A Reconfigurable MapReduce Accelerator for multi-core all-programmable SoCs

Christoforos Kachris, Georgios Ch. Sirakoulis
Department of Electrical & Computer Engineering
Democritus University of Thrace, DUTH
Xanthi, Greece
Email: {ckachris, gsirak}@ee.duth.gr

Dimitrios Soudris
Department of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
Email: dsoudris@microlab.ntua.gr

***Abstract*—Phoenix MapReduce is a programming framework for multi-core systems that is used to automatically parallelize and schedule the programs based on the MapReduce framework. This paper presents a novel reconfigurable MapReduce accelerator that can be augmented to multi-core SoCs and it can speedup the indexing and the processing of the MapReduce key-value pairs. The proposed architecture is implemented, mapped and evaluated to an all-programmable SoC with two embedded ARM cores (Zynq FPGA). Depending on the MapReduce application requirements, the user can dynamically reconfigure the FPGA with the appropriate version of the MapReduce accelerator. The performance evaluation shows that the proposed scheme can achieve up to 2.3x overall performance improvement in MapReduce applications.**

***Index Terms*—Multi-core programming, MapReduce, scratch-pad memory**

I. Introduction

Emerging web applications like streaming video, social networks and cloud computing has created the need for warehouse scale data centers hosting thousands of servers [1]. One of the main programming frameworks for processing large data sets in the data centers and other clusters of computers is the MapReduce framework [2]. MapReduce is a programming model for processing large data sets using high number of nodes. The user specifies the *Map* and the *Reduce* functions and the MapReduce scheduler performs the distribution of 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.

However, whenever the MapReduce framework is mapped to either high performance processors or low-power embedded processors targeting microservers [3], many computing resources are consumed for the scheduling and mapping of the tasks to the cores and the execution of the *Reduce* functions of the MapReduce framework.

The research project is partially implemented within the framework of the Action "Supporting Postdoctoral Researchers" of the Operational Program "Education and Lifelong Learning" (Action's Beneficiary: General Secretariat for Research and Technology), and is co-financed by the European Social Fund (ESF) and the Greek State.

This paper proposes a novel MapReduce accelerator that is used to store, access and process the key/value pairs that are used in the MapReduce framework. The proposed MapReduce accelerator can be incorporated within any multi-core SoC platform in order to accelerate the processing of the applications that are based on the MapReduce framework and off-load the processor for commonly used tasks. To meet the different requirements of the MapReduce applications, four different versions of the MapReduce accelerator have been implemented. All of the versions could be implemented in a single design by adding the required multiplexers but this approach costs more, consumes more power and is wasteful when the specific features are not utilized by the applications. The proposed scheme has been prototyped in an FPGA with two integrated ARM cores. The processor, depending on the application requirements, can configure at run-time the FPGA with the right version of the MapReduce accelerator.

Overall the main contributions of this paper are the followings:

- A novel MapReduce accelerator for multi-core SoCs and FPGAs that accelerate the processing of the key/value pairs in applications based on the MapReduce framework.
- Efficient implementation and mapping of the proposed architecture in a Zynq FPGA with two embedded ARM cores
- Dynamic reconfiguration of different MapReduce accelerators based on the applications requirements
- Performance evaluation on the FPGA using typical cloud applications based on the Phoenix MapReduce framework showing that the proposed architecture can provide up to 2.3x overall speedup to the execution time

II. Related work

In [4], a reconfigurable MapReduce framework is presented but the proposed scheme is implemented as a custom design that is used to implement only the RankBoost application entirely on an FPGA. Both of the *Map* and *Reduce* tasks for the specific application have been mapped to configurable logic and thus for any new application a new design has to be implemented. In [5] a MapReduce Framework on FPGA

accelerated commodity hardware is presented where a cluster of worker nodes is designed for the MapReduce framework, and each worker node consists of commodity hardware and special hardware. Microsoft has also presented a reconfigurable fabric for accelerating large-scale data center services in [6]. The proposed scheme however is mainly based on FPGA placed into each server and has been implemented in order to accelerate the web search engine applications.

However, to the best of our knowledge this is the first design of a MapReduce co-processor targeting multi-core all-programmable SoCs that can be used to accelerate the execution time of the cloud applications based on MapReduce. All Programmable SoCs are processor-centric platforms that offer software, hardware and I/O programmability in a single chip [7]. The main advantage of the proposed scheme is that the hardware accelerator can be used as a co-processor for a wide range of applications that are based on the MapReduce framework. The user does not have to design a new accelerator for each application neither does the user have to change the code. The only change that is required is the replacement of some standard MapReduce functions with new functions that are interfacing with the hardware accelerator. Furthermore, the proposed co-processor can be configured to meet the applications requirements in terms of number of supported keys, maximum length of keys, and maximum size of key's value. Different versions of the proposed MapReduce accelerator can be loaded on-demand by partially reconfiguring the FPGA with the right version for each kind of MapReduce applications.

III. The Phoenix MapReduce framework

One of the most widely used frameworks that are hosted in the data centers is the MapReduce framework. MapReduce is a programming framework for processing and generating large data sets [2]. Users specify a **Map** function that processes a *key/value* pair to generate a set of intermediate *key/value* pairs, and a **Reduce** function that merges all intermediate values associated with the same intermediate key. Finally, the last stage merge together all the *key/value* pairs (Figure 1).

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system. In fact, many of the cloud computing applications are based on the MapReduce framework [2].

The MapReduce framework has been also implemented as a programming framework for multi-core architectures by Stanford University (called Phoenix MapReduce) [8]. Phoenix MapReduce framework uses threads to spawn parallel *Map* or *Reduce* tasks. It also uses shared-memory buffers to facilitate communication without excessive data copying. The runtime schedules tasks dynamically across the available processors

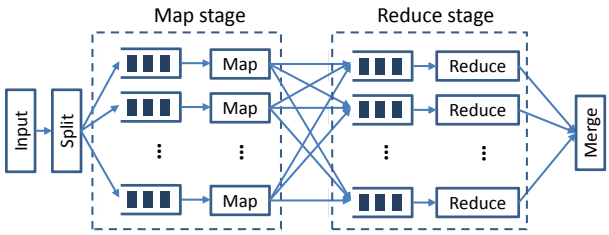

Fig. 1. The MapReduce programming framework

in order to achieve load balancing and maximizing task throughput. Locality is managed by adjusting the granularity and assignment of parallel tasks. Google's MapReduce implementation facilitates processing of Terabytes on clusters with thousands of nodes. The Phoenix MapReduce implementation is based on the same principles but targets shared-memory systems such as multi-core chips and symmetric multiprocessors. The main advantage of the Phoenix MapReduce framework is that it can provide a simple, functional expression of the algorithm and leaving parallelization and scheduling to the run-time system and thus making parallel programming much easier.

However, the MapReduce framework allocates several resources from these processors in order to perform the scheduling and the allocation of the tasks in the processors. Furthermore, the *key/value* pairs that are used, are organized in memory structures such as linked lists that can waste a lot of CPU cycles. To accelerate the execution time of the multi-core programs based on the Phoenix MapReduce framework, a MapReduce accelerator is proposed that is used for the fast indexing and processing of the *key/value* pairs of the applications.

IV. MapReduce Accelerator architecture

In this paper, we propose the incorporation of a MapReduce accelerator to multi-core SoCs that can be used to reduce the memory access time of the *key/value* pairs and can be also used to accelerate the processing of the *key/value* pairs. In essence, the MapReduce accelerator is used for the efficient implementation of the *Reduce* tasks that merges all intermediate values associated with the same intermediate key. In the original implementation of the MapReduce framework, every core processes a specific portion of the input data and whenever it encounter the predefined keys, it emits the key and the value to the *Reduce* tasks. On the other hand, the *Reduce* task merges all intermediate values associated with the same intermediate key. Every time that a *key/value* pair has to be updated with the new value, the processor has to load the key and the value from the memory, to process (e.g. accumulate) the old and the new value and then to store back the key and the value. Even if the *key/value* pair is in the cache, this operation takes many CPU clock cycles. On the other hand, the proposed MapReduce accelerator is used to replace the Reduce stage (Figure 1) with a single special scratchpad memory unit that is used to store and automatically process (e.g. accumulate) the values of the keys in MapReduce

Fig. 2. Programming framework

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

Code using the MapReduce Accelerator:
Map{
  Emit_Intermediate_accelerator(key,value);
}
```

application.

The main advantages of the proposed MapReduce scratch-pad memory are twofold: Firstly, it is a separate memory structure that is used only for the key/value pairs and thus it decreases the possibility of a cache miss if the *key/value* pairs were stored in the ordinary cache. Secondly, it merges efficiently the storing and the processing of the MapReduce values since there is an accumulator that can accelerate the addition of the current value with the new value every time that a processor emits a new value for a key that already exists. Therefore, it can reduce significantly the number of instructions that are required to accumulate the value of a *key/value* pair.

A. Programming framework

The code in Figure 2 shows the programming efficiency that this scratchpad memory can provide. In the original code, the *Map* stage emits the *key/value* pairs and the *Reduce* stage searches for this key and updates the new value by consuming several CPU clock cycles. On the other hand, using the MapReduce accelerator, the *Map* stage just emits the *key/value* pair and the MapReduce accelerator merges all the *key/value* pairs and updates the relevant entries thus eliminating the *Reduce* function.

B. Memory Architecture

The architecture of the MapReduce scratchpad memory is depicted in Figure 3. Each row of the scratchpad memory stores the key, the tags and the value for each MapReduce *key/value* pair. Since the key can be several bytes long, a hash unit is used to reduce the number of bytes to the maximum size of the cache.

The hash function can accelerate the indexing of the keys but it may create collision in case those two different keys have the same hash value. To address this problem, cuckoo hashing has been selected for resolving hash collisions. Cuckoo hashing [9] 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 the entry 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 constant lookup time $O(1)$ (lookup requires just inspection of two locations in the hash table) while the insert time depends on the cache

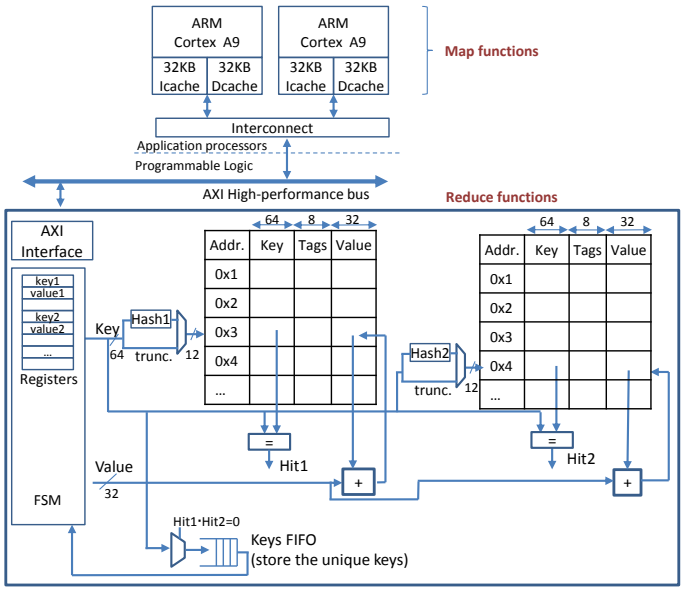

Fig. 3. Block diagram of the MapReduce accelerator

size $O(n)$. In case that the procedure enters an infinite loop the hash table is rebuild.

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 elements, 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 (i.e. $T1[h1(x)]$ or $T2[h2(x)]$. Lookups are therefore straightforward. For each of the element x that we need to look 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 if $T1[h1(x)]$ is empty. If it is empty, then we store it in this location. If not, we replace the element y that is already there in $T1[h1(x)]$ with x. We then check if $T2[h2(y)]$ is empty. If it is empty, we store it 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 [9], 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, whenever the operation enters in such a loop it stops the operation and return zero to the function call. The function call then it may initiate a rehashing or it may select to add the specific key in the software memory structure as in the original code.

Two block RAMs are used to store the entries for the two Tables, T1 and T2 as it is shown in Figure 3. These block RAMs store the key, the value and the tags. In the tag field one bit is used to indicate if 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 block RAMs. Every time that an access is required to the block RAMs, the hash tables are used to create the address and then two comparators are used that indicate if there is a HIT on the block RAMs (i.e.

TABLE I
HASH AND ARRAY MODE CONFIGURATION

Keys	Description	Hash units
:	Map tasks can emit any key	Enabled
*:k	Map tasks can emit to a fixed number of keys	Disabled

the key is the same as in key in the RAM and the valid bit is 1). The system is controlled by the Control unit (depicted in Figure 3). The Control Unit of the MapReduce scratchpad memory is implemented as a Finite State Machine (FSM) that executes the cuckoo hashing.

C. Accessing the keys

In many applications that are based on the MapReduce framework, the number of keys is deterministic and limited to a certain number. For example, in the *Histogram* application, the number of keys is limited and we are only interested about the final value of each key. Therefore, after the processing of the image, the processor can gather all of the *key/value* pairs just by requesting the value of a certain key.

However, in other applications like the *WordCount* application that will be described in the next section, the keys and the number are not known in advance. Therefore, after the end of the processing, the processors cannot request for the values of the keys since the keys are not know in advance. In the software implementation of the MapReduce, this problem is resolved by keeping a linked list that contains all of the keys. In the case of the software-hardware implementation using the proposed hardware accelerator, this issue can be resolved either in the software domain or in the hardware domain. In the latter case, the processors can keep a linked list of all the keys that have been emitted to the hardware accelerator. However, this solution requires a fast indexing of the keys as in the original code which may increase significantly the total execution time.

To solve this problem a hardware equivalent of the linked list has been implemented and augmented to the hardware accelerator. The hardware linked list is actually a FIFO that keeps all of the keys that have been inserted into the block RAMs (as it is depicted in Figure 3). Every time that a new *key/value* pair is added to the block RAMs of the hardware accelerator, the key is added to the FIFO. If the key is already in the block RAMs then the key does not have to be inserted in the FIFO. In this way, at the end of the processing the FIFO hosts all the unique keys stored in the block RAMs.

However, in case that the keys are known in advance, the proposed scheme can be configured to bypass the hashing units. In that case, the control unit receives the keys and the keys are directly used as index in the memory blocks. For example, in the case of the histogram applications, the keys are integer ranging from 0 to 255. Hence, the keys can be directly used as an index to the memory blocks (and a constant offset can be added to the key for different colors; blue; green and red). After the processing of the data, the user can just read the values for each key directly from the memory units, without the need of the hash units.

TABLE II
MAPREDUCE VERSIONS AND RESOURCE ALLOCATION IN ZYNQ7000

Function	Hashing	Processing	LUTs	BRAMs	File
Version1	YES	Accum.	3949(7%)	29(21%)	807KB
Version2	NO	Accum.	3916(7%)	29(21%)	792KB
Version3	YES	Average	4184(8%)	29(21%)	853KB
Version4	NO	Average	4151(8%)	29(21%)	847KB

D. Dynamically reconfigurable MapReduce accelerator

As it was described in the previous section, each application that is based on the MapReduce framework may have different requirements for the Reduce function. In the *Wordcount*, the *Histogram* and the *Linear Regression* applications, the *Reduce* function needs to accumulate the values for each key; In the *KMeans* application, the *Reduce* function calculates the average of the values for each key.

On the other hand, in the *Wordcount* and the *KMeans* applications the keys needs to be stored using a hash function while in the case of the *Histogram* and the *Linear Regression*, the keys are used directly as an index in the memory structures. Therefore, four different versions of the MapReduce accelerator has been developed as it is shown in the Table II. The first and the third version uses hashes to index the keys, while the second and the forth version uses directly the keys as an index. The first and the second version just accumulate the values of each key while the last versions calculate the average value for each key.

To evaluate the performance of the MapReduce scratchpad memory within a multi-core platform, the proposed scheme has been implemented and mapped to programmable FPGA-based SoC with hard-core processors. Specifically, the proposed scheme has been mapped to the Xilinx Zynq FPGA that incorporates two RISC Cortex A9 ARM cores and a programmable logic unit in a single chip [7]. Each of these cores has 32 KB Level 1 4-way set-associative instruction and data cache and they share a 512 KB Level 2 cache (Figure 4). The processors are clocked at 667 Mhz and they have coherent multiprocessor support.

Zynq platform has a high performance interface for the direct communication of the ARM cores with the programmable logic part. The high performance bus is based on the ARM AMBA 3.0 interconnection that has several advantages such as QoS, multiple-outstanding transactions and low-latency paths. Furthermore, there is a controller that is used for the configuration of the programmable logic called PCAP.

The four different modules of the MapReduce accelerator have been synthesized and implemented as partially bitstream files. The operation of the dynamic reconfiguration is depicted in Figure 4. The partially bitstream files are stored in a non-volatile memory. When the Zynq platform is turned on the whole bitstream file is transferred from the Flash memory to the programmable logic. The user could also transfer the partially bitstream files that contain the different version of the MapReduce accelerator to the DRAM for faster communication (shown as (1) in Figure 4).

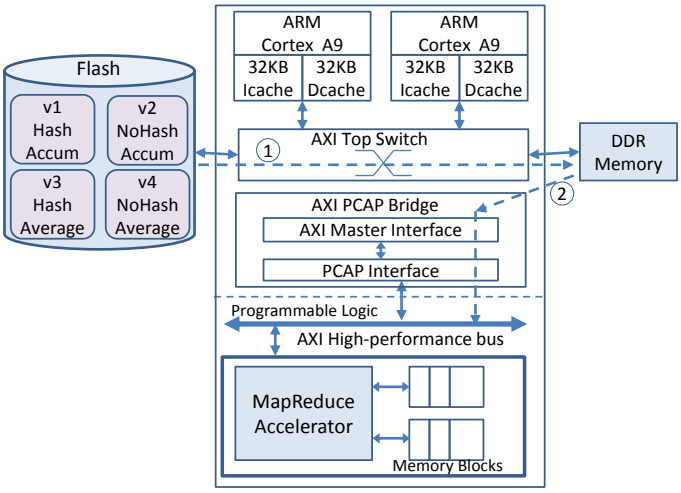

Fig. 4. Block diagram of the dynamic reconfiguration of the MapReduce

Depending on the application requirements, the system can partially reconfigure the MapReduce accelerator without affecting the operation of the rest of the device. Whenever a new bitstream needs to be configured, the bitstream is loaded from the DRAM and it is used to program the FPGA through the PCAP module using the AXI interface.

V. Performance Evaluation

The high level block diagram that integrates the MapReduce cache with the multi-core SoCs is shown is Figure 3. In this figure, the MapReduce scratchpad memory is incorporated into a programmable multi-core FPGA with two hard-wired ARM9 cores. These cores communicate with the peripheral using a shred interconnection network. The MapReduce scratchpad memory is communicating with the processors through a high performance bus that is attached to the interconnection network. The processors access the registers of the cache and they write the *key* and the *value* that they need to update during the *Map* stage. After the end of the *Map* tasks, the MapReduce accelerator has already processed the values for all the keys. The processors then, retrieve the final value of the key by sending only the key to the MapReduce scratchpad and reading the final value from the register. The main advantage of the proposed architecture is that it can accelerate significantly the MapReduce processing by sending a non-blocking transaction to the shared bus that includes the *key/value* pair that needs to be updated.

In the current configuration, the memory structure of the MapReduce scratchpad memory has been configured to host 2K *key/value* pairs. Each key can be 64-bits long and the value can be 32-bits long. The total size of the memory structure is $2K \times 104$ bits. The first 64 bits are used to store the key in order to compare if we have a hit or a miss using the hash function. The next 8 bits are used for tags and the next 32 bits are used to store the value. In the current configuration the maximum value of key is 64 bits and a hash function is used to map the *key* (64 bits) into the address (12 bits).

The Phoenix MapReduce framework [8] has been mapped to the embedded ARM cores under Linux Ubuntu. 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 four applications from the Phoenix framework have been used and modified to run utilizing the hardware accelerator: WordCound, Linear Regression, Histogram and Kmeans:

WordCount: The WordCount application is commonly used in search engines for the indexing of the web pages based on the words. It counts the frequency of occurrence for each word in a set of files.

Linear Regression: This application computes the line that best fits a given set of coordinates in an input file.

Histogram: It analyzes a given bitmap image to compute the frequency of occurrence of a value in the 0-255 range for the RGB components of the pixels. It can be used in image indexing and image search engines.

Kmeans: It implements the kmeans algorithm that groups a set of input data points into clusters.

A. Area resources

The main advantage of the implementation in an FPGA is that this scratchpad is configurable and it can be tuned based on the application requirements. The maximum size of each key is 8 bytes in the current configuration. In applications that the key is larger than 8 bytes for a specific application then the user can either increase the size of the words or follow a hybrid approach. In the hybrid approach, keys that are smaller than 9 bytes are stored in the hardware accelerator while keys that are 9 bytes or longer (rare case) are stored in a software structure. Table II shows the programmable logic resources of the MapReduce accelerator (including the AXI interface of the accelerator) and the percentage of the logic and the memory resources. In case of larger memory requirements, an external memory could be used (SRAM), slightly increasing the delay to access the keys.

B. Execution and Reconfiguration time

As it was described in the first section, the main advantage of the proposed scheme is that it can offload the processor from the indexing of the key/value pairs that is used for the reduce stage of the MapReduce framework. Figure 5 show the speedup of the proposed scheme using the MapReduce Scratchpad memory compared with the original code. All of the measurements are based on the Xilinx Zynq FPGA platform [7]. The measurement of the execution time is performed using the embedded Linux libraries (*sys/time*). In the original version, the applications were using only the ARM processors, while in the second case the applications were utilizing the MapReduce accelerator. The figure shows the speedup for all application using two different datasets (small and large). For example, in the case of the WordCount, the small dataset is 1MB and the large dataset is 10MB.

In the original application, the keys are identified by the *Map* task and are forwarded to the *Reduce* task. This task gathers all the *key/value* pairs and accumulates the value for

each key. In the case of the MapReduce accelerator, the *Map* task identifies the words and it just forwards the data to the MapReduce scratchpad memory through the high performance AXI bus. The *key/value* pairs are initially stored in the registers (that are different for each processor) and then the MapReduce scratchpad accumulates the values for each key by accessing the memory structure. The reduction of the execution time is due to the fact that in the original code, the *Reduce* task has to first load the *key/value* table, then to search through the table for the required key and after the accumulation of the value to store it back on the memory. By utilizing the MapReduce scratchpad memory we offload the processor from this task and thus reduce the total execution time of the MapReduce applications. The searching time of the key in the accelerator is kept low using the cuckoo hashing ($O(1)$) while the processor is not blocked during the update of the *key/value* pair.

As it is shown in this figure, the overall speedup of the systems ranges from $1.03\times$ to $2.3\times$. The actual time of the original application is shown above the columns. The overall speedup of the system depends on the characteristics of each application and the dataset. The lowest speedup is achieved in the *Linear regression* and the *Histogram* applications. In the case of the *Linear regression*, the performance is almost the same with the original code since the number of keys that are used is very small and thus index and the retrieval of the keys in the original code is very fast. The higher speedup is achieved in the case of the *WordCount* where both the hashing and the accumulation are used. In this case the number of keys and the keys are not known in advance and hence the use of the accelerator with the hash units for indexing can significantly improve the performance of the *Reduce* tasks. The lower speedup in larger files may be caused by the better caching of the keys in the original version.

The proposed scheme has been evaluated into a SoC with only two cores but it can support multiple cores depending on the application characteristics. For example, in the case of the *WordCount* application, the *key/value* pair is emitted to the accelerator every 260 ns. The average time of processing a pair by the accelerator is 25 ns (using 200 MHz clock frequency). Therefore, the proposed accelerator can sustain more than 10 cores in the same chip.

The reconfiguration time to load the MapReduce accelerator depends on the size of the configuration file (bitstream, Table II. In the worst case, the reconfiguration time is less than 4.2ms using the Processor Configuration Access Port (PCAP) module for programming the FPGA from the DRAM buffer. The low reconfiguration time shows that the proposed scheme can be used to dynamically reconfigure the FPGA with the MapReduce accelerators based on the application requirements without significant overhead.

VI. Conclusions

MapReduce framework can be widely used as a programming framework both for multi-core SoCs and for cloud computing applications. The proposed MapReduce accelerator located closely to the processors can be used to reduce

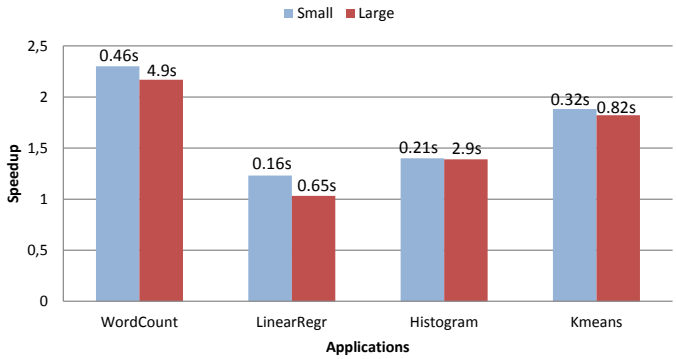

Fig. 5. Speedup of the applications using the MapReduce accelerator for small and large datasets. The actual time of the original version is shown in the number above the columns.

the execution time for the multi-core SoC and the cloud computing applications by accelerating the *Reduce* task of these applications. Furthermore, the proposed scheme can be configured to meet the application requirements by configuring the memory structure in the FPGAs (both in terms of key size and maximum depth). The performance evaluation using typical MapReduce applications showed that it can reduce significantly the execution time while offloading the processors for other tasks. The proposed scheme could be applied efficiently in FPGAs with embedded processors to accelerate typical applications based on MapReduce framework. The proposed scheme could be also incorporated to future multi-core processors used in data centers, accelerating significantly several cloud computing and server applications that are based on MapReduce framework.

References

[1] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commununications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[3] N. Bohra and C. Eylon, "Micro Servers: An Emerging Category For Data Centers," *Server Design Summit, November*, 2011.

[4] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce Framework on FPGA: A Case Study of RankBoost Acceleration," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2010, pp. 93–102.

[5] D. Yin, G. Li, and K.-d. Huang, "Scalable mapreduce framework on fpga accelerated commodity hardware," in *Internet of Things, Smart Spaces, and Next Generation Networking*, ser. Lecture Notes in Computer Science, S. Andreev, S. Balandin, and Y. Koucheryavy, Eds., vol. 7469. Springer Berlin Heidelberg, 2012, pp. 280–294.

[6] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=212001

[7] "Zynq-7000 Extensible Processing Platform Overview," in *Xilinx Technical Report, DS190*, 2012.

[8] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07, 2007, pp. 13–24.

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