MSim: A General Cycle Accurate Simulation Platform for Memcomputing Studies

Chun Zhang*, Peng Deng†, Hui Geng*, Jianming Liu* Qi Zhu† Jinjun Xiong‡ and Yiyu Shi*
*Department of Electrical and Computer Engineering, Missouri University of Science and Technology
†Department of Electrical Engineering, University of California, Riverside
‡IBM T.J. Watson Research Center

***Abstract*—The lack of accurate yet open to public simulation infrastructure has puzzled researchers in the memcomputing area for sometime. In this paper, we propose for the first time a full tool chain called MSim that supports the cycle-accurate microarchitecture level simulation for memcomputing studies. With MSim, the performance gains of utilizing memcomputing for arbitrary applications on user configurable computer system architectures can be evaluated in high accuracy. In addition, MSim provides flexible interfaces with pervasive object-oriented design, which makes it well-suited as a good base platform for researchers to explore new memcomputing technologies.**

I. Introduction

The ever increasing demand for high performance computing has introduced new challenges for computer systems. Among these challenges, energy efficiency and system reliability have emerged as major barriers to performance scalability for modern processors. Recently, the fascinating idea of in-memory computing, or memcomputing (i.e., utilize existing memory elements for computation purposes) has been proposed [7]. With rapid advancements in memory technologies, memcomputing presents a new dimension of exploration for computer systems, and has already received wide interests from both industrial and academic society.

The current studies are carried out in different levels. At device level, novel memory elements are designed for computation purpose. For example, Ventra et.al. have proposed the two-terminal electronic devices with memory (memelements), namely, memristive, memcapacitive or meminductive systems [9], to store and process information at the same physical location. On the other hand, circuit or architecture level memory technology innovations such as high-bandwidth 3D-integrated memories [11], high-density non-volatile memories (NVMs) [10] etc. have also been proposed and evaluated to carry out computation tasks. Finally, new system computing paradigms such as cache tuning [5] and reliability aware memcomputing [8], etc. are also studied which demonstrate the promising advantages of memcomputing.

However, although with the many researches in memcomputing as aforementioned, the validation of these memcomputing technologies are still performed by various in-house tools. The lack of open simulation platform has created an invisible barrier for those who newly enter this research field, and the ad-hoc tool chains also make it difficult for research collaboration (i.e., to fairly compare different memcomputing technologies). Worse still, most existing in-house tools are developed in high levels of system abstraction/model for reduced development efforts but at the cost of lacking microarchitecture level details, which is necessary to capture the true system performance. Although there exist many microarchitecture level simulators in the literature [4], [3] for computer systems, none of them supports the idea of memcomputing. To meet the need of researchers in the memcomputing area, we propose in this paper a flexible yet open-to-public cycle accurate microarchitecture level simulation platform MSim with the hope to further promote studies in memcomputing.

978-3-9815370-2-4/DATE14/©2014 EDAA

In particular, the simulation platform consists of the following tool chains as shown in Fig. 1: 1) A scheduler that statically schedules the operations to be computed in-memory. This is mainly done by profiling the instruction level data flow graph (DFG) of the application and then solves the optimal resource allocation and task scheduling problem. 2) An annotation engine that extends the existing instruction set architectures (ISAs) to support new memcomputing operations. Designed in non-intrusive manner, the annotation results provide backward compatibility with existing ISAs. 3) A cycle-accurate microarchitecture level simulation engine based on *gem5* [4] that accepts the annotated application for detailed simulation. Specifically, the extended memcomputing instructions are internally decoded to memory access micro operations of proper width, and in-memory LUT data organization is correctly handled. In addition, with well tailored object-oriented design, the simulation engine also provides flexible interfaces for users to assemble heterogeneous computing resources for design space exploration, embedding and evaluating new memory technologies for memcomputing, etc. 4) A report engine that provides user friendly simulation results. This includes an overall simulation summary for quick performance review and a detailed cycle-based event traces for detailed performance analysis and debugging.

The overall goal of MSim is to provide an open yet flexible infrastructure for various memcomputing studies. It can either be directly used to evaluate the performance of memcomputing technologies in conventional computer systems, or as the starting point to explore other architecture level or computing paradigm innovations with minimal development efforts. The inclusion of a scheduler into MSim provides the users with the full tool chain, which can transparently transform normal application executable (e.g., the one compiled by *gcc* that is unaware of memcomputing) into the one that utilizes the potential memcomputing resources. However, the scheduler itself is optional and users have the freedom to make use of the MSim's microarchitecture level simulation engine independently if desired. The full MSim package can be downloaded

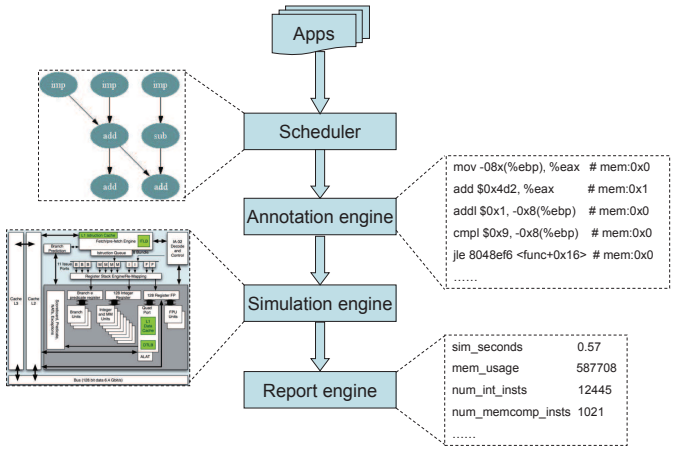

Fig. 1. The cycle accurate microarchitecture level simulation platform for memcomputing

at http://web.mst.edu/~yshi/downloads.html.

The paper is organized as follows. The microarchitecture level simulation engine as well as its interfaces, which are the kernels of MSim, are introduced in Section II. Then the scheduler of MSim is illustrated in detail in Section III. In Section IV, we demonstrate the full usage of MSim through a realistic case study. Finally, we provide a brief overview of on-going works to further enhance MSim's capabilities.

II. Cycle accurate microarchitecture level simulation engine for memcomputing

To provide accurate performance evaluations for memcomputing, we introduce our cycle accurate microarchitecture level simulation engine (i.e., MSim) in this section. In short words, it is built on top of the popular *gem5* simulation infrastructure [4] by integrating specific memcomputing ISAs into it. Thanks to the pervasive object-oriented design and script-based configurations of *gem5*, such integration works out in a natural way without the need to break any existing CPU or memory models. In addition, MSim is capable of accepting conventional application binaries (e.g., those compiled by *gcc*) as long as the memcomputing scheduling is provided in some manner (e.g., the scheduler to be introduced in Section III), and thus does not require any explicit programming efforts from the user side in order to utilize the memcomputing capability. The major components of MSim are introduced in the rest of this section.

A. Annotation engine

Accepting the outputs of the static memcomputing scheduling results such as those provided by the scheduler to be introduced in Section III, the annotation engine is in charge of generating a flag for each ISA instruction indicating whether it should be executed in-memory, or by the conventional function units of CPU (e.g., ALU). To avoid deprecations, we choose not to modify the original application binary (e.g., the one compiled by *gcc*). Instead, the annotation engine produces a separate annotation file containing the memcomputing flag corresponding to each of the machine instruction, which will later be parsed/decoded by the kernel simulation engine for annotated instructions to be executed in-memory. In this manner, the MSim is extended to support memcomputing without changing the existing ISAs.

Fig. 2 shows an example of the annotated ISA. In the second line, the *memcomp* flag is set to be 1, indicating this addition instruction is scheduled to be executed in-memory.

```
mov -08x(%ebp), %eax       # memcomp:0x0
add $0x4d2, %eax           # memcomp:0x1
addl $0x1, -0x8(%ebp)      # memcomp:0x0
cmpl $0x9, -0x8(%ebp)      # memcomp:0x0
jle 8048ef6 <func+0x16>    # memcomp:0x0
```

Fig. 2. An example of annotated ISAs, in which the *memcomp* flag annotates whether this is a in-memory computation instruction.

B. Cycle accurate microarchitecture level simulation engine

The simulation engine of MSim mainly extends *gem5* to support the execution of annotated memcomputing ISAs. In detail, it consists of major extensions to instruction decoding, micro-op assembling and memory content (i.e., LUT) organization.

Since a separate annotated ISA file is provided for memcomputing, the decoding stage of *gem5* is hacked to load this annotation file in addition to the binary executable. However, to achieve seamless integration, we utilize the ISA description language in *gem5* to characterize the meaning of an instruction, and an example is illustrated in Fig. 3. Note the operation width can be decided by the instruction postfix as well the width of operand registers so no other explicit hint is needed. In addition, the decoding process is described in C++–like manner, and thus we can reuse the internal *gem5* parser to automatically generate C++ code to be compiled for instruction decoding. Currently, six integral in-memory operations including, *addition*(ADD), *multiplication*(MLU), *square-root*(SQR), *division*(DIV), *sinusoid*(SIN) and *cosine*(COS) are supported to be consistent with the scheduler to be introduced in Section III.

```
decode MEMCOMP{
  0x0: Integer::add({{Ra = Rb + Rc;}})

  0x1: Memcomp::memadd({{
         Ra = load(translate(
                Rb,Rc,MEM_ADD));
       }});
}
```

Fig. 3. An example description to decode the in-memory addition instruction. In particular, if the *memcomp* flag is set to 1, the addition is transferred into a memory load, where the effective address is calculated by the *translate()* function based on the operands and the operation type.

For CISC ISAs like X86, an additional micro-op assembler is needed to translate the specific ISA instruction into a few fine-grained micro operations which are actually executed on CPUs. Similarly to the decoding stage, the micro-ops[1] each ISA instruction corresponds to is described in a specific python-like language as illustrated in Fig. 4. Note due to the exponential increase in memory space needed for wide operands, the micro-op assembler is also capable of

[1]The full list of supported micro-ops in *gem5* can be found in [2]

decomposing large arithmetic operations into smaller ones (i.e., decompose 32-bit addition into a few 8- or 16-bit additions).

```
def microop memadd(Ra, Rb, Rc){
  sll Rb, Rb, 8
  add Rb, Rb, Rc
  ld Ra, base_sig, base_sib, Ra
}
```

Fig. 4. An example description for micro-op assembly for the in-memory addition instruction. Assuming the memory look-up-table is organized as a continuous two-dimensional array with starting address described by *base_sig* and *base_sib* and 8-bit wide operands, the effective address containing the result for adding *Rb* and *Rc* is $Rb << 8 + Rc$. Thus, the in-memory addition can be decomposed into three micro-ops, where the first two compute the address in LUT and the result is calculated simply by fetching the memory content there.

Furthermore, the Msim simulation engine must be in charge of organizing and initiating memory contents to be looked up for in-memory computation. The current implementation organizes the LUT as a two-dimensional array for dual-operand operations (e.g., *addition*) and an one-dimensional array for single-operand operations (e.g., *sin*). Following similar calculations in [6], the memory space needed for an 8-bit addition is only 32KB.

Finally, to achieve extreme performance gains from memcomputing, we also implement the idea of having a specific memcomputing cache as suggested in [6]. In other words, this cache is used exclusively to accommodate LUTs for in-memory computation, and this is achieved by inheriting the *Cache* class and assembling it to the system through the flexible python-enabled configuration scripts in *gem5*. Note however, the memcomputing cache is not a necessary in MSim, but can be thought of as a solid example to demonstrate the capabilities of MSim to perform design space exploration for memcomputing aware computer microarchitectures.

C. Report engine

To enable full system analysis as well as to provide debugging aids, MSim is designed to dump out different levels of details for the simulation. As shown in Fig. 5, the summary report collects information of the entire run such as number of memcomputing instruction executed and simulated runtime of the application. This gives a quick overall system performance analysis database.

```
sim_seconds          0.57
mem_usage            587708
num_int_insts        12445
num_memcomp_insts    1021
......
```

Fig. 5. A snippet of the simulation summary

For those who requiring more information, the detailed report can also be generated to contain the events happed at each cycle. As shown in Fig. 6, this report is similar to the tracing information provided by *gem5*, but with extra information on how memcomputing instruction is fetched, decoded and executed. With these details, users are able to locate the bottlenecks/bugs in the current memcomputing paradigm, which is the basis for future improvement.

```
4201000: Fetch: PC:0x8048ef9
4201000: cpu0 : MemAdd : D=0x4d2
4201000: Event_20: tick @ 4201500
4201000: Fetch: PC:0x8048efb
4201500: cpu0 : IntAlu : D=0x3F
4201500: Event_20: tick @ 4202000
......
```

Fig. 6. A snippet of the detailed simulation report.

III. Memcomputing aware task scheduling

To supply the simulation engine of MSim with what part of the application to be computed in-memory, we propose a static scheduler in this section. Note this scheduler is an optional part of MSim in the way that users of MSim can provide their own task allocations for memcomputing by various techniques. However, for researchers who focus on memcomputing architecture studies, the proposed scheduler will save much of their efforts by providing a reasonably efficient transformation of conventional applications (e.g., those compiled by *gcc*) into memcomputing applications.

Given an application, we transform it into the equivalent data flow graph (DFG) representation. Then, our proposed static scheduler optimizes the application execution latency by mapping and scheduling of tasks[2] on various resources (including CPUs or memcomputing resources). The available resources for memcomputing (i.e., how much memory is assigned for memcomputing and for what types of operations) can be decided either through application profiling, or can be set up by the user directly. Given the available memcomputing resources and CPU resources, an MILP formulation is used to decide the task mapping and scheduling.

A. MILP for task mapping and scheduling

We assume a task graph $G(V, E)$ is given to represent the application, where the nodes V represent a set of relatively fine-grained atomic operations and the edges E indicate data dependency. The nodes in V are classified into two sets: computing node set V_C and memory node set V_M, i.e. $V = V_C \cup V_M$ ($V_C \cap V_M = \Phi$).

We also define a set of computing resources $R_C = R_P \cup R_M$ ($R_P \cap R_M = \Phi$), where R_P is a set of processor nodes and R_M is a set of memory computing nodes. Note that in this formulation, we assume the division between the memory computing nodes and the regular storage memory is given, and the configuration of the memory computing nodes is also given and will not change during the execution. R_M only includes the memory computing nodes and not the regular storage memory.

For any $v_i \in V_C$, we use $R_{v_i} \subseteq R_C$ to denote the set of resource nodes that v_i can be mapped to, including either processor nodes or memory computing nodes. For $r_j \in R_{v_i}$, C_{v_i,r_j} denotes the computation time of v_i when it is mapped to r_j. For any $v_i \in V_M$, we assume the access time to regular storage memory is given as C_{v_i}, which depends on the data size of the transaction (more accurate model may be developed later).

[2]Here, each task corresponds to one machine instruction to be executed.

For any $v_i \in V_C$, we define binary variable a_{v_i,r_j} to represent whether v_i is mapped to resource node r_j. s_{v_i} and f_{v_i} denote the starting time and finishing time of v_i, respectively. $Pred(v_i) \subseteq V$ is the set of nodes that immediately precede v_i, meaning there is an edge from v_j to v_i if $v_j \in Pred(v_i)$.

For any $v_i, v_j \in V_C$, h_{v_i,v_j} denotes whether v_i and v_j are mapped to the same resource node in R_C. p_{v_i,v_j} represents whether v_i has higher priority than v_j.

l_p is the latency of path p and v_p is the last node on path p. l_{max} is the maximum latency of all paths.

The MILP formula can be defined as follows:

$$min. \quad l_{max} \tag{1}$$

$$s.t. \sum_{r_j \in R_{v_i}} a_{v_i,r_j} = 1 \quad \forall v_i \in V_C \tag{2}$$

$$f_{v_i} = s_{v_i} + \sum_{r_j \in R_{v_i}} C_{v_i,r_j} \times a_{v_i,r_j} \quad \forall v_i \in V_C \tag{3}$$

$$s_{v_i} \geq f_{v_j} \times h_{v_i,v_j} \times p_{v_j,v_i} \quad \forall v_i, v_j \in V_C \tag{4}$$

$$s_{v_i} \geq f_{v_j} \quad \forall v_j \in Pred(v_i) \tag{5}$$

$$f_{v_i} = s_{v_i} + C_{v_i} \quad \forall v_i \in V_M \tag{6}$$

$$h_{v_i,v_j} \geq a_{v_i,r_m} + a_{v_j,r_m} - 1 \quad \forall v_i, v_j \in V_C \tag{7}$$

$$h_{v_i,v_j} \leq a_{v_i,r_m} - a_{v_j,r_m} + 1 \quad \forall v_i, v_j \in V_C \tag{8}$$

$$h_{v_i,v_j} \leq a_{v_j,r_m} - a_{v_i,r_m} + 1 \quad \forall v_i, v_j \in V_C \tag{9}$$

$$p_{v_i,v_j} = 1 \quad \forall v_j \in Pred(v_i) \wedge v_i, v_j \in V_C \tag{10}$$

$$p_{v_i,v_j} + p_{v_j,v_i} = 1 \quad \forall v_i, v_j \in V_C \tag{11}$$

$$p_{v_i,v_k} \geq p_{v_i,v_j} + p_{v_j,v_k} - 1 \quad \forall v_i, v_j \in V_C \tag{12}$$

$$l_p = f_{v_p} \tag{13}$$

$$l_{max} \geq l_p \tag{14}$$

Constraint (2) enforces node $v_i \in V_C$ to be mapped to only one computing resource node. Constraints (3, 4, 5) compute the starting and finishing time of v_i if $v_i \in V_C$. Constraints (5, 6) compute the starting and finishing time of v_i if $v_i \in V_M$. Constraints (7–9) determine whether v_i and v_j are mapped to the same computing resource r_m. Constraints (10–12) enforce the internal consistency of the priority assignment for each node $v_i \in V_C$. Constraints (13–14) compute the latency of each path and the maximum latency of all path.

Note that constraint (4) is not linear (shown in its current form for readability), but it can be converted to a set of linear constraints.

IV. A CASE STUDY OF MSIM USAGE

To illustrate the usage of MSim, we here go through one typical example as a case study. The application we target is a second order differential equation solver, *hal* [1], whose instruction-level DFG is shown in Fig. IV. The major architecture parameters and settings the scheduler used in MSim simulation are summarized in Table I. For *hal*, three 8-bit dual-operand operations (ADD, MLU, DIV) and three 16-bit single-operand operations (SQR, SIN and COS) are candidate operations for memcomputing, and the full memory space needed to build the LUT for each operation is 128KB.

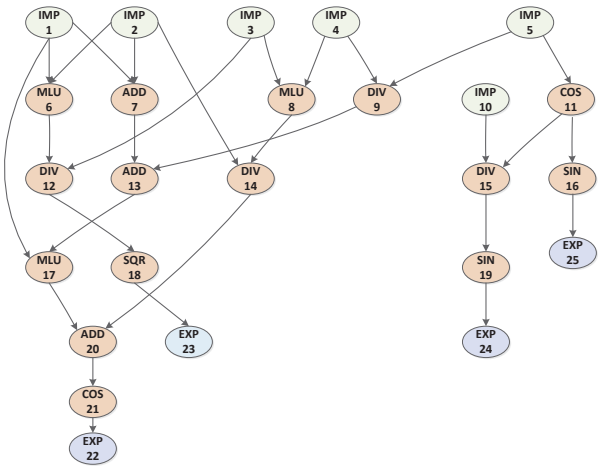

Fig. 7. The data flow graph of a second order differential equation solver

TABLE I. ARCHITECTURE PARAMETERS USED IN MSIM

ISA	X86
CPU frequency	2GHz
CPU model	AtomicSimpleCPU, TimingSimpleCPU
L1 data cache size	32KB
L1 data cache assoc	4
L2 cache size	1MB
L2 cache assoc	2
Main memory size	512MB
Assumed idle memory size	512KB
Memcomp instructions	ADD, MLU, SQR, DIV, SIN, COS

However, as mentioned in [6], it is not necessarily to load the entire LUT into memory due to the prevalence of operands locality existing in the application. As such, this is an upper bound of the memory space needed for memcomputing.

Table II illustrates the simulation results (in terms of performance speedup) of MSim for *hal* under different memcomputing configurations[3] with and without memcomputing, and two observations are worth noting. Firstly, a maximum 2x speedup can be achieved with proper memcomputing configurations, which demonstrates the strong capability of in-memory computing in improving application performance. Secondly, it is shown that different memory configurations have obvious impact on the final application, and the use of MSim is useful for such design space exploration.

V. FUTURE WORK

While the current implementation of MSim provides reasonably adequate capabilities for accurate simulation platoform

[3]A memcomputing configuration is defined as the operations the memory is configured to perform, or in other words, the contents in LUTs.

TABLE II. SCHEDULING AND SIMULATION RESULTS OF *hal* WITH AND WITHOUT MEMCOMPUTING

	w/o memcomputing	w/ memcomputing
ADD, MLU, SQR, SIN	1.0x	1.50x
ADD, MLU, DIV, SIN	1.0x	1.20x
ADD, SQR, COS, SIN	1.0x	1.80x
ADD, DIV, COS, SIN	1.0x	1.38x
ADD, MLU, DIV, COS	1.0x	1.29x
ADD, MLU, SQR, COS	1.0x	1.64x
MLU, SQR, COS, SIN	1.0x	2.00x

for memcomputing, we are also working the developing a few additional features including:

- The dynamic memcomputing scheduler. Although the static scheduler proposed in Section III provides reasonably efficient task assignment, it is not adapt to runtime changes and thus can become inefficient when the context changes. As such, a run time scheduler that dynamically assigns instructions (i.e., operations) into memory based on current resource availability and data dependency is desired. This includes memcomputing prediction, integration with normal out-of-order executions, etc.
- Runtime reconfigurable memcomputing resources. Similarly, the statically allocated memory LUTs may become inappropriate at different stages of application's lifetime. Consequently, we are working on developping the infrastructures to support runtime memory reconfiguration and evaluation such that the memcomputing resources can change adaptively according to the characteristics of applications current running.
- With the prevalence of multi-core systems, we are also making efforts to extend MSim to support multi-core platforms. This involves advanced support for cache coherence, etc.

REFERENCES

[1] The second order differential equation solver HAL. http://www2.imm.dtu.dk/SoC-Mobinet/modules/HLS/benchmarks/hal/hal.html.

[2] X86 microop ISA. http://www.m5sim.org/X86_microop_ISA#Load.2FStore_Ops.

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[5] H. Hajimiri, P. Mishra, and S. Bhunia. Dynamic cache tuning for efficient memory based computing in multicore architectures. In *VLSI Design*, pages 49–54, 2013.

[6] S. Paul and S. Bhunia. Dynamic transfer of computation to processor cache for yield and reliability improvement. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(8):1368–1379, 2011.

[7] S. Paul and S. Bhunia. Key features of memory-based computing. In *Computing with Memory for Energy-Efficient Robust Systems*. Springer, 2014.

[8] S. Paul, S. Mukhopadhyay, and S. Bhunia. A variation-aware preferential design approach for memory based reconfigurable computing. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 180–183. ACM, 2009.

[9] D. M. Ventra, V. Y. Pershin, and O. L. Chua. Putting memory into circuit elements: Memristors, memcapacitors and meminductors. In *Proceedings of the IEEE*, page 97, 2009.

[10] X. Wang, S. Narasimhan, S. Paul, and S. Bhunia. Nemtronics: Symbiotic integration of nanoelectronic and nanomechanical devices for energy-efficient adaptive computing. In *Nanoscale Architectures (NANOARCH), 2011 IEEE/ACM International Symposium on*, pages 210–217. IEEE, 2011.

[11] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.