SMC Simulation Environment (Scripts and Utilities)

Erfan Azarkhish (erfan.azarkhish@unibo.it)

October 12, 2016

Partners: Davide Rossi (davide.rossi@unibo.it)
Igor Loi (igor.loi@unibo.it)
Supervisor: Professor Luca Benini (luca.benini@iis.ee.ethz.ch)

This document provides description for the utilities and scriptes provided in the SMC Simulation environment.

Contents

1 The Main Simulation Script: (smc.sh)

This is the main script for running the SMC simulation environment. It can be either executed directly, or called from the simulation scripts available in *scenarios/*. Use *./smc.sh -h* to see a list of the available options. This script wraps all executed tools and passes the parameters defined in *UTILS/default_params.sh* to them.

2 Common Utilities: (common_utils.sh)

Common utilities shared between all scripts including gem5 and ModelSim wrappers are defined here. The relative path to this script is UTILS/common_utils.sh.

2.1 General Functions

2.1.1 export_array()

By default, bash does not allow exporting arrays. This function provides this facility by exporting all elements of the array. The array must be first stored in a temporary variable such as P, then the export function should be called specifying the name of the array and the variable P:

```
P=(V0 V1 V2);
```

```
export_array MY_ARRAY P
```

After execution of this command, the following environment variables will be exported:

```
MY_ARRAY_0=V0
```

```
MY_ARRAY_1=V1
```

```
MY_ARRAY_2=V2
```

2.1.2 print_xx()

The available printing functions are: print_colored, print_msg, print_err, print_war.

2.1.3 clonedir()

Clone an entire directory to the SCENARIO_CASE_DIR then change current directory to it. This is to ensure that all intermediate files are generated inside the scenario directory associated with the scenario script, and that the source directories remain untouched and clean. After calling this function, at some point user can call returntopwd() to return to the previous directory.

2.1.4 zpad()

This function adds zero padding before numbers to create clean and sorted scenario-case names. This is used to create statistics in sorted numerical order. Please refer to one of the scenario scripts to see its usage.

2.1.5 param_xx()

These set of functions export bash parameters to different environments (e.g. python, c, verilog). If the bash parameter is undefined, they will fill the generated parameter with an illegal value. *param_python*, *param_shell*, *param_cpp*, *param_vlog* are the available utilities.

2.1.6 load_model()

Load a model script from the *UTILS/models* directory. Different models are available and can be used based on user's needs. For more details, please refer to Section 6.

2.1.7 run_vsim()

This function changes directory to VSIM_BASE_DIR and executes the *vsim.sh* script, as the main script for cycle-accurate simulation. For more details please refer to Section 7.

2.2 GNU-Plot Functions

2.2.1 plot_xx()

These functions utilize *gnuplot* to plot the statistics gathered in current simulation scenario: *plot*, *plot_a_versus_b*, *plot_bar_chart*, *multiple_plot_to_file*, *multiple_plot*.

2.2.2 check_gnuplot_terminal()

Check availability of a specific GNU-Plot terminal.

2.3 Simulation Scenarios

2.3.1 create_scenario()

Set the __SCENARIO and __CASE variables based on current scenario, current case, and execution path of the scenario script. This function calls automatic_scenario() to create a name for the executing scenario.

2.3.2 update_scenario_location()

This function prepares the directory structure for execution of the senario by setting SCENARIO_HOME_DIR and SCENARIO_CASE_DIR based on the parameters specified in the scenario script. This function does not make any changes to the secondary storage yet.

2.3.3 automatic_scenario()

This function automatically create a scenario based on the name and execution path of the simulation script. The scenario location is created using the first characters of the execution path, and is stored in SCENARIO_HOME_DIR. For example, if we execute the following scenario:

```
./scenarios/c-gem5/f-sgraph/d-bellmanford/1-kernel.sh
```

The resulting scenario name will be:

```
cfd17
```

2.3.4 recalculate_global_params()

Before actual execution of the wrapped tools such as gem5 and ModelSim, global parameters need to be updated based on user inputs and default values. This function takes care of this step. Any intermediate parameter which is directly calculated from other parameters or user inputs should be assigned in this function. It is ensured that these parameters are updated before the actual execution takes place.

2.4 Disk Image Manipulation

2.4.1 mount_disk_image()

Mount a disk image to a local directory, to be able to write to or read from it.

2.4.2 copy_to_disk_image()

This function mounts a disk image, copies the specified files to it, and then unmounts it. It can get 8 input files as its arguments and it copies the files to the base directory (/) of the intended disk image.

2.4.3 copy_to_main_image()

Copy a list of files to gem5's main disk image identified by GEM5_DISKIMAGE.

2.4.4 copy_to_extra_image()

Copy a list of files to gem5's extra disk image (GEM5_EXTRAIMAGE) which is used for communication between host and the guest OS.

2.5 ELF Binary Manipulation

2.5.1 get_symbol_addr()

Get the relative memory address of the symbol $1 from the ELF binary $2 and export it in the name of $1=address. This function uses the *nm* utiliy to parse the ELF file and extract its symbol table. The resulting symbols are stored in the scenario working directory in symbols.txt. Later smc.sh passes these parameters to gem5 for simulation.

2.5.2 check_symbol_addr()

This function checks if a symbol's address has changed since the last compilation. This check is necessary for the dynamic binary offloading mechanism to work correctly.

2.5.3 get_symbol_size()

Get the size of the symbol $1 from the ELF binary $2 and export it as $1_size=size. This function is required for dynamic binary offloading, and the exact number of bytes to offload is identified by it.

2.5.4 get_section_info()

This function gets the memory address, size, and binary offset of the section $1 from the ELF binary $2 and export it in the name of $1_addr, $1_off, $1_size. *readelf* is utilized to parse the ELF file and the resulting variables are used for dynamic binary offloading.

2.5.5 hexdump_section()

Dump the binary code only for a specific section from the ELF binary into *.hex format. The actual parsing is handled by UTILS/dump_hex.py. Notice that the output file will be appended, and the resulting hex file is ready to be offloaded to PIM by the aid from the API. Taking a look at SW/PIM/resident/build.sh reveals that *.text* and *.rodata* sections of the ELF binary are extracted using this function to be offloaded to PIM.

2.5.6 check_section_size()

Check if a section fits into the available size dedicated to that section. If not, an overflow error occurs.

2.6 Manipulation of the Statistics

2.6.1 conv_to_csv()

This function calls the UTILS/conv_to_csv.py utility to convert the gathered stats (statistics.txt) into Comma Separated Values (CSV) format. This is automatically done at the end of the simulation.

2.6.2 conv_to_gnu()

This function calls the UTILS/conv_to_gnu.py utility to convert the gathered stats into GNU-Plot format. This is done only for one stat and the result is directly displayable in GNU-Plot by a call to the plot functions.

2.7 Process Management

2.7.1 control_c()

By pressing Ctrl+C control reaches this function. You can do the required cleaning up here.

2.7.2 safe_exit_process()

This function is called, whenever the simulation script is about to terminate due to any reason.

2.7.3 kill_process()

Kill a process if it exists, by asking for user's permission.

3 Gem5 Related Utilities: (common.sh)

Main utilities used by smc.sh and all gem5-based scenarios are defined here. These utilities are used for running the simulations, gathering statistics, measuring power consumption, etc. The relative path to this script is UTILS/common.sh.

3.1 Simulation Related Utilities

3.1.1 setup_directory_structure()

Setup the directory structure for intermediate files of the current simulation scenario. This function creates the required directories and actually touches the hard disk.

3.1.2 finalize_gem5_simulation()

This function is called once at the end of each simulation scenario script (after the execution of all cases has finished successfully). It performs some checks and then gather all required statistics by calling gather_gem5_stats().

3.2 Gathering Statistics

3.2.1 gather_software_stats()

Extract the statistics related to the software stack reported in *system.terminal*. All stats generated by the device driver and API are gathered using this function.

3.2.2 report_gem5_stats()

This function is the main utility which reports all statistics given by globally exported variable GEM5_STATISTICS (see one of the scenario scripts as an example). First, an alias is created for some of the highly used stats by a call to *alias_gem5_statistics()*. Next power consumption is estimated and stats related to power are generated by a call to *calculate_power_consumption()*. Lastly, individual statistics are reported by subsequent calls to *report_gem5_stat()*. Notice that this function is called every time a simulation case is finished to report stats related to that specific case. At the end of the simulation, however, *gather_gem5_stats()* is called which is described in the next sections.

3.2.3 gather_gem5_stats()

This function is called once at the end of the simulation. It gather all statistics specified by the user in GEM5_STATISTICS, and reports them in statistics.txt. Moreover, by a call to conv_to_csv(), it also generates statistics in CSV format.

3.2.4 get_stat()

Get the value of a statistic from current gem5 simulation. If not found, zero is returned.

3.2.5 set_stat()

Create a new stat and add it to the statistics of current simulation. For example, power can be calculated based on other stats and added as a new statistic.

3.2.6 alias_gem5_statistics()

This functions allows for renaming common gem5 statistics to more meaningful names. For example, since kernel offloading occurs in *timestamp2*, the stat *timestamp2.sim_ticks* is aliased automatically to *sim_ticks.offload_kernel*. The

raw statistics gathered in gem5 are stored in *stats_gem5.txt* while *stats.txt* contains the aliased statistics, as well as, the newly generated stats and the original ones. More aliasing rules can be added to this function if required.

3.3 Disk Image Preparation

3.3.1 check_disk_images()

This function checks for the existence of the required disk images and complains if they are not found. Moreover, it copies the required utilities for booting and automatic scenario execution to the disk images. This utility is executed only if $SMC_VARS_DIR/DISK_IMAGES_CHECKED.txt is not available. Therefore, in order to prepare fresh disk images it is enough to delete DISK_IMAGES_CHECKED.txt and run the scenario script. Check and preparation of the disk images is done automatically.

3.4 Estimation of Power Consumption

3.4.1 calculate_power_consumption()

This is the main function called by report_gem5_stats() to calculate power consumption for different components. First, it converts all parameters of the gem5 simulation from *json* format to text format. Then, it calls CACTI to estimate energy consumption per access in the caches of the system. Next, by calling *calculate_power_for_timestamp()* power consumption related to specific timestamps are extracted. As described in the previous report, different phases of execution are broken into timestamps to make sure statistics are gathered separately for each phase. For example, *timestamp6* is when PIM executes the offloaded kernel and *timestamp8* is when the host processors are running the same kernel. timestamps are defined in SW/host/app/offload_xx, and are introduced in [5].

3.4.2 calculate_power_for_timestamp()

This function estimates power consumption in timestamp $1 based on the statistics gathered in that specific timestamp. Currently *timestamp6* and *timestamp8* are of interest representing PIM and host's execution of the kernel, respectively. However, other timestamps can be passed to this function, as well. Power consumption is estimated for each component type differently, and the results are added up in the end. Here is a list of the components considered in power consumption:

Serial Links: A communication monitor called Hmon is responsible for measuring the total bandwidth delivered to SMC. Energy per bit consumed in the serial links is adopted from [10], and the dynamic power consumed in the serial links is calculated based on it (link_dyn_pwr). When PIM executes the kernels, we can completey shutdown the serial links by means of power gating [9]. In this mode, we can assume that the power consumption of the

links is negligible. However, when the host executes the computation kernels, it is not possible to turn off the serial links. This is because power state transition for the serial links introduces long sleep latency in the order of a few hundred nanoseconds, and a wakeup latency of a few microseconds [1]. For this reason an idle-power term has been added to the consumption due to the transmission of the Null flits and keeping the links active. Idle power has been estimated based on the maximum power reported in [8] and link-efficiency in [11].

DRAM: DRAMPower is already integrated in gem5, estimating power consumption based on the activities performed in each DRAM bank. *averagePower* is directly reported in gem5 and we have verified its results versus public excel sheets of some DRAM manufacturers. For the atomic operations done in the vaults, we have added statistics to count the number of these operations: *atomic_fadd_count*, *atomic_min_count*, *atomic_inc_count*. Power consumed by these atomic operations is estimated by multiplying their energy/operation by their number total executions and dividing by the period of execution. For vault-controllers, energy per bit was estimated based on [3] and multiplied by their total bandwidth.

Caches: CACTI [14] has been used to estimate the power consumed in caches of the host system. The details of this procedure are handled in calculate_cache_power().

Interconnects: Energy per bit in different interconnects in the system has been estimated based on logic synthesis in a 28nm low-power technology library, and their consumed power is estimated based on their total number of bytes transferred over a period of time.

SMC Controller: The SMC Controller has been estimated to consume 10pj/bit based on previous experience in [12], and again its total bandwidth was used to estimate its power consumption.

Processors: The power consumed in the processors is estimated in calculate_cpu_power.

Finally, total power is estimated by summing up the power consumed in different components. *power_related_to_host* represents the power consumption when host is executing the computation kernel, and *power_related_to_pim* shows consumed power when PIM is executing. A more detailed description along with the obtained results is available in [2].

3.4.3 calculate_cache_power()

The power consumed in caches is estimated based on the total number of accesses to each cache and the energy per access achieved from CACTI [14]. CACTI is previously executed by a call to call_cacti(). Leakage power of the caches is also extracted and added to this value to give an estimation of the total power consumption.

3.4.4 calculate_cpu_power()

In this function at first, operating voltage of the processors is estimated based on their frequency (ranging from 1GHz to 2GHz) [7]. Next, the idle and active percentage for each processor in the system is extracted and idle and active power consumptions are estimated based on [7] and [13]. Through our experiments we obtained that this simple approximation leads to more meaningful results in comparison with McPAT power estimation tool.

4 First-Use Configuration Utility: (configure.sh)

This script should be executed on first use. It checks the availability of the required tools and reports their versions. Also, the directory structure and accessibility to the work directory is verified by this script. The relative path to this script is UTILS/configure.sh.

5 Default Parameters: (default_params.sh)

This script contains the definitions and default values for all environment variables utilized in the simulator. Each simulation script, first loads these default values, and then modifies some of them to create alternative cases. Apart from simulation parameters, directory structure is also defined inside this script. The full path to this script is UTILS/default_params.sh.

6 Loadable Models and Templates

This section describes the available models and templates. User can load them using *load_model()* function. All these scripts are located in UTILS/models/.

6.1 gem5_automated_sim.sh

This script provides the facility of fully automated batch simulation (non-interactive mode). To speed-up automated simulation, this script takes a checkpoint upon first execution, after the guest OS boots completely. This checkpoint is restored for later executions. Two types of checkpointing is provided: homogeneous (*homo*) and heterogeneous (*hetero*). When all cases of one scenario share the exact same architecture with minor differences in software or parameters such as clock period, *homo* checkpointing can be used. In this mode, only one checkpoint is taken for all cases, resulting in lower setup overhead. However, if simulation cases differ in the underlying architecture (e.g. number of CPUs, or levels of caches), *hetero* must be utilized, otherwise the checkpoint cannot be resumed correctly. After successful checkpointing, the simulation scenario will

be executed as a single job, and when finished, gem5 simulation is terminated and statistics are gathered.

6.2 gem5_perf_sim.sh

This script disables gem5's debugging features to speed-up simulation. In addition, it disables debugging messages from PIM's driver, API, and its resident code. This is necessary for performance analysis and gathering statistics.

6.3 gem5_vsim_if.sh

This script enables closed-loop simulation of gem5 and ModelSim. An example usage of this script is available in:

`scenarios/c-gem5/a-tests/2-arm-external-vsim.sh`

6.4 memory/hmc_2011.sh

This is an example script containing the default parameters for an HMC device. This model can used in the scenario scripts and all its parameters will be propagated to the underlying tools such as ModelSim, gem5, and DRAMSim2.

6.5 system/gem5_fullsystem_arm7.sh

As the name implies, this script contains parameters for an ARMv7 based host platform. Similarly gem5_fullsystem_arm8.sh contains definitions for ARMv8.

6.6 system/gem5_pim.sh

All parameters related to hardware/software of the PIM subsystem are defined in this script. For example PIM_CLOCK_FREQUENCY_GHz is a hardware parameter defining PIM's clock frequency, and PIM_DMA_XFER_SIZE is a software parameter defining the requested DMA transfer sizes. A function is also provided here (*pim_print_mem_map()*) for reporting PIM's memory map.

6.7 system/get_data_from_host_arm7.sh

As described in [6], a file transfer mechanism between host and guest operating systems is provided which allows for copying files into gem5. The *get_data_from_host* scripts have been designed for this purpose. User copies the required files to the *extra.img* disk image by a call to *copy_to_extra_image()*. Then inside the guest OS execution of *get_data_from_host* mounts the disk image, moves data to a work directory, and then unmounts the image. When *gem5_automated_sim* is used, all these steps happen automatically.

6.8 system/pim_bare_metal_link.sh

In order to build the resident code executing on PIM a link script is required. Execution of this script generates the required link script based on the parameters defined in UTILS/models/system/gem5_pim.sh. This script is later used by SW/PIM/resident/build.sh to link the object files and generate the final ELF binary for PIM.

6.9 system/cacti_l1.sh

This script and cacti_l2.sh translate the parameters loaded in current scenario into CACTI's input script. Some parameters are directly extracted from system parameters and statistics (e.g. block size and associativity of the caches), and some are directly assigned (e.g. transistor types).

6.10 traffic/gem5_traffic.sh

This script replaces the host processors with a traffic generator and sets its parameters. This is useful for accuracy comparison of the gem5-based SMC model and directly comparing it with the cycle-accurate model.

7 Cycle-Accurate Simulation

Cycle-accurate (CA) simulation (previously explained in [4]) is still available and supported in this version. All simulation scenarios located in *scenarios/a-vsim* should work without any problem. Here is a list of the scripts related to CA simulation along with a brief description.

7.1 HDL/VSIM/vsim.sh

This is the main wrapper for cycle-accurate simulation. It is executed by calling run_vsim() and it generates the required wrappers and intermediate files for cycle-accurate simulation.

7.2 HDL/VSIM/scripts/vsim_utils.sh

The main utilities related to cycle-accurate simulation such as gathering statistics from the cycle-accurate simulation, and plotting the results are defined here.

7.3 traffic/vsim_traffic.sh

To configure the traffic generators in the cycle-accurate simulation, this script must be utilized. This is a model script and is located in *UTILS/models/*.

8 Scenario Scripts

This section describes the recently added scenario scripts. These scenario scripts are used for the results reported in [2] and are located in *scenarios/e-DATE2016/*. All these scenarios operate on sparse graphs, therefore, their user level application is located in *SW/HOST/app/offload_sgraph/*, and their computation kernels executing on the PIM processor are *SW/PIM/kernels/sgraph_xx.c*.

8.1 General Description

All scenario scripts described in this section follow the same execution flow. First, they should be executed using an argument specifying the architecture. Here is an example showing how we have run these scripts:

```
./scenarios/e-DATE2016/1-offload-overhead.sh 7 -o
```

All scripts iterate over a set of parameters specified by VALUES0 to VALUESn, and report the statistics listed in the GEM5_STATISTICS variable. After loading the default models and values, some parameters are overridden. The common parameters used in all these scripts are defined in *common_params_date2016.sh* Next, automated gem5 simulation is enabled by loading the model *gem5_automated_sim.sh*. Scenarios which change the underlying architecture use *hetero* mode, and others use *homo*.

After this step, required software directories are cloned and built. Resident code, device driver, API, and finally the user level application are the required layers which are built in this step. The resulting builds from this step are copied to gem5's extra disk image using *copy_to_extra_image()*, and then the main simulation is started.

8.2 1-offload-overhead.sh

This scenario reports the overheads associated with offloading procedures. The *sim_ticks.offload_kernel* stat is the time it takes to offload a binary kernel code to PIM, and *sim_ticks.offload_task* is the time it takes to offload the pointer to the data structure to PIM. This includes cache flushing, building the slice table, and passing the pointers to PIM. Cache flushing overhead is approximated by iterating over the ranges to be flushed and sending request packets towards the caches. The actual flushing inside the caches is done in zero time. Another point to mention is that, even though the overhead of pinning the pages to the main memory (get_user_pages() system call) is considered in the simulation environment, in [2] we have omitted this overhead and assumed that data is preloaded in the memory. This is a fair assumption, because data must be loaded in the main memory whether host operates on it or PIM, so this cannot be considered an overhead specific to PIM.

8.3 2-graph-size.sh

In this scenario the number of nodes in the sparse graphs is changed from 4000 to 256000 and PIM's speedup in comparison with the host is reported. One point to mention is that, OFFLOADED_KERNEL_NAME identifies the name of the computation kernel to be executed on the graphs, and all kernel codes are located in SW/PIM/kernels. However, taking a look at the kernel codes reveals that each one can have different implementations (e.g. with or without DMA). For this reason another variable named OFFLOADED_KERNEL_SUBNAME is used to identify which implementation to use. By default this variable is set to "best". This results in execution of the *kernel_best* implementation in all cases.

8.4 3-pim-tlb.sh

This scenario studies the effect of PIM's TLB on total execution time in comparison with an ideal TLB. For this reason the number of elements in it is changed from 2 to 5 once for a timing refill model, and once for a functional refill model. PIM_DTLB_DO_IDEAL_REFILL distinguishes between these two cases.

8.5 4-host-cache.sh

In this scenario, for all caches on the host SoC, cache line size is changed from 64 Bytes to 256 Bytes. Execution time ratio of host to PIM and miss rate of the L2 cache during the execution of the host are reported as statistics:

sim_ticks.ratio_host_to_pim

timestamp8.system.l2.overall_miss_rate::total

8.6 5-pim-dma.sh

This script changes a software parameter (PIM_DMA_XFER_SIZE) which defines the number of bytes to request in each DMA transfer from 32 to 512 Bytes. This parameter should not exceed PIM_DMA_MAX_XFER_SIZE, and both of these parameters must fit in PIM_VREG_SIZE. To support bursts larger than 512 Bytes, PIM_VREG_SIZE must be increased. This may require an increase in the ELF sections of PIM defined in (UTILS/models/system/gem5_pim.sh).

8.7 6-hmc-atomic.sh

This scripts studies the effect of atomic hmc commands on execution time of PIM. USE_HMC_ATOMIC_CMD enables or disables use of atomic commands, and the size of the graphs are changed from 4000 to 256000. Taking a look at the kernel codes related to sparse graphs (sgraph_xx.c) reveals that this variable simply translates into the "#ifdef USE_HMC_ATOMIC_CMD" statement. sgraph_bellman_ford.c uses atomic integer min (HMC_ATOMIC___IMIN), sgraph_page_rank.c uses atomic floating point addition (HMC_ATOMIC___FADD),

sgraph_teenage_follower.c uses atomic integer increment (HMC_ATOMIC___INCR), and sgraph_bfs.c does not utilize any atomic commands.

8.8 7-pim-clk.sh

The clock frequency of PIM's processor has been swept from 1GHz to 2GHz, and its effect on execution time and power consumption is studied. For power consumption, as described in 3.4.4, voltage of the PIM processor is scaled down based on required frequency and then power consumption is estimated for the PIM processors.

8.9 8-host-acc.sh

This script tries to compare PIM with a similar host-side accelerator. An environment variable named MOVE_PIM_TO_HOST has been devised for this purpose, interpreted in the python script: *GEM5/utils/python/ethz_configs.py*. If this parameter is set to TRUE, the whole PIM subsystem will be detached from SMC and moved to the host side (behind the *membus* interconnect). All functionalities (e.g. DMA, TLB, atomic commands) remain the same, with the only difference being longer distance to the main memory. Graph size is also changed and the computation kernels related to sparse graphs are studied.

References

[1] J. Ahn, S. Yoo, and K. Choi. Low-power hybrid memory cubes with link power management and two-level prefetching. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[2] E. Azarkhish, I. Loi, D. Rossi, and L. Benini. Design and exploration of a processing-in-memory architecture for the smart memory cube. In *Submitted to DATE 2016*.

[3] B. Boroujerdian, B. Keller, and Y. Lee. LPDDR2 memory controller design in a 28nm process.

[4] I. L. L. B. Erfan Azarkhish, Davide Rossi. A cycle accurate simulation environment for the smart memory cube. Technical Report 1, ETHZ University, Nov. 2014.

[5] I. L. L. B. Erfan Azarkhish, Davide Rossi. Design and exploration of a processing-in-memory architecture for the smart memory cube. Technical Report 3, ETHZ University, Aug. 2015.

[6] I. L. L. B. Erfan Azarkhish, Davide Rossi. A full-system simulation environment for the smart memory cube. Technical Report 2, ETHZ University, Mar. 2015.

[7] J. Heinlen. Leveraging advanced physical IP to deliver optimized SoC implementation at 40nm and below. Talk by ARM Physical IP Division, Nov. 2010.

[8] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88, June 2012.

[9] G. Kim, J. Kim, J. H. Ahn, and J. Kim. Memory-centric system interconnect design with hybrid memory cubes. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 145–155, Sept 2013.

[10] S. Lloyd and M. Gokhale. In-memory data rearrangement for irregular, data-intensive computing. *Computer*, 48(8):18–25, Aug 2015.

[11] P. Rosenfeld. *Performance Exploration of the Hybrid Memory Cube*. PhD thesis. Univ. of Maryland, 2014.

[12] M. Schaffner, F. K. Gürkaynak, A. Smolic, and L. Benini. DRAM or no-DRAM? exploring linear solver architectures for image domain warping in 28 nm CMOS. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 707–712, San Jose, CA, USA, 2015. EDA Consortium.

[13] B. M. Tudor and Y. M. Teo. On understanding the energy consumption of ARM-based multicore servers. *SIGMETRICS Perform. Eval. Rev.*, 41(1):267–278, June 2013.

[14] S. Wilton and N. Jouppi. CACTI: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, May 1996.