Accelerate Cloud Computing with the Xilinx Zynq SoC

A novel reconfigurable hardware accelerator speeds the processing of applications based on the MapReduce programming framework.

by Christoforos Kachris
Researcher
Democritus University of Thrace
ckachris@ee.duth.gr

Georgios Sirakoulis
Professor
Democritus University of Thrace
gsirak@ee.duth.gr

Dimitrios Soudris
Professor
National Technical University of Athens (NTUA)
dsoudris@microlab.ntua.gr

Emerging Web applications like streaming video, social networks and cloud computing have created the need for warehouse-scale data centers hosting thousands of servers. One of the main programming frameworks for processing large data sets in the data centers and other clusters of computers is the MapReduce framework [1]. MapReduce is a programming model for processing large data sets using a high number of nodes. The user specifies the "Map" and the "Reduce" functions, and the MapReduce scheduler distributes the tasks to the processors.

One of the main advantages of the MapReduce framework is that it can be hosted in heterogeneous clusters consisting of different types of processors. The majority of data centers are based on high-performance, general-purpose devices such as the Intel Xeon, AMD Opteron and IBM Power processors. However, these processors consume a lot of power even in cases when the applications are not so computationally intensive but rather, are I/O-intensive.

To reduce the power consumption of the data centers, microservers have recently gained attention as an alternative platform. These low-cost servers are generally based on energy-efficient processors such as the ones used in embedded systems (for example, ARM® processors). Microservers mainly target lightweight or parallel applications that benefit most from individual servers with sufficient I/O between nodes rather than high-performance processors. Among the many advantages of the microserver approach are reduced acquisition cost, reduced footprint and high energy efficiency for specific types of applications.

In the last few years several vendors, such as SeaMicro and Calxeda, have developed microservers based on embedded processors. However, the MapReduce framework allocates several resources from the embedded processors, reducing the overall performance of the cloud-computing application running on these platforms.

To overcome this problem, our team has developed a hardware acceleration unit for the MapReduce framework that can be combined efficiently with ARM cores in fully programmable platforms. To develop and evaluate the proposed scheme, we selected the Xilinx® Zynq®-7000 All Programmable SoC, which comes hardwired with a dual-core Cortex-A9 processor onboard.

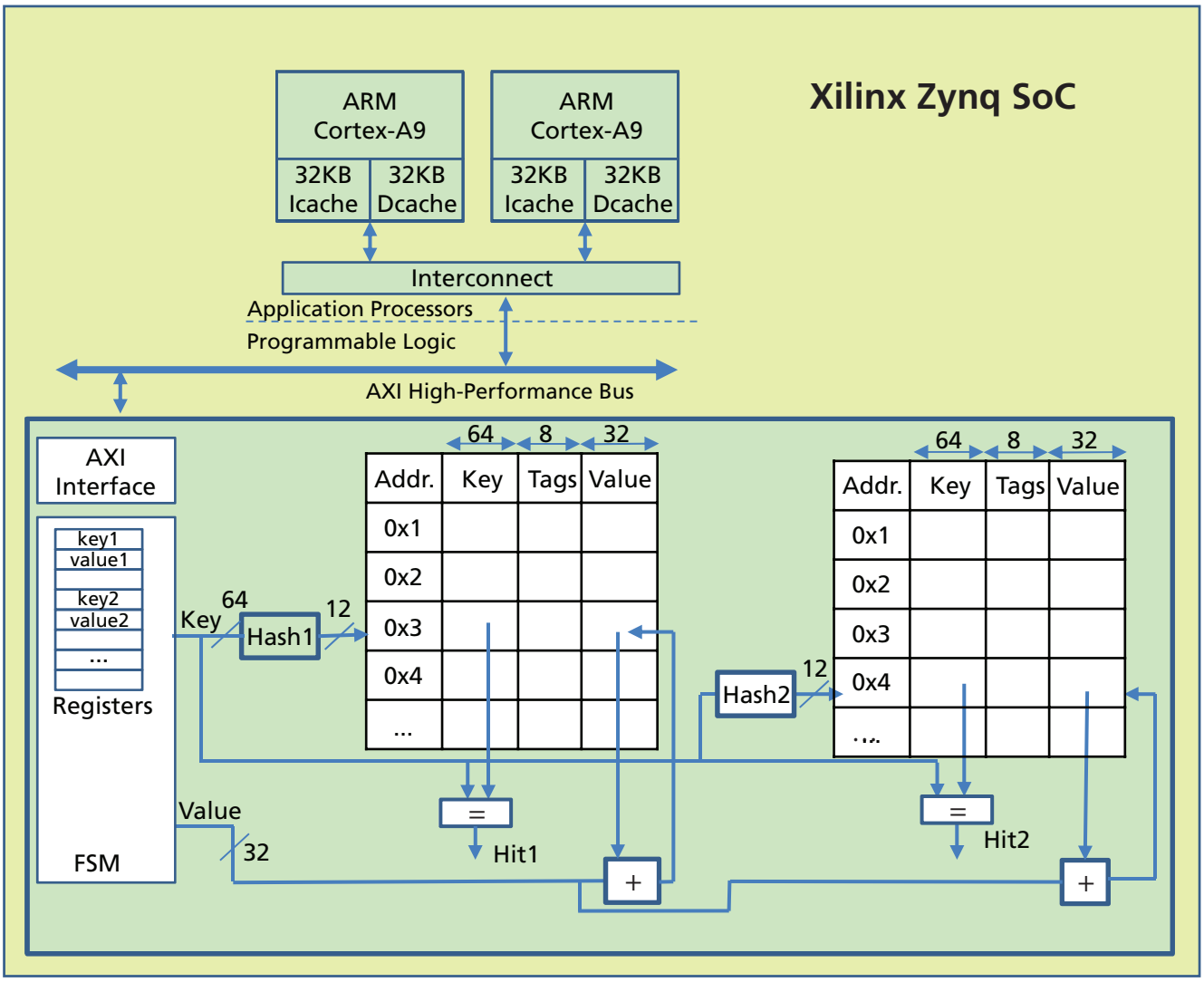

Figure 1 – Block diagram of the MapReduce hardware accelerator

MAPREDUCE HARDWARE ACCELERATOR UNIT

The MapReduce acceleration unit handles the efficient implementation of the Reduce tasks. Its main job is to merge the intermediate key/value pair from all the processors and to provide a fast access for the insertion of new keys and the updates (accumulation) of key/value pairs. We implemented the MapReduce accelerator as a coprocessor that can be augmented to multicore processors through a shared bus. The block diagram of the accelerator inside multicore SoCs is shown in Figure 1.

As the figure shows, we have incorporated the hardware accelerator unit into a Zynq SoC equipped with a pair of ARM Cortex™-A9 cores. Each of these cores has its own instruction- and data-level cache, each of which communicates with the peripheral using a shared interconnection network. The accelerator communicates with the processors through a high-performance bus that is attached to the interconnection network. The processors emit the key and the value that have to be updated to the MapReduce accelerator by accessing specific registers of the accelerator. After the end of the Map tasks, the accelerator has already accumulated the values for all the keys. The processors retrieve the final value of a key by sending only the key to the accelerator and reading the final value from the register. In this way, the proposed architecture can accelerate the MapReduce processing by sending a nonblocking transaction to the shared bus that includes the key/value pair that needs updating.

PROGRAMMING FRAMEWORK

Figure 2 shows the programming framework of the MapReduce applications using the hardware accelerator. In the original code, the Map stage emits the key/value pairs and the Reduce stage searches for this key and updates (accumulates) the new value by consuming several CPU clock cycles. By contrast, using the MapReduce accelerator, the Map stage

The hash function can accelerate the indexing of the keys, but it may create a collision if two keys have the same hash value. We selected cuckoo hashing as the best way to resolve hash collisions.

just emits the key/value pair; the MapReduce accelerator merges all the key/value pairs and updates the relevant entries, thus eliminating the Reduce function.

Communication from the application level running under Linux to the hardware accelerator takes place using the memory map (mmap) system call. The mmap system call is used to map a specified kernel memory area to the user layer, so that the user can read or write on it depending on the attribute provided during the memory mapping.

We use a control unit to access these registers and serialize the updates of the key/value elements. The key/value pairs are stored in a memory unit that you can configure based on the application requirements. The memory block contains the key, the value and some bits that are used as tags. These tags indicate if the memory line is empty and whether it is valid. To accelerate the indexing of the keys, a hash module translates the initial key to the address of the memory block.

In the current configuration, we have designed the memory structure to host 2K key/value pairs. Each key can be 64 bits long (eight characters) and the value can be 32 bits long. The total size of the memory structure is 2K x 104 bits. The first 64 bits store the key in order to compare whether we have a hit or a miss using the hash function. The next 8 bits are used for tags and the next 32 bits store the value. In the current configuration, the maximum value of a key is 64 bits and a hash function is used to map the key (64 bits) into the memory address (12 bits).

```
Original Code:
Map{
   Emit_Intermediate(key, value);
}
Reduce(key, value){
  search(key);
  update(key, value);
}

Accelerator code:
Map{
   Emit_Intermediate_Accel(key,value);
}
…
Emit_Intermediate_Accel(key,value)
{
   mmapped_addr = mmap(MapReduce_Accel);
   send(mmapped_addr + 0x4, key);
   send(mmapped_addr + 0x8, value);
}
```

Figure 2 – The programming framework

CUCKOO HASHING

The hash function can accelerate the indexing of the keys but it may create a collision in case two different keys have the same hash value. To address this problem, we selected cuckoo hashing as the best way to resolve hash collisions. Cuckoo hashing [2] uses two hash functions instead of only one. When a new entry is inserted, then it is stored in the location of the first hash key. If that location is occupied, the old entry is moved to its second hash address and the procedure is repeated until an empty slot is found. This algorithm provides a constant lookup time O(1) (lookup requires just inspection of two locations in the hash table), while the insert time depends on the cache size O(n). If the procedure should enter an infinite loop, the hash table will be rebuilt.

The cuckoo hashing algorithm can be implemented using two tables, T1 and T2, for each hash function, each of size r. For each of these tables, a different hash function is used, h1 and h2 respectively, to create the addresses of T1 and T2. Every element x is stored either in T1 or in T2 using hash function h1 or h2 respectively—that is, T1[h1(x)] or T2[h2(x)]. Lookups are therefore straightforward. For each of the element x that we need to look up, we just check the two possible locations in tables T1 and T2 using the hash functions h1 and h2, respectively.

To insert an element x, we check to see if T1[h1(x)] is empty. If it is empty, then we store the element in this loca-

Resources	Number	Percentage
Slice Registers	843	< 1%
Slice LUTs	903	< 1%
Block RAMs	29	21%

Table 1 – Programmable logic resource allocation

tion. If not, we replace the element y that is already there in T1[h1(x)] with x. We then check whether T2[h2(y)] is empty. If it is empty, we store the element in this location. If not, we replace the element z in T2[h2(y)] with y. We then try to place z in T1[h1(z)], and so on, until we find an empty location.

According to the original cuckoo hashing paper [2], if an empty location is not found within a certain number of tries, the suggested solution is to rehash all of the elements in the table. In the current implementation of our software, whenever the operation enters such a loop, it stops and returns zero to the function call. The function call then may initiate a rehashing or it may choose to add the specific key in the software memory structure as in the original code.

We implemented cuckoo hashing for the MapReduce accelerators as depicted in Figure 1. We used two Block RAMs to store the entries for the two tables, T1 and T2. These BRAMs store the key, the value and the tags. In the tag field, one bit is used to indicate whether a specific row is valid or not. Two hash functions are used based on simple XOR functions that map the key to an address for the BRAMs. Every time an access is required to the BRAMs, the hash tables are used to create the address and then two comparators indicate whether there is a hit on the BRAMs (i.e., that the key is the same as the key in the RAM and the valid bit is 1). A control unit coordinates access to the memories. We implemented the control unit as a finite state machine (FSM) that executes the cuckoo hashing.

PERFORMANCE EVALUATION

We have implemented the proposed architecture in a Zynq SoC. Specifically, we mapped the Phoenix MapReduce framework to the embedded ARM cores under Linux 3. Whenever the processors need to update the key/value pairs, they send the information through specific function calls. For the performance evaluation of the system, we used three applications from the Phoenix framework, modified to run utilizing the hardware accelerator: WordCount, Linear Regression and Histogram.

The proposed scheme is configurable and it can be tuned based on the application requirements. For the performance evaluation of the Phoenix MapReduce framework application, we have configured the accelerator to include a 4K memory unit (4,096 key/value pairs can be stored: 2K in each BRAM). The maximum size of each key is 8 bytes.

Table 1 shows the programmable logic resources of the MapReduce accelerator. As you can see, the accelerator is basically memory-intensive while the control unit that is used for the finite state machine and the hash function occupy only a small portion of the device.

Figure 3 compares the execution time of the original applications and the execution time of the applications using the MapReduce accelerator. Both of these measurements are based on the Xilinx Zynq SoC design.

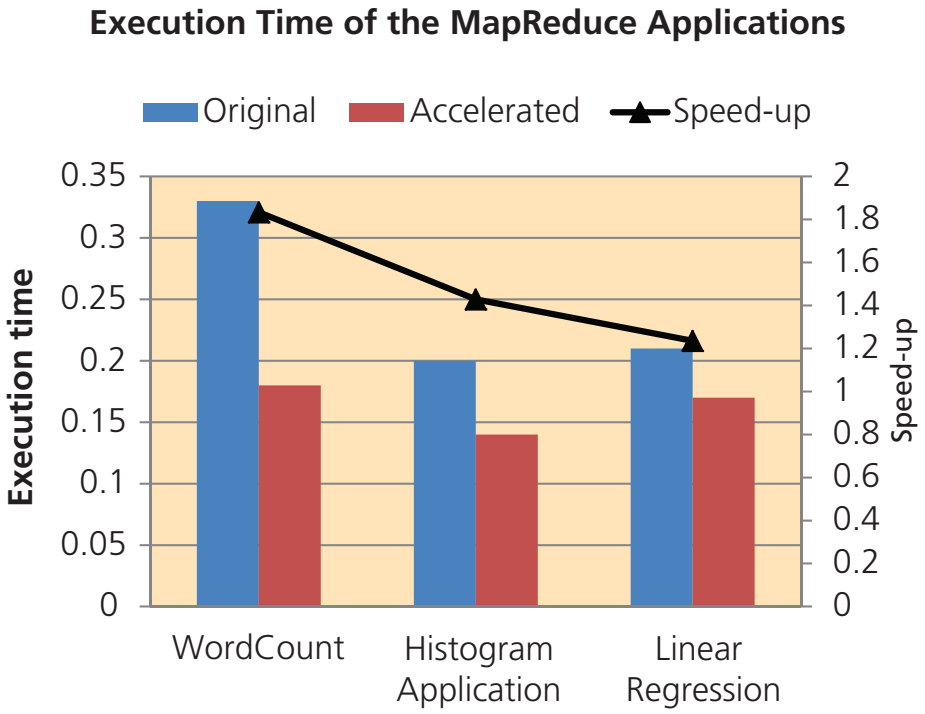

Figure 3 – The overall system speed-up was 1.23x to 1.8x, depending on the application.

In the case of the WordCount, in the original application the Map task identifies the words and forwards them to the Reduce task. This task gathers all the key/value pairs and accumulates the value for each key. In the accelerator case, the Map task identifies the words and forwards the data to the MapReduce accelerator unit through the high-performance AXI bus. The key/value pairs are stored in the registers (which are different for each processor), and then the accelerator accumulates the values for each key by accessing the memory structure.

PROCESSOR OFFLOADED

The reduction in execution time is due to the fact that in the original code, the Reduce task has to first load the key/value table, then search through the table for the required key. Then, after the accumulation of the value, the Reduce task must store the key back on the memory. By utilizing the MapReduce accelerator, we offload the processor from this task and thus reduce the total execution time of the MapReduce applications. Cuckoo hashing (O(1)) keeps the searching time of the key in the accelerator, while the processor is not blocked during the update of the key/value pair.

As Figure 3 shows, the overall speed-up of the system is from 1.23x to 1.8x. The speed-up depends on the characteristics of each application. In cases where the mapping functions are more complex, the MapReduce accelerator provides a lesser speed-up. In applications with simpler mapping functions that are allocated less of the overall execution time, the speed-up is greater, since a high portion of the total execution time is used for communication between the Map and the Reduce functions. Therefore, in these cases the MapReduce accelerator can provide much more acceleration. Furthermore, the MapReduce accelerator results in the creation of fewer new threads in the processors, which translates into less context switches and therefore reduced execution time. For example, in the case of WordCount, the average number of context switches dropped from 88 to 60.

The MapReduce framework can be widely used as a programming framework both for multicore SoCs and for cloud-computing applications. Our proposed hardware accelerator can be used to reduce the total execution time for the multicore SoC platforms such as the Xilinx Zynq SoC and the cloud-computing applications based on the MapReduce framework by accelerating the Reduce task of these applications. For more information on accelerating cloud computing with the Zynq SoC platform, contact the lead author, Dr. Christoforos Kachris, or visit *www.green-center.weebly.com*.

References

1. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Commununications of the ACM, vol. 51, no. 1, pp. 107–113, January 2008

2. R. Pagh and F. F. Rodler, "Cuckoo Hashing," Proceedings of ESA 2001, Lecture Notes in Computer Science, vol. 2161, 2001

3. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis, "Evaluating MapReduce for Multicore and Multiprocessor Systems," Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, HPCA '07, 2007, pp. 13–24

Acknowledgments

The authors would like to thank the Xilinx University Program for the kind donation of the Xilinx EDA tools. The research project is implemented within the framework of the action "Supporting Postdoctoral Researchers" of the operational program "Education and Lifelong Learning" (Action's Beneficiary: GSRT) and is co-financed by the European Social Fund (ESF) and the Greek State.