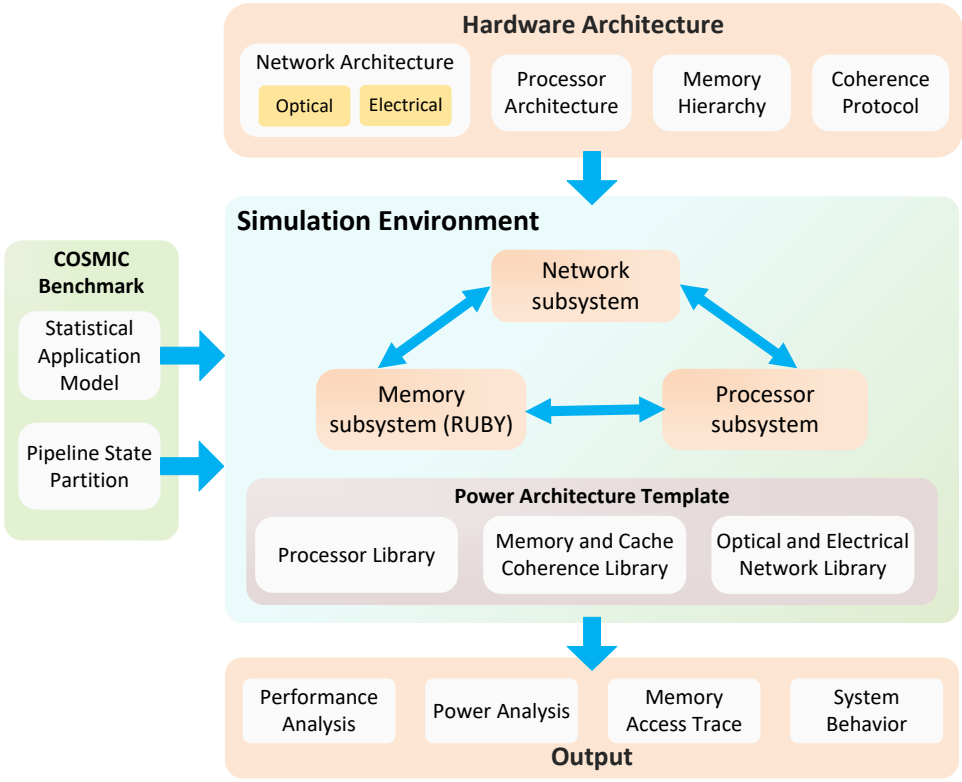


Fig. 1: Overview of JADE.

By performing cycle-level simulations, JADE can model the detailed behaviors of each module and comprehensively profile the characteristics of the system components. A set of templates are already provided with different network-on-chip (NoC) and common communication topologies, such as Crossbar, Ring, Torus, Mesh, Folded-Torus, and Fat-tree with a various count of cores, caches, and memories. These templates can be easily modified or extended to accommodate a wide range of simulation schemes, allowing high flexibility of components placement and the creation of new regular and irregular topologies.

JADE uses the COSMIC Heterogeneous Multiprocessor Benchmark as a stimulus. COSMIC provides application models profiled in two different instruction set architectures: `ARM-v8` and `x86-64`. The application models statistically mimic the behavior of the original application by profiling the memory and synchronization traces. Though a few application benchmarks are packed along with JADE 5.0, readers are referred to the COSMIC release page at https://eexu.home.ece.ust.hk/COSMIC.html for more application benchmarks.

A. New features in JADE Version 5.0

The new version of JADE includes important improvements in benchmark applications as well as their models, the mapping and scheduling algorithm, the execution mode, and the inter-chip network.

The newest version of JADE includes more applications from some widely-used benchmark suites, including APEX [2], NAS [3], SPEC2006 [4], SPLASH-3 [5] and PARSEC 3.0 [6]. Applications are modeled by the fork/barrier/join model. In this model, the execution flow of the application is divided into parallel sections (PSs) whenever threads are created, merged, or synchronized. We insert instrumentation calls in the source code and use QEMU to get the application and memory traces automatically. In the new application model, each PS consists of multiple basic thread blocks (BTBs). We extract the main behavior statistics with micro-architecture independent metrics for each BTB, including memory locality for data and instruction and the distribution of the request type. The locality model does not only consider local memory accesses but also takes data sharing among threads into consideration to capture the memory behaviors more accurately. During the simulation, the memory trace is reproduced according to the new application model.

In JADE 5.0, the mapping and scheduling is done at runtime hierarchically, i.e., first at chip-level and then at core-level. We offer several commonly-used algorithms in JADE 5.0, but users can modify the source code to specify their preferred mapping and scheduling approaches. JADE 5.0 offers two simulation modes: multi-threading and pipelined execution. For pipelined execution, we provide a tool to partition applications into multiple pipeline stages. The execution of one stage can be constrained to a specific machine, a specific chip, or a specific region of the chip during the simulation.

The new version of JADE can simulate computing systems in which chips are interconnected by optical links. The inter-chip optical interconnect can be implemented as one of the following switch fabrics: Crossbar, DRAGRON, Spiral, PILOSS, Benes, Baseline, FODON, and AWGR. Simulating the system based on a variety of synthetic traffic is another new feature of JADE 5.0. It can generate one of the following synthetic traffic: uniform, tornado, hotspot, hotpair, transpose, bit complement, bit reverse, bit rotation and shuffle. Using synthetic traffic promises to analyze a large-scale computing system with a high-radix optical switch.

III. JADE ORGANIZATION

This section briefly describes the directory organization of JADE.

A. Root directory

In table I JADE directory organization is briefly explained. In this manual, it is supposed that the JADE root folder is located at `/home/JADE-v5.0` and all commands would be executed from this directory. The relevant directories for users are classified as `User space` and there reside all files required to realize simulations with different topologies, applications, and components. Additional directories include the source codes of JADE, and a set of tools provided task manipulations and file generation.

TABLE I: JADE-v5.0 root directory organization and description

Class	Directory	Description
User space	applications	Folder to place applications-related files, such as application statistical models, pipeline stage partitioning results and COSMIC APIs. To download a full list of application benchmarks, check the COSMIC page at https://eexu.home.ece.ust.hk/COSMIC.html .
	architectures	Holds hardware models, for instance architecture description, network topology, and components configurations
	workspace	Keeps all binary files and results generated from the simulation.
	documents	JADE user instructions.
Source code	common	Global headers and main definitions.
	network	Network related files.
	processor	Processing elements source code.
	ruby	Cache subsystem model.
	slicc	Code generation for cache controllers.
	protocols	Coherence protocol files.
	support	Power management files.
	dramsim2	Detailed memory activity.
Tools	scripts	Helper scripts for compilation.
	utils	Utility tools.

B. Workspace

The user workspace is composed of multiple directories for each JADE compilation and it depends on the components configuration and the coherence protocol, see section V for further details. For each coherence protocol and configuration file, a new workspace directory is created under the directory workspace in which the final binary file `jade.exec` will reside. The workspace would have a name following the configuration file used, and the coherence protocol. Down below is the syntax adopted:

`<config>-<coherence_protocol>`

Table II details more the purpose of each directory under the workspace.

TABLE II: Workspace directory description.

Class	**Directory Name**	**Description**
User Environment	bin	Keeps the compiled JADE executable file.
	result	Default destination for statistics results and system behavior output.
Compilation Output	src	Source code generated with respect to the protocol specified.

IV. Installation guideline

A. Requirements

In order to compile JADE successfully, a few software and libraries should be present in your system. Most recent Linux distributions would attend these requirements, we provide here a reference just in case you encounter any problem. Table III shows a summarized list of dependencies of JADE divided into two classes: those mandatory and those optional. Make sure all of the mentioned external software are properly set in your environment variables (`PATH`).

TABLE III: External dependencies

Software	**Version**	**Purpose**	**Quick reference**
GCC	5.2	Compilation	https://gcc.gnu.org/
CMAKE	3.15.2	Compilation	https://cmake.org/
Python	2.7.5	Code generation helper	https://www.python.org/
dos2unix	6.0.3	Convert text files from DOS format to Unix	http://www.gzip.org/
Boost	1.53.0	Provide useful data structure and functions	https://www.boost.org/
libglib2.0-dev	2.54.2	Provide general-purpose C library	https://pkgs.org/download/libglib2.0-dev/
zlib-devel	1.2.7	Data-compression library	https://www.zlib.net/

B. Quick compilation

All of the steps explained here suppose that the root JADE folder is located at `/home/JADE-v5.0`. The easiest way to compile JADE is to enter the JADE root directory and type `make`. This will compile using the default coherence protocol (MSI_MOSI_CMP_directory) and the default components configuration (`generic`) and the binary would be located at: `/workspace/generic-3level_msi_mosi_cmp_directory/jade.exec`.

```
$ cd /home/JADE-v5.0
$ make
```

When the compilation is over, a message similar to the one shown below should appear in the terminal:

```
[100%] Built target jade.exec
```

From this point, JADE is ready to be used and one may jump to section VI to have more details about how to simulate applications.

C. Advanced compilation

There are three variables that can be configured at compilation time:

- `SYSTEM_SETTINGS`: sets the default system system parameters;
- `PROTOCOL`: sets the cache coherence protocol;
- `DEBUG`: enable compilation with debugging symbols and no optimization, e.g., make DEBUG=1.

Configuration files are located under the directory `/home/JADE-v5.0/architectures/config`. The detailed configurations for each `SYSTEM_SETTINGS` can also be changed in configuration files as

discussed in subsection VIII-F. Available protocols are located under the directory `/home/JADE-v5.0/protocols`.

Example IV.1. JADE with generic components and MSI_MOSI_CMP_directory:

```
$ cd /home/JADE-v5.0
$ make SYSTEM_SETTINGS=generic PROTOCOL=MSI_MOSI_CMP_directory DEBUG=1
```

For each pair `<configuration, protocol>`, a new folder will be created inside the workspace directory and be named as `<configuration>_<protocol>`, following the component configuration file name and the cache coherence protocol used. In example IV.1, the destination folder is at `/home/JADE-v5.0/workspace/generic_msi_mosi_cmp_directory`. And the debug version will be built.

V. Recommended workflow

JADE allows customization of the components configuration such as cache size, memory latency, cache latency, and many others. The default configuration are based on a generic configuration, derived from the classical 5-stage MIPS pipeline. A few other configurations are provided under the directory `/home/JADE-v5.0/architecture/config`. Table IV lists some of the important configurations. A brief description of other important parameters of the configuration file is shown in section VIII-F. To use a different component configuration, user should compile JADE again using the flag `SYSTEM_SETTINGS`, refer to section IV for usage.

TABLE IV: Some memory configurations provided in config. files (two-level cache).

	generic	aarch64	x64
Main Memory Size (total)	8GB	8GB	8GB
L2 Cache Config. (per-core)	512KB, 8-way	512KB, 8-way	512KB, 16-way
L1 Cache Config. (per-core)	32KB, 4-way	32KB, 4-way	64KB, 8-way
Cache Line Size	64 bytes	64 bytes	64 bytes

For each coherence protocol and configuration file, a new workspace directory is created under the directory workspace in which the final binary file `jade.exec` will reside. The workspace would have a name following the configuration file used, and the cache coherence protocol. Down below is the syntax used:

`<config>_<coherence_protocol>`

Example V.1. For generic components configuration using MSI_MOSI_CMP_directory coherence protocol, the workspace would be named `generic_msi_mosi_cmp_directory`

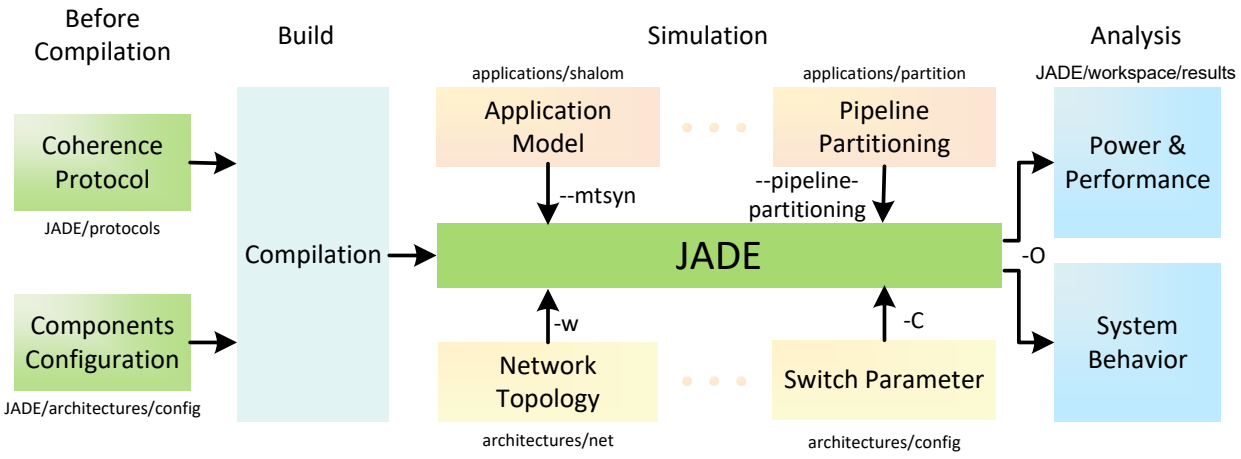


Fig. 2: Recommended workflow.

Figure 2 shows the recommend outlined workflow of JADE. The first stage is to choose which coherence protocol and components configuration is more appropriate for the simulation, compile it using make and the appropriate coherence protocol and components configuration file, then execute JADE passing the appropriate command line arguments for different architectures, network topologies, and application. After

analyzing the results, if any changes need to be made for the protocol or the components configuration re-compilation of JADE is required. To simulate with different architecture, network topology or application does not require recompiling the code, and users can freely select different inputs by using corresponding flags as given in Figure 2. It is worth noting that benchmark files are distinguished by instruction set architectures; check your JADE compilation configuration before downloading your right benchmark file at https://eexu.home.ece.ust.hk/COSMIC.html.

VI. Running simulation

This section is dedicated to clarifying how to simulate your desired system and applications using JADE. It has been divided into two parts, the first explains the simplest way to get JADE running for the first time, and later a more detailed description of each part is provided.

TABLE V: Simulation command line argument description

Class	Flag	Required	Description
General	-h	No	Print help information.
	-H	No	Print the advanced command line options to tune architecture parameters.
	-v	No	Print JADE version.
Simulation control	-r	No	Set the random seed. If not specified, value from configuration will be used. Use 0 for time() and larger than 0 for a fixed value.
	-N	No	Set the number of iterations to be performed by repeatedly executing the statistical application.
	–max-sim-time	No	Maximum simulation time. Given in time unit(ps).
	–trace-reduction	No	Will simulate with 1/X of original memory traces, X is the value set.
	–ignore-pipeline-fill	No	Will not simulate the initial phase when the pipelining is filling / draining (applicable in pipelined simulation mode).
Software and mapping	–mtsyn	YES	Statistical application model file. Under directory: JADE/applications/shalom.
	–pipelined-application	No	Pipelined execution simulation. Should give number of stages.
	–pipeline-partitioning	No	Application pipeline stage partitioning telling which PS belongs to which stage.
	–chip-region	No	Create regions within a single chip.
	–static-pipeline-stage-to-region	No	Will map pipeline stage 0 to chip region 0, stage 1 to region 1, and so on.
Hardware	-w	YES	Network topology file. Under directory: JADE/architecture/net.
	-A	No	Set the technology node (nm). Default is 7nm FINFET.
	-L	No	Set the power library file. Default power library is located at architectures/pwr/pwr lib.
	-S	No	Optical switch type. Examples: DRAGON, FODON, Benes.
	-C	No	Optical switch configuration file (.cfg). Examples are under directory JADE-v5.0/architecture/config.
Output	-O	No	Specify the desired output directory and file name.

A. *Quick guideline*

The simplest command line is shown below, where `NETWORK` and `APP` are the file locations for the network topology and the statistical application model, respectively. `OUTPUT` is the location of the output file. Note that if a different configuration file is used, the workspace directory will be changed as explained in the subsection IV-C.

```
$ ./workspace/generic-3level_msi_mosi_cmp_directory_3level_inclusive/bin/jade.exec
   -w NETWORK --mtsyn APP -O OUTPUT
```

The network file is the first thing users have to decide and it determines the number of cores in the system and the network topology. Several widely-used topologies are provided in the directory `/home/JADE-v5.0/architectures/net`. Users can also create their network files or modify the existing ones to simulate more specific systems.

The second step is to decide which application should be used for simulation. A few example application benchmarks are included in `/home/JADE-v5.0/applications/shalom` for users to test out their simulation environment. To download a full list of application benchmarks, readers are referred to the COSMIC page at https://eexu.home.ece.ust.hk/COSMIC.html. It is worth noting that benchmark files are distinguished by instruction set architectures; check your JADE compilation configuration before picking a right benchmark file. For each application, we profile it with a different number of threads, and a file named `i.txt` means that the statistical application model is obtained by profiling using `i` threads.

Example VI.1. Simulating FFT in a 4x4 mesh topology with 16 threads, assuming a ARM build has been already set up:

```
./workspace/aarch64_msi_mosi_cmp_directory/bin/jade.exec -w ./architectures/net/mesh_4x4.txt
--mtsyn ./applications/shalom/aarch64/splash/fft/16.txt
```

Example VI.2. Simulating H.264 video encoder in a 64-core system with fattree topology:

```
./workspace/aarch64_msi_mosi_cmp_directory/bin/jade.exec -w ./architectures/net/fattree_64.txt
--mtsyn ./applications/shalom/aarch64/parsec/x264/64.txt
```

Using the approaches presented in example VI.1 and example VI.2, the results will be stored at `/home/JADE-v5.0/workspace/aarch64_msi_mosi_cmp_directory/result/results.txt` and it will overwrite any previously obtained result. To set a different location to store results, one should use special arguments when launching JADE with the `-O` option. A brief description of the most commonly used command line arguments is shown in table V. The detailed usage is shown in appendix IX.

VII. Benchmark applications

JADE takes either a realistic application or synthetic traffic as input. The synthetic traffic is based on the Poisson process. The realistic applications are provided by the COSMIC application benchmark suite, which includes applications from some widely-used benchmark suites, including APEX [2], NAS [3], SPEC2006 [4], SPLASH-3 [5] and PARSEC 3.0 [6]. In COSMIC, applications are modeled by the fork/barrier/join model. In this model, the execution flow of the application is partitioned into multiple parallel sections (PSs), and transitions between PSs happen when threads are created, joined, or synchronized.

A. Realistic statistical application model

We represent the fork/barrier/join model using what we refer to as a basic thread block (BTB) graph, as shown in Fig. 3. This example shows four PSs, each of which is composed of one or more BTBs. We define BTB as a sequential execution section that is executed by one physical thread. The edges in the BTB graph represent the flow dependency between each of the BTBs. Single-threaded application will be represented as a single BTB. We profile the application during runtime to obtain a BTB graph as well as the profiling information for each BTB. There are two types of information we need to observe during profiling. One type is function calls to synchronizations (i.e. fork, join, and barrier), and we refer to this type as synchronization trace. The second type is the actual memory traces for instructions and data.

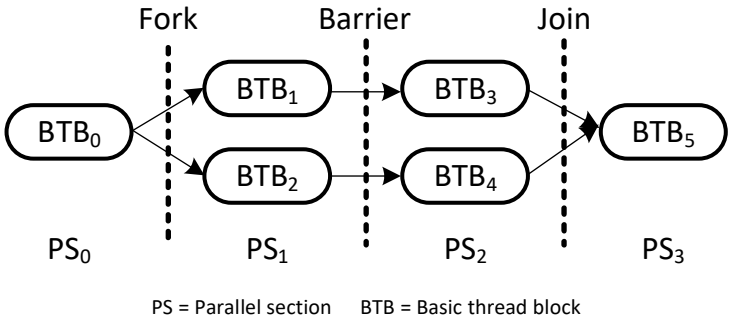


Fig. 3: Application model based on fork/barrier/join.

Applications are profiled using two different instruction sets and processor configurations: `ARM-v8` and `x86-64`. Users should select the appropriate application model for the given components configuration selected during compilation time.

The COSMIC benchmarks are also provided as a stand-alone suite. Please visit https://eexu.home.ece.ust.hk/COSMIC.html for a more detailed description of COSMIC.

B. Directory organization and naming of application models

The application files are located under `/home/JADE-v5.0/applications/shalom`, and profiling results based on x86-64 and arm-v8 are put in the `/x64` and `/aarch64`, respectively. For each application, we profile it with a different number of threads, and a file named `i.txt` means that the statistical application model is obtained by profiling using `i` threads.

Example VII.1. Statistical application model for FFT (in SPLASH-3) with 8 threads for arm-v8

```
Application file: JADE-v5.0/applications/shalom/aarch64/splash/fft/8.txt
```

C. Pipeline stage partitioning

During the simulation, we can partition the application into several stages for pipelined execution. As the system scale increases, the performance improvement brought by thread-level parallelism is vanishing due to high synchronization overhead. The introduction of pipelining provides an additional level of parallelism and can potentially lead to a more efficient use of hardware resources. When evaluating the performance of large-scale systems, pipelined execution could be the preferred simulation mode. The related python scripts are located in `/home/JADE-5.0/utils/shalom_tools/pipeline`.

After partitioning, each pipeline stage consists of one or multiple PSs of the application. During the simulation, the application starts at stage 0 and execute each stage sequentially (0, 1, 2, 3 ...). As soon as one stage finishes we re-start that stage. The execution of one stage can be constrained to a specific machine, a specific chip, or a specific region of the chip.

The algorithm for the application partitioning takes into consideration the computation (i.e., number of instructions) and the communication traffic among stages. It is a simple greedy algorithm that tries to minimize the standard deviation of the load for each of the stages. In other words, the algorithm tries to search for a partitioning that attempts to distribute the load evenly across chips or chip regions while considering the communication traffic.

An example on how to use the mapping tool is shown in VII.2.

Example VII.2. Partition the FFT application profiled with four threads into at most 8 stages.

```
./pipelinePartitioning.py ../../../applications/shalom/aarch64/splash/fft/4.txt
../../../applications/partition/aarch64/splash/fft/4.txt 8
```

In the command above, the statistical application model `shalom/aarch64/splash/fft/4.txt` is the input, `partition/aarch64/splash/fft/4.txt` is the output file, and `8` is the maximum number of stages users would like to have. In this case, we will generate partitions with 2, 4, and 8 stages. If the application has fewer PSs than the given number, we will ignore numbers that are larger than the PS count. A sample output file is given as VII.3.

Example VII.3. An example of the output file when the max number of partitions is set to 8.

```
2 4
4 3 4 5
8 1 2 3 4 5 6 7
```

The first column is always the number of stages. The following numbers define the partitions of the parallel sections. The sequence of numbers represents the index of where we put a delimiter for the partition. For example, the first line has “2 4” meaning that there are two stages. Parallel sections with an index below 4, which are 0, 1, 2, and 3, are mapped to stage 0 while parallel sections with an index equalling or above 4 are mapped to stage 1. Some sample outputs of the pipeline stage partitioning are provided under the directory `/home/JADE-5.0/applications/partition`.

To simulate in this mode, users need to set a few flags in the command line.

- **–pipeline-application**: this flag tells to enable pipelining simulation. You need to pass as an argument the number of stages. Usually, you want to set the number of stages the same as the number of chips.
- **–pipeline-partitioning**: this flag is used to set the partitioning of the application (i.e.: which parallel sections belong to the same pipeline stage). Partitioning for several of the benchmarks is provided. They are located under `/home/JADE-5.0/applications/partition`.
- **–workload-repeat**: this flag indicates how many times you will repeat each stage. Usually, you want to set this number larger than the number of stages you have so that the whole system is exercised.
- **–ignore-pipeline-fill**: this flag indicates how many times you will repeat each stage. Usually, you want to set this number larger than the number of stages you have so that the whole system is exercised.

D. Mapping and scheduling

In contrast to JADE-4.0, mapping and scheduling are implemented at runtime rather than fixed before execution. The scheduling is done hierarchically at a thread level. Whenever a new thread is spawned or resumed, it is firstly scheduled to a chip by the system scheduler. After that, the chip scheduler selects a core and puts the thread into the active task queue of that core. There are three mapping and scheduling algorithms available in JADE:

- Dynamic scheduling: a thread can be scheduled to different chips and cores.
- Static chip dynamic core scheduling: a thread is always executed on the same chip, but it can be assigned to different cores.
- Static chip region and dynamic core scheduling: a thread is always executed on the same chip region, but it can be assigned to different cores.

Dynamic scheduling is the default scheduling algorithm when simulating in multi-threading mode. We define the chip affinity by process ID and the core affinity by thread ID. When making mapping decisions, the thread will be assigned to the affined core if it's available. Otherwise, the scheduler will select a different core. Maintaining the thread to core affinity can help reduce the cache misses.

Static chip and dynamic core scheduling is the default scheduling algorithm during pipelined simulation mode. In this case, one pipeline stage will be mapped to a fixed chip, so the number of chips should be the same as the number of pipeline stages. If users want to apply pipelined execution in a single chip, they can enable the static chip region dynamic core algorithm by specifying the `--static-pipeline-stage-to-region` option in the command line. In this case, the option `--chip-region arg` is also required. If the argument `arg` is a number, cores will be uniformly divided into regions. It can also be the name of an input file, which defines a specific region partition. In this file, each line is a set of core IDs belonging to the same region.

In the case of a multi-machine system, users can assign one scheduler to manage a set of machines. The scheduler will then distribute the workload in this set of machines dynamically. This is useful when you want to run two workloads and you want to reserve a set of machines for one workload, and another set of machines for another workload. This is also useful for the pipeline simulation mode,

where one pipeline stage is treated similarly to one application. To enable multi-chip scheduler, the `--multi-chip-scheduler` should be used. This command option receives a file where each line defines a new scheduler. Each line contains a set of core IDs that this scheduler manages. The sequence of the core is the relative priority of each of the cores.

E. Synthetic traffic

In the simulations based on synthetic traffic, packets are generated as an independent Poisson process. The average generation interval of two sequent packets in one core depends on the offered load. Different types of synthetic traffic patterns determine the destination of each packet.

Under uniform traffic, the packets from each core will be sent to all other cores with the same probability. Under tornado traffic, each packet for core s will be always sent to the core d. The relation between s and d can be expressed by tornado permutation, given as $d = s + (\lceil \frac{N}{2} \rceil - 1)\%N$, where N is the number of cores.

Hotspot traffic patterns and hotpair traffic patterns are similar. A packet generated from core s has the probability of α to be sent to one determined destination core, and has the probability of $1-\alpha$ to be sent to all the other cores with the same probability. The default value of α is 0.5. The difference between hotspot and hotpair is the following: the determined destination core for the packets from different source cores under hotspot traffic is the same, while the determined destination cores for the packets from different source cores under hotpair traffic are different. The relationship between the determined destination core and the source core is defined as tornado permutation.

The rest synthetic traffic patterns are also permutation traffic patterns, where all traffic from each source core is directed to one specific destination core, and the destination is determined by different permutation. For example, the relationship between the destination core and the source core is determined by transpose permutation under transpose traffic pattern.

Here is an example of simulation using synthetic traffic.

Example VII.4. Simulating a 1024-core system with 3-level cache and MOSI protocol based on uniform synthetic traffic. Each core composes a chip. The chips are connected by a 1024-port FODON switch fabric. The packet length is set as 64 Bytes, and it will not change during the simulation. The offered load per core is 0.5, and the simulation time is 5ms:

```
./workspace/generic-3level_msi_mosi_cmp_directory_3level_inclusive/bin/jade.exec
-w ./architectures/net/OSwitchNet_1x1024.txt --max-sim-time 5e9 --bandwidth-mult 23.04 -y -S FODON
-C ./architectures/switch/parameter_setting_FODON.cfg --synthetic_pattern_type Uniform
--pkt_len_type Fixed -R 0.5 --dynamic-mtu 16 -O ./synthetic_results.txt
```

VIII. ARCHITECTURES

A. Network topology file

The network file defines the connection among different components of the system, such as L1, L2, L3, main memory, and router/switch. It should be noticed that in three-level cache architecture we name the first level cache as L0, L1 the second level cache, and L2 the last level cache. For two-level cache hierarchy we name the first level of cache as L1 and the last level of cache as L2. In a three-level cache organization, L0 and L1 caches are private for each core while the last level cache (named L2) is shared for all cores. For two-level cache organization, the first level of cache is L1 and is private. While the last level of cache (L2) is shared among cores.

We use directory-based cache coherence protocols. In the network file, each directory stands for a memory controller and the main memory. We provide various network files for a different number of processors and network topologies, each of them is named based on its topology and number of memories. For instance, `mesh_8x8_8mem.txt` is a 8x8 mesh topology with 8 main memories. Detailed instruction about writing a network file can be found at: [7].

JADE has full support for many-chip simulation. In JADE, each chips have its own private memory space. When simulating multi-chip systems, you can assign each chip a separate memory space and configure the relevant parameters. You can specify the number of chips in your network files.

A short list of network topologies already provided is shown in table VI. For a full list of files, users are encouraged to have look in the directory `/home/JADE-v5.0/architectures/net`. Each network file has a header that briefly describes its topology, number of processors, memory controllers, and number of L2 cache banks and LLC slices. It should be noticed that in multiple chips cases, the total number of processors and processors per chip should be specified accordingly in the header. The processing cores are evenly assigned to each chip based on the numbering of the L1 cache (means a core). Within each chip, the placement of L2/L3 cache bank, directory, and their counts can be decided independently.

TABLE VI: Examples of network files already provided.

Topology	**File name**	**Number Processors**	**Number L2 banks**	**Number Memory Controllers**
Crossbar	crossbar_1	1	1	1
	crossbar_4_1mem	4	4	1
Mesh	mesh_4x4	16	16	16
	mesh_8x8_2L2Cache_2mem	64	2	2
Fattree	fattree_8_2mem	8	8	2
	fattree_16_2mem	16	16	2
Torus	torus_2x2_1mem	4	4	1
	torus_8x16_16mem	128	128	16
Folded-torus	folded-torus_4x4_16mem	16	16	16
	folded-torus_8x8_64mem	64	64	64
Ring	ring_16	16	4	4
	ring_64	64	16	4

B. Custom network topologies

Creating customized topologies with the adjustable placement of components can be accomplished by two methods. The first one is by using the network generator script we provide, located at `/home/JADE-v5.0/scripts/networkGenerator.py`. The current implementation can generate Mesh, Torus, Folded-torus, Crossbar and Fattree. The number of memory controllers can be selected, as well as its placement. The network generator syntax is detailed in table VII. Note that for crossbar and fattree, it should be inserted a number zero right after the number of cores. For other topologies, the number of cores is given by rows $\times$ columns.

TABLE VII: Network generator usage.

Topolgies	**Syntax**					
Mesh, torus, folded-torus	networkGenerator.py	topology	rows	columns	Number of Memories	List of memories location
Crossbar, fattree	networkGenerator.py	topology	cores	0	Number of Memories	List of memories location

The network generator script will print the output file in the `stdout`, therefore it should redirect the standard output to the proper destination file, as shown in examples VIII.1 and VIII.2.

Example VIII.1. To generate a Mesh 4x4 with 1 memory located at node 0, the following command line should be used:

```
./scripts/networkGenerator.py mesh 4 4 1 0 > outputNetwork.txt
```

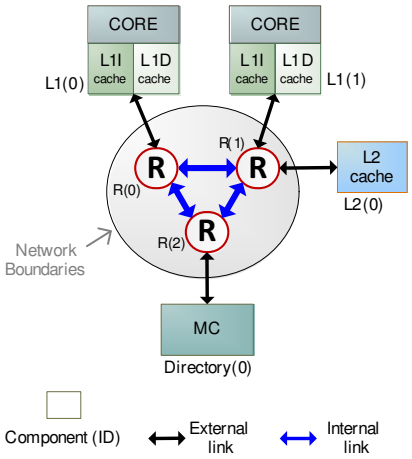
Example VIII.2. To generate a Torus 2x2 with 2 memories located at node 0 and 3, the following command line should be used:

```
./scripts/networkGenerator.py torus 2 2 2 0 3 > outputNetwork.txt
```

In order to obtain more complex and irregular topologies, users can generate a regular topology with the script provided and remove or insert specific elements or nodes. To facilitate the elaboration of huge topologies, the Python API to generate network files can be used and is located at `/home/JADE-v5.0/scripts/networkApi.py`. A network object can be created by calling `createNet`, passing the number of chips, the number of processors per chip, the number of L2 cache banks and the number of memory controllers. This object is printable, so users can use this object to write to a specific file or the `stdout`, respecting python syntax.

Any element that is outside the network is treated as external. Therefore, connecting processors, L2 caches, and directories to the network can be accomplished by using the `external` method, giving as argument the component unique identifier (ID), the router node that it is going to be connected and the link latency. This simple model can incorporate off-chip components by setting the proper link latency. Each node in the network is implemented as a router, and in order to connect routers, the method `link` should be used. Similarly, it should be given the pair of routers to be connected, the latency between them and a weighting for routing decision.

Figure 4 shows a simple example on how to use the `networkApi.py` to generate a network topology. A system composed of two processors, one L2 cache, and one memory controller (and its directory) is



(a) Example of network.

```
// Initialize object for network
net = createNet(1, 2, 1, 1)

// External components placement
net += external('L1Cache', 0, 0, 1)
net += external('L1Cache', 1, 1, 1)
net += external('L2Cache', 0, 1, 1)
net += external('Directory', 0, 2, 1)

// Internal connection routers
net += link(0, 2, 1, 1)
net += link(0, 1, 1, 1)
net += link(1, 2, 1, 1)
```

(b) Example of code to generate network file.

Fig. 4: Example network generated using the Python API provided (`networkApi.py`).

shown in 4a. The code used to generate the system is shown in 4b. The first step is to create the network object, then connect the external elements to routers (nodes) and finally create the desired internal structure by connecting the routers.

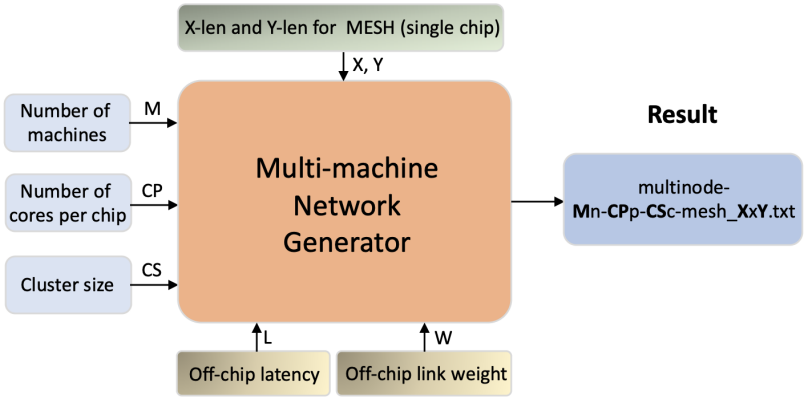


Fig. 5: Multi-node network file generation.

JADE-v5.0 also provides a tool for generating multi-node network files. The tool uses a python script to generate network files to simulate multi-node systems. The script is located under `JADE-v5.0/scripts/multiMachineNetworkGen.py`. Fig. 5 shows the overview of the generator. The script receives the following arguments: number of machines, number of cores per chip, number of cores per cluster, X-length for a mesh (per chip), Y-length for a mesh (per-chip), the off-chip latency, and the off-chip link weight.

C. Optical switch fabrics

JADE 5.0 is capable of analyzing the multi-chip systems where different chips are interconnected by an optical switch. Eight types of switch fabrics have been supported in total. They are Crossbar, DRAGRON, Spiral, PILOSS, Benes, Baseline, FODON and AWGR. Users can also implement other

types of optical switch fabrics, such as Butterfly, Clos, dilated Benes, etc., by inheriting corresponding base classes provided in JADE.

The type of optical switch fabrics is passed by the command line option. For example, if a DRAGON switch fabric needs to be simulated, the command line option `-S DRAGON` should be used. It is worth noting that the port of optical switch fabric is determined by the number of chips, which is specified in the network file (passed by the `-w` option). Other configurations of the optical switch fabric, including the number of WDM channels, the loss parameters of photonic devices, and the parameters of lasers, are specified in the optical switch fabric configuration files, which are located under the directory `JADE-v5.0/architectures/switch`, and users can specify the configuration file through the `-C` option. The optical bandwidth of inter-chip optical links is passed by the `--bandwidth-mult` or the `-B` option.

D. Memory hierarchy and cache coherence protocol

JADE comes with plenty of templates to model a diverse set of memory hierarchies. The templates are located inside the directory *protocols*. You can modify the cache hierarchy by setting the *PROTOCOL* flag during compilation time. See section V for details.

We provide several examples of cache hirearchy with 2-levels and 3-levels. The templates are all based on the SLICC (Specification Language for Implementing Cache Coherence) provided by the Ruby memory simulator [8]. The default cache hierarchy is a 2-level with the directory-based cache coherence MSI-MOSI *MSI_MOSI_CMP_directory*. We also provide templates for inclusive and exclusive 3-levels of caches.

By default, the last level cache is shared across all cores while the other caches are private to each core. The corresponding parameters can be configured in configuration files VIII-F. You can choose to use whatever cache hierarchy and protocols as you like by specifying the protocol file listed under directory *protocols* when compiling JADE.

E. External memory simulation

In JADE, memory behavior is simulated by DRAMSim2 [9]. It is a cycle-accurate model of a DRAM memory controller, the DRAM modules which comprise system storage, and the bus by which they communicate. All major components in a modern memory system are modeled as their respective objects within the source, including ranks, banks, command queue, the memory controller, etc.

The overview of the memory module is shown in Fig. 6. In JADE, the directory module is in charge of the memory issues. It receives read or write requests from caches via the network on chip, and generates corresponding requests to the memory modules (DRAMSim2). Since the main body of JADE is running in event-based mode while DRAMSim2 is running on cycle-based mode, an event to cycle interface is inserted between them. The event generated from directories is put in an input event pool. Every cycle, the interface will check the contents in the input event pool. If the pool is not empty, then the interface would send the corresponding requests to the memory module. On the other hand, the packet from the memory modules is converted into an event by the interface at a specific cycle, and put in an output event pool. There is a clock sequence generator inside the interface.

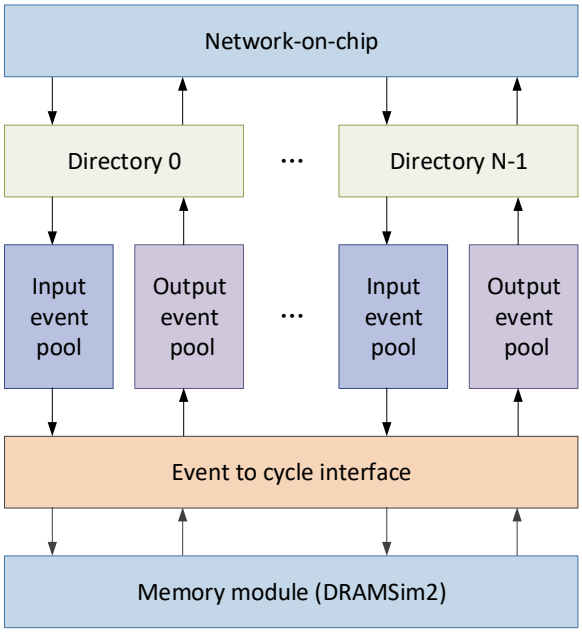


Fig. 6: The interface between directories and DRAMSim2.

The API of the memory modules is shown in Table. VIII, There are three major functions. The addTransaction() is to send a memory request to the memory module. The request includes the type of the transaction (read or write) and the address. The update() is to run the DRAM module by one cycle. The returnReadDataCB() could obtain the content of the read request back from the memory modules, which includes the address of the transaction. The parameters of memory modules is shown in Table. IX. These parameters can be modified in file `JADE-v5.0/dramsim2/ini/myConfig.ini`. Users can also use the predefined parameter file directly in the folder `JADE-v5.0/dramsim2/ini/`.

TABLE VIII: The APIs of memory modules.

API function	Description
addTransaction()	Send a memory transaction request to the memory controller.
Update()	Run DRAM by one cycle.
returnReadDataCB()	Data read from the DRAM.

TABLE IX: The parameters of memory modules.

Category	Parameter	Description
Configuration	NUM_BANKS	Number of banks
	NUM_ROWS	Number of rows
	NUM_COLS	Number of columns
	DEVICE_WIDTH	width of the data port
	REFRESH_PERIOD	Refresh period in ns
Timing	tCK, CL, AL, BL, tRAS, tRCD, tRRD, TRC, tRP, tCCD, tRTP, tWTR, tWR, tRTRS, tRFC, tFAW, tCKE, tXP, tCMD	All kinds of time parameters related to the DIMM
Power consumption	IDD0, IDD1, IDD2P, IDD2Q, IDD2N, IDD3Pf, IDD3Ps, IDD3N, IDD4W, IDD4R, IDD5, IDD6, IDD6L, IDD7	All kinds of current parameters related to the DIMM
	Vdd	Supply voltage

F. Components configuration file

Many components of JADE can be individually tuned. For instance, one can set buffer sizes of NoC routers, link latencies, size of memory and caches, latency to access cache, and many other parameters. A few configuration files are already provided, located in `/home/JADE-v5.0/architectures/configs`. Although many of the parameters are explained inside the configuration files, we provide in Table X a more detailed description of some selected configuration items, since they are important for simulation. When the configurations in the configuration file are changed, it is necessary to remake JADE to make these configurations effective. For easy use, we also provide command line setting for some frequently used parameters, as listed in the appendix. It will take effect without remaking JADE but complicating the command line. It is up to users to choose the way they like.

TABLE X: Selected configuration items

Parameter	Description	Example
g_RUNNING_CYCLES	Specify number of cycles to finish the simulation.	0 means run the simulation to the end.
g_INTERLEAVED_MEMORY	The way an address mapped to memory controllers.	True means interleaved mapping, False means continuous mapping.
g_simple_NoC	Specify the NoC granularity.	True means simplified NoC, False means detailed NoC.
g_JADE_E_NETWORK	Specify if electrical architecture used.	True, electrical network is used.
g_MEMORY_SIZE_BYTES	Main memory size in bytes for each memory.	For 4GB memories set 4294967296.
g_DATA_BLOCK_BYTES	Data block size in bytes. Corresponds to cache line sizes in all levels.	For cache line size of 64, set 64.
L1_CACHE_NUM_SETS_BITS	Number of cache sets in bits per core.	To obtain L1 cache of size 64Kbytes, in which data block is 64 bytes, and associtivity is 4, you need to set this variable to 8. $2^8 * 4 * 64 = 64KB$
L2_CACHE_NUM_SETS_BITS	Number of cache sets in bits per core.	For example, to set a total size of L2 of 2MBytes, with data block 64B, associtivity 8, and and assuming there are 4 processors, you need to set this variable to 10. $2^{10} * 8 * 64 * 4 = 2MB$

Users should select the appropriate application model file in accordance with the components configuration used to compile JADE. For instance, in case `SYSTEM_SETTINGS=aarch64` were used to build JADE, users should use application files under the directory `aarch64`. Please refer to [10] for more descriptions of memory parameters. We further provide a generic configuration parameter based on a generic MIPS implementation, to allow users to modify and tune each parameter individually.

G. Power models

In JADE-v5.0, we incorporate holistic Power Management (PM) function for managing the system power and generating power statics of different components. The PM includes three subsystems: Dynamic Voltage and Frequency Scaling(DVFS), Energy Consumption Calculation and Power Delivery System(PDS)

Models. The central PM controller in JADE is called PMU (Power Management Unit), which is defined under the directory *support*. PMU is responsible for the following functions:

- Initializing Power Management(PM) States, which represents the voltage and frequency pairs
- Launching Power Tools and get corresponding power consumption results for target processor configurations and defined PM states
- Maintain and update real-time PM states for different components (DVFS)
- Calculate energy consumptions for different components (Core / L2)
- Create Power Delivery System Model
- Print power results at the end of the simulation
- Generating per-core frequency/dynamic power/static power variation based on process variation

The PMU initialization workflow of the JADE simulation is as shown in Figure 7. Firstly, it defines the frequencies and voltage pairs for different PMStates in VFLists for different components. Secondly, it launches the Power Tool (Currently McPAT) in parallel to compute the dynamic and static power for all the PMStates of different components. The temporary configuration file is constructed. Then the power tool is launched directly by ./utils/mcpat/bin/powerWizard.py using the above config file. Finally, the computation of power tools, corresponding results are written in Power Vectors. These results could be further used for DVFS and energy consumption calculation. For all components, including processor cores, routers, caches, etc., both dynamic and static energy is calculated.

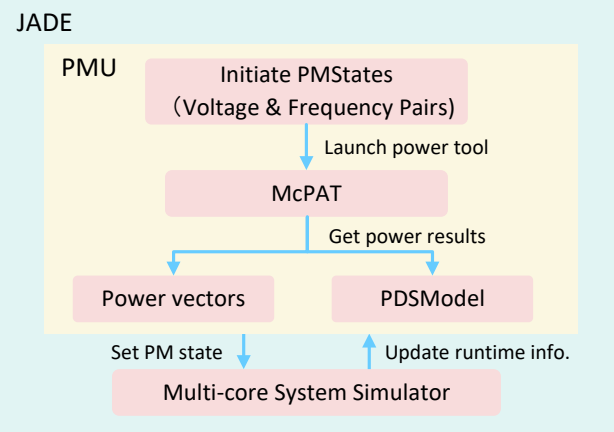


Fig. 7: The PMU workflow in JADE.

Currently, the DVFS function supports the following components, and each component can have its own voltage and frequency settings. Processor Cores: PM state is changed by the function SetProcPMState(). Users can set PMState at different temporal granularities. Voltage/Frequency Scaling Time (VScalingTime) is considered if the PDSModel is enabled. It is generated by PDSModel. PMU also maintains the working states for each core (idle or running) for PDSModel Energy Calculation. We use PMU::UpdateProcWorkingState(Processor* proc) to update the working state and compute the PDS efficiency at the same time.

Network Interfaces: Use PMU::SetPMState_NI(w_NetworkInterface* NI) to change PMstate of NI in its wakeup() function. Routers: Use PMU::SetPMState_Router(w_router* router) to change PMstate of router in its wakeup() function. By default, the DVFS for processor, router, and network interface are

all disabled. If you want to use this feature, you can set “Flag_DVFS_proc”, “Flag_DVFS_router”, “ Flag_DVFS_NI” to be true in `/home/JADE-v5.0/support/PMU.h`.

JADE also includes a process variation generator to simulate the variation of core frequencies, average dynamic power, and average static power. The fundamental variations of two device-level parameters that cause the above variations are the effective gate length L_{eff} and the threshold voltage V_{th}. We model L_{eff} and V_{th} as a spatial correlated normal distribution, and use a spherical function to generate covariance between cores.

IX. Output items

Several statistics are provided by JADE including holistic performance and power analysis, system behavior, and memory access trace. For a given input application and architecture, several simulation iterations are executed allowing the application behavior to be randomly constructed. From the simulation we use a configurable amount of intermediate results, ignoring initial and final iterations to allow system warm-up and sink.

Displayed outputs include overall application execution time, individual processor statistics such as total busy time, and waiting time to depict the utilization ratio of processors. The network counters shown include packet injection rate for individual routers, average packet delay, throughput, and flow control statistics. JADE also incorporates power library models, for selected technology nodes, and reports power and energy usage for various components within the system.

Memory access trace is also recorded, supplementing further investigation of memory behavior of applications. Besides conventional outputs, researchers and designers need to get more detailed behavior of some specific component in the system to better understand it and lately tune. To attend different needs, a custom output is also possible by adding customized recorders to the modules, enabling extraction of specific metrics and behavior.

Although we intend to print out the statistics of system behavior and performance as detailed as possible, we still understand that users might be interested in certain perspectives that we have not covered in the current version. To this end, users may need to go through the source codes and add probes to realize their requirements.

AGREEMENT AND LICENSE

JADE copyrights can be found in its root directory. If you use JADE in your research, please cite the following paper (http://dx.doi.org/10.1145/2857058.2857066):

Rafael K. V. Maeda, Peng Yang, Xiaowen Wu, Zhe Wang, Jiang Xu, Zhehui Wang, Haoran Li, Luan H. K. Duong, Zhifei Wang, *JADE: a Heterogeneous Multiprocessor System Simulation Platform Using Recorded and Statistical Application Models*, HiPEAC Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems, Prague, January 2016.

REVISION HISTORY

TABLE XI: Revision history

Version	Changes	Date
2.1	Cache coherence protocols	1-Jun-16
2.2	Inter-chip communication and separate memory space	10-Oct-16
2.3	Optimized memory usage and compatible with APEX benchmarks	10-Dec-16
2.4	Power management unit	2-Feb-17
2.5	Process variation	1-Apr-17
2.6	Internal test and verification	1-May-17
3.0	Public release	1-Aug-17
3.1	Adoption of COSMIC v3.0	1-Jul-18
3.2	Enhanced CPU models.	1-Aug-18
3.3	Adoption of more cache coherence protocols.	1-Aug-18
4.0	Public release	1-Sept-18
5.0	Public release	1-May-20

REFERENCES

[1] R. K. V. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian, "Fast and accurate exploration of multi-level caches using hierarchical reuse distance," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 145–156.

[2] "APEX benchmarks," https://www.lanl.gov/projects/apex, accessed 28-Feb-2020.

[3] "NAS parallel benchmarks," https://www.nas.nasa.gov/publications/npb.html, accessed 28-Feb-2020.

[4] "SPEC2006 benchmarks," https://www.spec.org/cpu2006, accessed 28-Feb-2020.

[5] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 101–111.

[6] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[7] http://research.cs.wisc.edu/gems/doc/gems-wiki/moin.cgi/Understanding%20Network%20Files.

[8] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[9] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[10] http://research.cs.wisc.edu/gems/doc/gems-wiki/moin.cgi/.

APPENDIX-A: BASIC COMMAND LINE OPTIONS

There are two classes of command line. The basic options and the advanced options. This section shows the basic options whereas in Appendix IX we show the advanced options. To obtain the full basic command line options you can run the command:

```
./workspace/aarch64_msi_mosi_cmp_directory/bin/jade.exec --help
```

The basic command line option is shown below:

```
JADE command line options:

Usage options:
  -h [ --help ]                 List command line options.
  -H [ --help-arch ]            List architecture parameters options.
  -v [ --version ]              Print JADE version

Basic options (all required):
  -w [ --net ] arg              Topology file (.net). Examples under directory:
                                JADE/architectures/net.
  -O [ --out ] arg              Results output file. Default file:
                                workspace/dir/result/results.txt
  --mtsyn arg                   Use synthetic multithreaded workload

Logging options, use --log-help for details.:
  --log arg                     Enable prints for debugging for a particular
                                subset of messages. Example --log="network,cpu"
  --log-help                    Will print all possible logs and a comment
                                about their purposes.

Simulation control:
  -P [ --periodic-stats ] arg   Print results periodically. Should pass the
                                period to print. Unit in cycles.
  --max-sim-time arg            Maximum simulation time. Given in time unit
                                (ps). Double value accepted. e.g: 10e3 will
                                simulate 10000ps.
  -r [ --random ] arg           Random seed. Use 0 for time() and larger than 0
                                for fixed seed.
  -N [ --iterations ] arg       How many iterations the same application should
                                be executed. The results will be accumulated.
  --workload-repeat arg         Number of times to run the same application.
  -D [ --trace ] arg            Trace file when running cycle-accurate.
                                Examples under JADE/applications/app. You still
                                need to input the (.sam) file using --app.
  -y [ --cached-net ]           Use cached routing information to build the
                                network. Saves simulation time.
  -Y [ --gen-net ]              Do not simulate, only generate the cached
                                routing information.
```

```
  --limit-task-iteration arg    Limit the total number of task to simulate per
                                iteration. If 0 this is disabled..
  --nosharing                   Do not simulate the sharing between thread
                                blocks.
  --omit-net-traffic            Will omit results for SRC-DST message size
                                traffics.
  --trace-reduction arg         Will simulate with details 1 / X, where X is
                                the value set. If X<0 it will use automatic
                                sampling with at least X detailed simulation
                                length.
  --sampling-rate arg           Same as trace-reduction
  --max-sampling-rate arg       default (1000). Maximum sampling rate in
                                automatic mode.
  --conv-factor arg             Defult 0.5. It means that the convergence can
                                be 50% error. Need to set sampling-rate -1.
  --sampling-warm-length arg    Only useful for cycle-accurate.
  --no-func-warmup              Will NOT do functional warmup during sampling.
  --prof-tnumber arg            Number of threads for profiling. Value (0) will
                                disable it.
  --print-results-per-phase     Will print the results every time a new phase
                                is reached (.e.g: new thread is created)
  --ignore-init-phase           Will not simulate the initial phase of the
                                application.
  --ignore-pipeline-fill        Will not simulate the initial phase when the
                                pipelining is filling / draining.Only
                                applicable when simulating pipelined
                                application.
  --ideal-data-cache            Will simulate as if all access to DATA cache is
                                a hit. Allow quick simulation of the inter-chip
                                net.
  --ideal-inst-cache            Will simulate as if all access to INSTRUCTION
                                cache is a hit. Allow quick simulation of the
                                inter-chip net.

Mapping and scheduling options:
  --static-chip-mapping              Use static chip mapping and dynamic
                                     mapping for each core scheduling (under
                                     development).
  --pipelined-application arg        Set application running into pipeline.
                                     Should give number of stages.
  --pipeline-partitioning arg        Application partitioning telling which PS
                                     belongs to which stage.
  --static-pipeline-stage-to-region  Will map pipeline stage 0 to chip region
                                     0. Only works in pipeline mode, and using
                                     region. See chip-region option for more
                                     details.
```

```
  --ps-per-chip arg                Set the number of parallel sections per
                                   chip. How many parallel sections to run in
                                   a chip before trying to find a new chip.
  --chip-region arg                Create regions withing a single chip. All
                                   chips will have the same regions. This
                                   flag can receive an integer or a file
                                   name. In the case of an integer, N, then
                                   we will divide the chip into N regions.
                                   For example, if you simulate a 8-core chip
                                   and use N=2 we will set one region for
                                   cores 0 to 3 and another region for cores
                                   from 4 to 7. In the case of a file name,
                                   then we will read the file and each line
                                   of the file will have the list of cores
                                   for that region.
  --multi-chip-scheduler arg       Create custom schedulers that manipulate
                                   multiple chips. The chips could belong to
                                   the same machine or not. The scheduler
                                   will assign workload to these
                                   chips/machines assuming there is a equal
                                   cost to assign an workload to either of
                                   the machine. This argument receives a file
                                   where each line define a new scheduler.
                                   Each line contain a set of coreid that
                                   this scheduler manages. The sequence of
                                   the core is the relative priority of each
                                   of the cores.

Basic system parameters.:
  --base-period-ps arg             Baseline period (in picoseconds) of all components.
                                   Default: 500ps = 2GHz.
  --comp-param arg                 Set the value of a particular component parameter.
                                   See comp-param-list to list all possible parameters.
                                   This option expects the name of the component,
                                   the id of the component, and the value
                                   (all in a single string). For example, to set the
                                   issue width of a CPU to 8 use the following,
                                   --comp-param "TIpcCpu 3 maxMemIssuePerCycle 8".
                                   You can use this argument as many time as needed.
  --comp-param-list                List all parameters of components that are
                                   configurable. This is per-component setting. For
                                   system-wide settings, see --help-arch.

Optical inter-chip switch options:
  -S [ --OSwitch_Type ] arg        OSwitch Type. Examples: Fat_Tree, FODON, Benes
  -C [ --OSwitch_Filename ] arg    OSwitch Configure file(.cfg). Example under
```

```
                                  directory: JADE/architectures/config.

Synthetic traffic options:
  -R [ --injection-rate ] arg     Injection rate for the synthetic-traffic.
  -T [ --dynamic-mtu ] arg        Dynamic MTU for inter-chip communication.
  -B [ --bandwidth-mult ] arg     Inter-chip interconnect bw multiplier
  -E [ --electrical-icn ]         Use electrical inter-chip network
  -F [ --performance-ratio ] arg  Set the processor performance ratio
  -M [ --integrated-nic ]         Integrated NIC
  --IC_WORD_SIZE arg              Set the size of flit (basic unit) for
                                  inter-chip packets, which is the word size in
                                  the default mode
  --synthetic_pattern_type arg    Synthetic pattern type: Uniform, Tornado,
                                  Hotspot, HotPair, ...
  --pkt_len_type arg              Distribution of packet length under synthetic
                                  pattern type: Fixed, Exponential, ...

Power parameters:
  -L [ --power-lib ] arg          Set the power library file.
  -A [ --tech-node ] arg          Set the technology node (nm).
  --enable-power-tool             Run the power tool (McPat) to estimate energy
                                  and power values.
  --enable-pds                    Activate PDS simulation, requires user to set
                                  the output file (pds-outfile)
  --pds-result-path arg           Set the path to write the PDS results. Default:
                                  workspace/dir/result/pds/*
  --pds-baseline-iterations arg   Baseline iterations to compute average pds stats.
  --pds-iterations-average arg    Number of iterations to compute average pds stats.
```

APPENDIX-B: ADVANCED COMMAND LINE OPTIONS

The advanced command line options sets specific architecture parameters such as cache size, cache latencies, and many more. You are advised to use these options carefully since misconfiguration of the parameters might cause unpredictable behavior in the simulation. You can obtain the full advanced options by running JADE with the command:

```
./workspace/aarch64_msi_mosi_cmp_directory/bin/jade.exec --help-arch
```

Here is the full command line option shown:

```
Advanced architecture parameters. Use carefully:

System parameters.:
  --g_NUM_MEMORIES arg                  (uint)
  --g_PROCS_PER_CHIP arg                (uint)
  --g_NUM_CHIPS arg                     (uint)
  --g_NUM_CHIP_BITS arg                 (uint)
  --g_NUM_PROCESSORS_BITS arg           (uint)
  --g_PROCS_PER_CHIP_BITS arg           (uint)

Network parameters:
  --g_adaptive_routing arg              (boolean)
  --g_PRINT_TOPOLOGY arg                (boolean)
  --g_JADE_E_NETWORK arg                (boolean)
  --g_JADE_SUOR arg                     (boolean)
  --g_INIC arg                          (boolean)
  --g_simple_NoC arg                    (boolean)
  --g_DETAIL_NETWORK arg                (boolean)
  --g_GARNET_NETWORK arg                (boolean)
  --g_NETWORK_TESTING arg               (boolean)
  --g_bash_bandwidth_adaptive_threshold arg
                                        (float)
  --g_ODB arg                           (float)
  --InjectRate arg                      (float)
  --g_topology_file arg                 (string)
  --g_NETWORK_TOPOLOGY arg              (string)
  --g_DP arg                            (uint)
  --g_PF arg                            (uint)
  --g_D_MTU arg                         (uint)
  --NETWORK_LINK_LATENCY arg            (uint)
  --COPY_HEAD_LATENCY arg               (uint)
  --ON_CHIP_LINK_LATENCY arg            (uint)
  --g_endpoint_bandwidth arg            (uint)
  --NUMBER_OF_VIRTUAL_NETWORKS arg      (uint)
  --g_SUOR_cluster_size arg             (uint)
  --g_FLIT_SIZE arg                     (uint)
  --g_NUM_PIPE_STAGES arg               (uint)
  --g_VCS_PER_CLASS arg                 (uint)
  --g_BUFFER_SIZE arg                   (uint)
  --g_SpecifiedGenerator arg            (string)

Processor parameters.:
  --g_SYNTHETIC_DRIVER arg              (boolean)
  --g_DETERMINISTIC_DRIVER arg          (boolean)
  --PROFILE_HOT_LINES arg               (boolean)
  --PROFILE_ALL_INSTRUCTIONS arg        (boolean)
  --USER_MODE_DATA_ONLY arg             (boolean)
  --TRANSACTION_TRACE_ENABLED arg       (boolean)
  --g_SIMICS arg                        (boolean)
  --FINITE_BUFFERING arg                (boolean)
  --g_trace_warmup_length arg           (uint)
  --g_tester_length arg                 (uint)
  --ENABLE_DVFS arg                     (boolean)
  --g_synthetic_locks arg               (uint)
  --g_deterministic_addrs arg           (uint)
  --g_callback_counter arg              (uint)
  --g_NUM_SMT_THREADS arg               (uint)
  --SIMICS_RUBY_MULTIPLIER arg          (uint)
  --OPAL_RUBY_MULTIPLIER arg            (uint)
  --g_NUM_PROCESSORS arg                (uint)
  --PROCESSOR_BUFFER_SIZE arg           (uint)
  --SEQUENCER_TO_CONTROLLER_LATENCY arg (uint)

Memory parameters:
  --PERFECT_MEMORY_SYSTEM_LATENCY arg   (uint)
  --L0_CACHE_ASSOC arg                  (uint)
  --L0_CACHE_NUM_SETS_BITS arg          (uint)
  --L1_CACHE_ASSOC arg                  (uint)
  --L1_CACHE_NUM_SETS_BITS arg          (uint)
  --L2_CACHE_ASSOC arg                  (uint)
  --L2_CACHE_NUM_SETS_BITS arg          (uint)
  --g_DATA_BLOCK_BYTES arg              (uint)
  --g_PAGE_SIZE_BYTES arg               (uint)
  --g_NUM_L2_BANKS arg                  (uint)
  --g_MEMORY_SIZE_BITS arg              (uint)
  --g_DATA_BLOCK_BITS arg               (uint)
  --g_PAGE_SIZE_BITS arg                (uint)
  --g_NUM_L2_BANKS_BITS arg             (uint)
  --g_NUM_L2_BANKS_PER_CHIP_BITS arg    (uint)
  --g_NUM_L2_BANKS_PER_CHIP arg         (uint)
  --g_NUM_MEMORIES_BITS arg             (uint)
  --g_NUM_MEMORIES_PER_CHIP arg         (uint)
  --g_MEMORY_MODULE_BITS arg            (uint)
  --DIRECTORY_CACHE_LATENCY arg         (uint)
  --ISSUE_LATENCY arg                   (uint)
  --CACHE_RESPONSE_LATENCY arg          (uint)
  --L2_RESPONSE_LATENCY arg             (uint)
  --L2_TAG_LATENCY arg                  (uint)
  --L1_RESPONSE_LATENCY arg             (uint)
  --L0_RESPONSE_LATENCY arg             (uint)
  --MEMORY_RESPONSE_LATENCY_MINUS_2 arg (uint)
  --DIRECTORY_LATENCY arg               (uint)
  --RECYCLE_LATENCY arg                 (uint)
  --L2_RECYCLE_LATENCY arg              (uint)
  --TBE_RESPONSE_LATENCY arg            (uint)
  --L0_REQUEST_LATENCY arg              (uint)
```

```
  --L1_REQUEST_LATENCY arg              (uint)
  --L2_REQUEST_LATENCY arg              (uint)
  --L0CACHE_TRANSITIONS_PER_RUBY_CYCLE arg
                                        (uint)
  --L1CACHE_TRANSITIONS_PER_RUBY_CYCLE arg
                                        (uint)
  --L2CACHE_TRANSITIONS_PER_RUBY_CYCLE arg
                                        (uint)
  --DIRECTORY_TRANSITIONS_PER_RUBY_CYCLE arg
                                        (uint)
  --NUMBER_OF_TBES arg                  (uint)
  --NUMBER_OF_L0_TBES arg               (uint)
  --NUMBER_OF_L1_TBES arg               (uint)
  --NUMBER_OF_L2_TBES arg               (uint)
  --PROTOCOL_BUFFER_SIZE arg            (uint)
  --MEM_BUS_CYCLE_MULTIPLIER arg        (uint)
  --BANKS_PER_RANK arg                  (uint)
  --RANKS_PER_DIMM arg                  (uint)
  --DIMMS_PER_CHANNEL arg               (uint)
  --BANK_BIT_0 arg                      (uint)
  --RANK_BIT_0 arg                      (uint)
  --DIMM_BIT_0 arg                      (uint)
  --BANK_QUEUE_SIZE arg                 (uint)
  --BANK_BUSY_TIME arg                  (uint)
  --RANK_RANK_DELAY arg                 (uint)
  --READ_WRITE_DELAY arg                (uint)
  --MEM_CTL_LATENCY arg                 (uint)
  --REFRESH_PERIOD arg                  (uint)
  --BASIC_BUS_BUSY_TIME arg             (uint)
  --TFAW arg                            (uint)
  --MEM_RANDOM_ARBITRATE arg            (uint)
  --MEM_FIXED_DELAY arg                 (uint)
  --PRINT_INSTRUCTION_TRACE arg         (boolean)
  --REMOVE_SINGLE_CYCLE_DCACHE_FAST_PATH arg
                                        (boolean)
  --g_INTERLEAVED_MEMORY arg            (boolean)
  --SyntheticTraffic arg                (boolean)
  --MAP_L2BANKS_TO_LOWEST_BITS arg      (boolean)
  --g_MEMORY_SIZE_BYTES arg             (ulong)
  --g_MEMORY_MODULE_BLOCKS arg          (ulong)
  --XACT_CONFLICT_RES arg               (string)
  --PERFECT_MEMORY_SYSTEM arg           (boolean)

Instrumentation parameters.:
  --RANDOMIZATION arg                   (boolean)
  --g_FILTERING_ENABLED arg             (boolean)
  --PROTOCOL_DEBUG_TRACE arg            (boolean)
  --PERIODIC_TIMER_WAKEUPS arg          (boolean)
  --PERFECT_FILTER arg                  (boolean)
  --PERFECT_VIRTUAL_FILTER arg          (boolean)
  --PERFECT_SUMMARY_FILTER arg          (boolean)
  --ENABLE_MAGIC_WAITING arg            (boolean)
  --ENABLE_WATCHPOINT arg               (boolean)
  --DEBUG_FILTER_STRING arg             (string)
  --DEBUG_VERBOSITY_STRING arg          (string)
  --DEBUG_OUTPUT_FILENAME arg           (string)
  --g_RANDOM_SEED arg                   (uint)
  --g_DEADLOCK_THRESHOLD arg            (uint)
  --g_RETRY_THRESHOLD arg               (uint)
  --g_FIXED_TIMEOUT_LATENCY arg         (uint)
  --g_NUM_COMPLETIONS_BEFORE_PASS arg   (uint)
  --g_think_time arg                    (uint)
  --g_hold_time arg                     (uint)
  --g_wait_time arg                     (uint)
  --g_DEBUG_CYCLE arg                   (uint)
  --NULL_LATENCY arg                    (uint)
  --TIMER_LATENCY arg                   (uint)
  --ABORT_RETRY_TIME arg                (uint)
  --DEBUG_START_TIME arg                (ulong)

Legacy parameters.:
  --g_DISTRIBUTED_PERSISTENT_ENABLED arg
                                        (boolean)
  --g_DYNAMIC_TIMEOUT_ENABLED arg       (boolean)
  --XACT_ENABLE_VIRTUALIZATION_LOGTM_SE arg
                                        (boolean)
  --XACT_EAGER_CD arg                   (boolean)
  --XACT_LAZY_VM arg                    (boolean)
  --XACT_VISUALIZER arg                 (boolean)
  --XACT_NO_BACKOFF arg                 (boolean)
  --g_REPLACEMENT_POLICY arg            (string)
  --BLOCK_STC arg                       (boolean)
  --PROFILE_EXCEPTIONS arg              (boolean)
  --ATMTP_ENABLED arg                   (boolean)
  --ATMTP_ABORT_ON_NON_XACT_INST arg    (boolean)
  --ATMTP_ALLOW_SAVE_RESTORE_IN_XACT arg
                                        (boolean)
  --READ_WRITE_FILTER arg               (string)
  --VIRTUAL_READ_WRITE_FILTER arg       (string)
  --SUMMARY_READ_WRITE_FILTER arg       (string)
  --g_CACHE_DESIGN arg                  (string)
  --ATMTP_XACT_MAX_STORES arg           (uint)
  --ATMTP_DEBUG_LEVEL arg               (uint)
  --FAN_OUT_DEGREE arg                  (uint)
  --XACT_DEBUG_LEVEL arg                (uint)
  --XACT_NUM_CURRENT arg                (uint)
  --XACT_LAST_UPDATE arg                (uint)
  --XACT_COMMIT_TOKEN_LATENCY arg       (uint)
  --XACT_LOG_BUFFER_SIZE arg            (uint)
  --XACT_STORE_PREDICTOR_HISTORY arg    (uint)
  --XACT_STORE_PREDICTOR_ENTRIES arg    (uint)
  --XACT_STORE_PREDICTOR_THRESHOLD arg  (uint)
  --XACT_FIRST_ACCESS_COST arg          (uint)
  --XACT_FIRST_PAGE_ACCESS_COST arg     (uint)
  --XACT_LENGTH arg                     (uint)
  --XACT_SIZE arg                       (uint)
  --PROFILE_XACT arg                    (boolean)
  --PROFILE_NONXACT arg                 (boolean)
  --XACT_DEBUG arg                      (boolean)
  --XACT_MEMORY arg                     (boolean)
  --XACT_ENABLE_TOURMALINE arg          (boolean)
  --XACT_ISOLATION_CHECK arg            (boolean)
```

APPENDIX-C: LIST OF APPLICATIONS IN COSMIC BENCHMARK SUITE

COSMIC includes statistical application models that are obtained from profiling applications from multiple widely used benchmark suites, including APEX [2], NAS [3], SPEC2006 [4], SPLASH-3 [5] and PARSEC 3.0 [6], for a target processor. The brief description of each application benchmark is listed in table XII. Please check out the full list on the COSMIC release page at https://eexu.home.ece.ust.hk/COSMIC.html.

TABLE XII: List of applications in COSMIC benchmark suite

Suite	Application	Description
APEX	hpcg	Conjugate gradient algorithm.
	dgemm	Dense matrix multiplication.
	pennant	Lagrangian staggered-grid hydrodynamics algorithm on 2-D unstructured finite-volume mesh.
	stream	Synthetic benchmark measuring the memory bandwidth and a corresponding computation rate for four simple vector kernels.
NAS	bt	Solving a synthetic system of partial differential equation using block Tri-diagonal solver.
	cg	Estimating the smallest eigenvalue of a large sparse symmetric positive-definite matrix with the conjugate gradient method.
	dc	Data cube operator.
	ep	Generating independent Gaussian random variates using the Marsaglia polar method.
	ft	Solving a 3D partial differential equation using the fast Fourier transform (FFT).
	is	Sorting small integers using the bucket sort.
	lu	Solving a synthetic system of partial differential equation using lower-upper Gauss-Seidel solver.
	mg	Approximating the solution to a 3D discrete Poisson equation using the V-cycle multigrid method.
	sp	Solving a synthetic system of partial differential equation using scalar Penta-diagonal solver.
	ua	Solving heat equation with convection and diffusion from moving ball.
PARSEC 3.0	blackscholes	Option pricing with Black-Scholes partial differential equation.

Continued on next page

TABLE XII – continued from previous page

Suite	Application	Description
	canneal	Cache-aware simulated annealing to optimize routing cost of a chip design.
	dedup	Compression with data de-duplication.
	facesim	Simulating the motions of a human face.
	ferret	Content-based similarity search.
	fluidanimate	Simulating fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method.
	freqmine	Frequent itemset mining.
	streamcluster	Online clustering of an input stream.
	swaptions	Pricing of a portfolio of swaptions using the Heath-Jarrow-Morton (HJM) framework.
	x264	H.264 video encoder.
SPEC2006	bzip2	File compression and decompression based on bzip2.
	cactusADM	Solving Einstein evolution equations with staggered-leapfrog method.
	calculix	Finite element method for 3D structural analysis applications.
	dealII	Adaptive finite element method for solving partial differential equations.
	gobmk	Analysing and playing Go game.
	hmmer	Profile hidden Markov models for protein sequence analysis.
	lbm	Lattic Boltzmann method (LBM) for simulating 3D fluids dynamics.
	libquantum	Quantum computer simulation.
	mcf	Single-depot vehicle scheduling.
	namd	Simulating large biomolecular systems.
	omnetpp	Discrete event simulation of a large Ethernet network.
	povray	Ray tracing, rendering algorithm.
	sjeng	Playing chess.
	soplex	Simplex linear program solver.
	specrand	Generating pseudorandom numbers.
	specrand_i	Generating pseudorandom integer numbers.
	sphinx	Speech recognition.
	xalancbmk	Transforming XML documents to other document types.

Continued on next page

TABLE XII – continued from previous page

Suite	Application	Description
SPLASH-3	barnes	Simulating N-body problem with Barnes-Hut method.
	cholesky	Blocked Cholesky factorization on a sparse matrix.
	fft	Fast Fourier Transform.
	fmm	Simulating N-body problem with Fast Multipole Method.
	lu	Factorizing a dense matrix into the product of a lower triangular and an upper triangular matrix.
	ocean	Simulating large-scale ocean movements.
	radix	Integer radix sort.
	raytracing	Rendering a 3D scene onto a 2D image plane with ray tracing.
	volrend	Rendering a 3D volume onto a 2D image plane using an optimized ray casting technique developed by Marc Levoy.
	water-nsquared	Simulating the molecular dynamics N-body problem.
	water-spatial	Simulating the molecular dynamics N-body problem.