IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 30, 2021, accepted August 25, 2021, date of publication August 27, 2021, date of current version September 7, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3108475

Developing a Multicore Platform Utilizing Open RISC-V Cores

HYEONGUK JANG[1,2], KYUSEUNG HAN[1] (Member, IEEE), SUKHO LEE[1], JAE-JIN LEE[1,2], SEUNG-YEONG LEE[3], (Student Member, IEEE), JAE-HYOUNG LEE[3], (Student Member, IEEE), AND WOOJOO LEE[3], (Member, IEEE)

[1]Electronics and Telecommunications Research Institute, Daejeon 34129, South Korea
[2]Department of ICT, University of Science and Technology, Daejeon 34113, South Korea
[3]School of Electrical and Electronics Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Woojoo Lee (space@cau.ac.kr)

This work was supported in part (50%) by the Ministry of Science, ICT (MSIT), South Korea, through the Development of Ultra-Low Power Intelligent Edge SoC Technology Based on Lightweight RISC-V Processor supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant 2018-0-00197, and in part (50%) by Chung-Ang University Research Grants in 2019.

ABSTRACT RISC-V has been experiencing explosive growth since its first appearance in 2011. Dozens of free and open cores developed based on this instruction set architecture have been released, and RISC-V based devices optimized for specific applications such as the IoT and wearables, embedded systems, AI, and virtual, augmented reality are emerging. As the RISC-V cores are being used in various fields, the demand for multicore platforms composed of RISC-V cores is also rapidly increasing. Although there are various RISC-V cores developed for each specific application, and it seems possible to pick them up to create the most optimized multicore for the target application, unfortunately it is very difficult to realize this in reality. This is mainly because most open cores are released in the form of a single core without cache coherence logic, which requires expensive design effort and development costs to address it. To tackle this issue, this paper proposes a method to solve the cache coherence problem without additional effort from the developer and to maximize the performance of the multicore composed of the RISC-V core selected by the developer. Along with a description of the sophisticated operating mechanisms of the proposed method, this paper details the architecture and hardware implementation of the proposed method. Experiments conducted through the prototype development of a RISC-V multicore platform involving the proposed architecture and development of an application running on the platform demonstrate the effectiveness of the proposed method.

INDEX TERMS Multicore platform, RISC-V, system-on-chip (SoC), electronic design automation (EDA).

I. INTRODUCTION

Instruction set architecture (ISA) is the essential vocabulary that allows hardware and software to communicate [1]. Over the past two decades, two major companies, ARM and Intel, have dominated ISA, and as a result, their microprocessors are now embedded in all computing devices from smallest to the fastest. However, after the recent rise of the RISC-V ISA [2], all of this is changing, and the microprocessor industry is turning upside down [3]. The RISC-V is a free and open instruction set with well-structured modularity, providing a very high level of flexibility at a very low cost and allowing users to produce custom chips suited to specific applications. As Linux gained popularity and acclaim in the operating systems, RISC-V pursues to become Linux in the processors [4], and is beginning to be used in various commercial products one after another.

As RISC-V is expected to be used in the design of new and more specialized processor cores that will soon emerge in wearables, home appliances, robots, autonomous vehicles and factory equipment, the need for RISC-V based multicore platforms is becoming increasingly urgent. Currently, there are various types of RISC-V cores that have been released, and by using them, it is ideally possible to configure them as customized multicores for various applications. However, in terms of practicality, building a multicore platform using

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

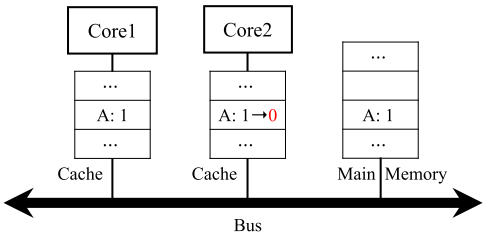


FIGURE 1. Example of the cache coherence problem.

the open RISC-V cores, especially released as a single core, poses unfortunately enormous challenges. This is mainly due to the cache coherence problem. Without solving this problem, the correct operation of the multicore is not guaranteed, or the expected performance is not exhibited. The currently possible method to solve this problem is to implement the cache coherence logic (CCL) for each multicore by the platform developers themselves, but this has a critical limitation that the design effort and development costs are very expensive and sometimes impossible.

To address the difficulty of developing multicore platforms utilizing RISC-V cores, we propose a method that solves the cache coherence problem without CCL and maximizes the performance of multicore regardless of which RISC-V cores are used in the multicore platform. The main idea of the proposed method is to avoid cache coherence problem by disallowing caching on shared data by default, and to allow temporarily caching on data that are obviously not shared for a period of time in order to compensate for the performance degradation caused by the inability to use the cache. We put it into the role of the programmers to determine which data are temporarily cached, but provide simple application programming interface (API) functions to make it easier for the programmers to apply this method when developing applications. In addition, we analyze and identify problems that may arise when the proposed method is applied to existing cache structure, and devise sophisticated behavior mechanisms of the proposed method to address them. Next, we develop the architecture to realize the proposed method in RISC-V core-based multicore platform, and implement the necessary hardware. We then build the proposed architecture into a network-on-chip (NoC) responsible for IP-to-IP communication in system-on-chip (SoC), so even if the developer selects any RISC-V cores and configures multicores, the proposed method can be applied. Moreover, by including the proposed architecture in the RISC-V-based SoC automatic design tool, we try to increase the usability of the proposed method. Finally, to verify the effectiveness of the proposed method, we implement the RISC-V multicore prototype platform including the proposed architecture on an FPGA, and develop a camera input-based handwriting recognition program as an application. Through the experimental work based on the application running on the FPGA, we confirm that the performance of the platform on which the proposed method is applied shows a performance improvement of about 37% over those on which it is not.

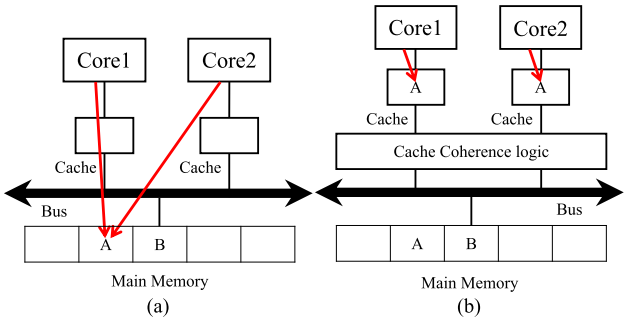


FIGURE 2. Solutions to the cache coherence problem: (a) simply non-caching the shared data and (b) using CCL.

The main contributions of this paper may be summarized as follows:

- As the most practical solution to the cache coherence problem in multicore development with RISC-V cores, a *temporary caching* (TC) method is presented.
- Along with the sophisticated operation mechanism of the proposed TC, a detailed description of the hardware and software development for TC is provided.
- Through prototyping of a RISC-V-based multicore platform to which TC is applied and the development of the application running on this platform, the effectiveness of the proposed solution is verified.

The remainder of this paper is organized as follows. Section II elucidates the cache coherence problem that can occur when a multicore is configured based on the RISC-V cores, and the existing solutions associated with them. Section III introduces the main idea of the proposed method and discusses problems that may arise from the proposed method. Next in Section IV, a detailed description of the architecture and hardware implementation for the proposed method are presented. Implementing the proposed architecture on NoC and developing it to be automatically designed are provided in Section V. Section VI is to develop a prototyped RISC-V multicore platform and applications as test benches and to provide experimental results obtained from them. Finally, Section VII concludes the paper.

II. CACHE COHERENCY PROBLEM IN RISC-V MULTICORE

Cache coherency problem is a well-known problem on multicore due to the caches being distributed across individual cores. Since each core has its own cache, the copy of the shared data in that cache may not always be the most up-to-date version, resulting in data synchronization failures that possibly crash the program or the entire computer. FIGURE 1 shows a simple example of the cache coherency problem. In the figure, there is a dual-core processor with Core1 and Core2, where each core brought a memory block for the variable *A* into its private cache. And then Core2 writes 0 to *A*. When Core1 attempts to read *A* from its cache, it will not have the latest version, producing incorrect results.

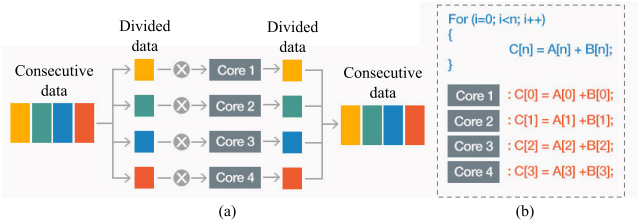


FIGURE 3. Example of (a) conceptual diagram of temporary caching in parallel processing and (b) its code.

To tackle the cache coherence problem in multicore, tremendous research efforts have been continuing for decades, and various solutions have been proposed. These solutions can be divided into software-based and hardware-based schemes. The software-based schemes refer to approaches of caching and maintaining data coherency in software by analyzing shared data [5]–[9]. These schemes mainly solve the cache coherence problem by improving the compiler, and sometimes by requiring special hardware assist. Unfortunately, however, a compiler that completely solves the cache coherence problem has not yet appeared on the market [10].

The software-based scheme that can be used in practice is to allocate all the shared data used by multiple cores into a non-cacheable region at compile time. Then the cores read the shared data directly from the main memory without caching, which is described in FIGURE 2 (a). This scheme has advantages in terms of practicality because the system developer does not need to modify the existing compiler, and it has the advantage in terms of programmability because the program works correctly even if the software developer does not consider the cache. Of course, the speed at which the core accesses shared data is slowed, so performance degradation is unavoidable with this scheme. In particular, in the case of applications that have a lot of data access, such as image processing, performance can be greatly degraded.

Due to the shortcomings of software-based schemes in terms of performance, hardware-based schemes are widely used in typical multicore systems. By utilizing additional hardware to synchronize the data in the caches, which is called CCL (cache coherence logic) as shown in FIGURE 2 (b), the schemes achieve the high performance of multicore platforms [10]–[13]. However, since the CCL is closely related to the cache structure, adding the CCL to an already-designed open source core is very expensive in design effort and development cost unless the CCL is considered and designed together when designing the core. Moreover, it is very difficult and impractical to implement CCL that targets several different cores rather than one kind of core.

Based on the above discussion, it may be very hard to solve the cache coherence problem by using the hardware-based scheme to develop a multicore platform using the RISC-V cores. To more realistically examine the development of a multicore platform using RISC-V cores, TABLE 1 lists existing RISC-V cores. As shown in the table, most of the RISC-V cores were released in the form of a single core without CCL. In order to configure a multicore with them while using a hardware-based scheme, platform developers have no choice but to implement the CCL by themselves. In the worst case, some cores do not provide readable RTL code, making it impossible to add the CCL. In addition, there are some RISC-V cores released as multicores and

TABLE 1. Features of the existing RISC-V cores.

Supplier	Core Name	# of Cores	Cache available?	CCL available?
Andes	A25,D25F,N22, N25F,NX25F	1	O (Yes)	X (No)
	A25MP,AX25MP	1∼4	O	O
	AX25	1	O	X
ETH	Ariane, RI5CY, Zero-riscy (PLUP)	1	O	X
CloudBEAR	BI-350	1	O	X
	BI-651,BI-671	1∼4	O	O
	BM-310	1	X	X
Codasip	Bk3, Bk5, Bk7	1		X
Darklife	DarkRISCV	1		X
Bob Hu	Hummingbird E200	1		X
FPGA Cores	Instant SoC	1		X
Cornell	Lizard	1	O	X
J. Chrisostomo	Maestro	1		X
LambdaConcept	Minerva	1	O	X
Tom Verbeure	MR1	1		X
OnChipUIS	OPenV/mriscv	1	X	X
VectorBlox	ORCA	1	O	X
Clifford Wolf	PicoRV32	1	X	X
Lucas Castro	ReonV	multi	O	X
MIT	RiscyOO	multi	O	O
Gavin Stark	Reve-R	1	X	X
SiFive	rocket	multi	O	O
Roa Logic	Roa Logic RV12	1	O	X
S. Tonello	RV01	1	X	X
Domipheus	RPU	1	X	X
rsd-devel	RSD	1	O	X
IQonIC Works	RV32IC_P5	1	O	X
	RV32EC_FMP5	1		X
	RV32EC_P2	1	X	X
Syntacore	SCR1	1	X	X
	SCR3,SCR4,SCR5	1∼4	O	O
	SCR7	1∼8	O	O
Olof Kindgren	SERV	1		X
Western Digital	SweRV EH1	1	O	X
IIT Madras	Shakti-Eclass	1	X	X
	Shakti-Cclass,-Iclass	1	O	X
risclite	SSRV	1	X	X
T-Head	XuanTie C910	1∼16	O	O
	XuanTie E902	1	O	X
SpinalHDL	VexRiscv	1	O	X

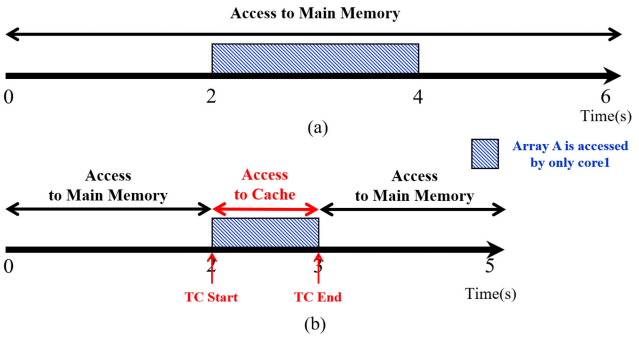


FIGURE 4. Memory access over time on Core1 for array A, when the TC is (a) not applied and (b) applied.

they support CCL, but these RISC-V cores have a big limit on scalability because they have a fixed number of cores. Furthermore, there may be cases where platform developers want to use existing RISC-V cores to construct heterogeneous multicores, even in these cases, developers are still faced with the problem of having to implement their own CCL. After all, platform developers want to choose the most suitable RISC-V core from the list, but the reality is that they will have a hard time building a multicore platform by scaling regardless of the cache coherence type.

III. TEMPORARY CACHING

A. MAIN IDEA

In this paper, as the most practical solution to develop multicore with RISC-V cores, we propose a new software-based scheme that can compensate for the performance degradation of the conventional software-based schemes without developing a new compiler or losing programmability. FIGURE 3 shows the motivation and main idea of the proposed scheme. In software-based schemes, when a program has a consecutive array and the array is generally shared data that can cause cache coherence problem, caching that array is strictly not allowed. However, as shown in the figure, if the array can be split into multiple pieces within a loop statement and can be executed independently on each core, performance can be improved if the programmer can temporarily allow caching of this array during the loop statement. Of course, after the loop is over, the caching for that array should be disabled.

In other words, the main idea of the proposed scheme is to allow the programmer to temporarily cache data when possible, i.e. we call this technique *TC* (temporary caching). The proposed TC improves performance by making it possible to dynamically cache data that originally had to be accessed from the main memory. For example, if array A is shared data as shown in FIGURE 4 (a), it must be accessed directly from the main memory. However, if A is accessed only by Core1 in a certain time, the programmer can program A to be cached for that time. FIGURE 4 (b) conceptually shows that the application of TC reduces the memory access time. In applications such as deep neural network operations and image processing where there is a lot of memory access and

```
1   void time_consuming_function(int *a)
2   {
3     int i;
4     for(i=0; i<1000; i++)
5     {
6       ... = a[i+1];
7       a[i] = ... ;
8     }
9   }
```

(a) Original Code

```
1   void time_consuming_function(int *a)
2   {
3     int i;
4     int* x;
5     x = tc_malloc(a, 1000);
6     for(i=0; i<1000; i++)
7     {
8       ... = x[i+1];
9       x[i] = ... ;
10    }
11    tc_free(x);
12  }
```

(b) Modified Code

FIGURE 5. Progamming example.

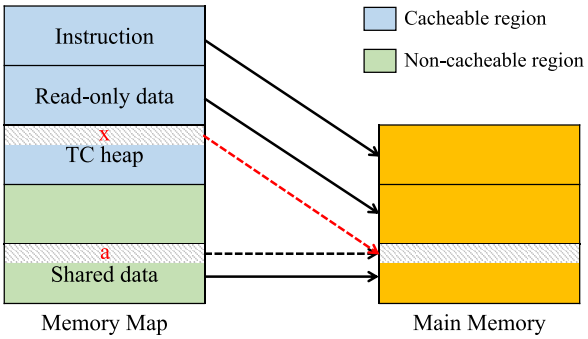
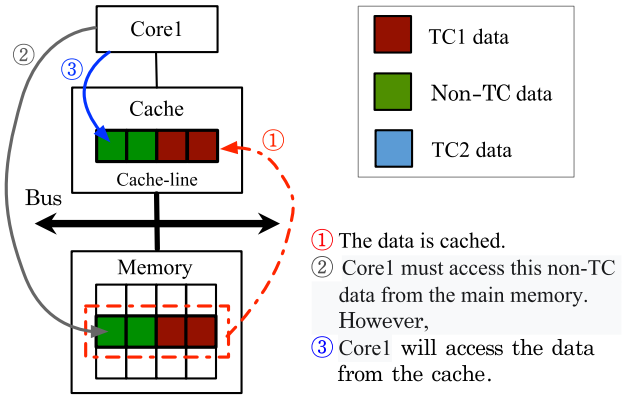


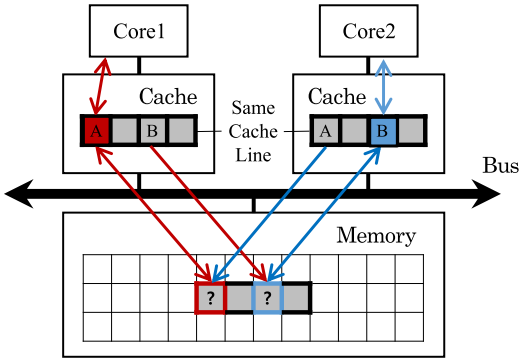
FIGURE 6. Mapping mechanism between memory map and main memory for TC.

TC can be applied frequently, the benefits of TC can be greatly appreciated.

In order for programmers to apply TC easily, we develop an API with `tc_malloc` function to start TC and `tc_free` function to end TC. FIGURE 5 shows an example of how to apply TC to the original program code using the provided API. In the original code, shown in FIGURE 5 (a), there is a variable `a`. When TC can be applied to the variable `a` in the program, as shown in the 5^{th} line of FIGURE 5 (b), the programmer calls `tc_malloc` with the start address and size of `a` as parameters. Then, the starting address of the new variable `x` that can be cached while having the same data as variable `a` is returned. More in detail, as shown in FIGURE 6, `tc_malloc` creates `x` allocated the same size as `a` in the space called TC heap in the cacheable region. For reference, the TC heap is secured from address space in the cacheable region in the memory map that is not actually mapped to memory or MMIO and not for the space for instruction data and read-only data. Then, `tc_malloc` dynamically sets the memory map so that this `x` is mapped to the space of the main



(a) TC data and non-TC data on the same cache-line



(b) Two different TC data on the same cache-line

FIGURE 7. Problematic cases when adopting TC.

memory mapped with `a`. After that, instead of using `a`, `x` is used in the code, allowing the core to fetch the same data as `a` from the main memory, put it in the cache, and use it. Finally, when it is no longer necessary or possible to cache the data, the programmer calls the `tc_free` with the starting address of `x` as a parameter, and `tc_free` flushes the cached TC data from the cache, frees the space allocated for `x` from the TC heap, and cut off the mapping to the corresponding main memory.

Meanwhile, when applying TC, programmers must take into account real-time computing, reliability, and power/energy consumption resulting from cache usage. More precisely, the use of cache in shared data can cause cache interference issues between tasks, which can significantly hamper the predictability and analysis of multicore real-time systems [14], [15]. Recent studies on cache architecture and cache coherence show that they have a significant impact on system reliability [16], [17]. Additionally, cache architecture and operational policies are well known to have a significant impact on overall system power and energy, so optimizing them has been intensively studied for more than a decade [18]–[20]. Furthermore, as the power density of chips increases, thermal design power (TDP) has become an important concern in modern chip designs [21], [22], and some studies have pointed out that the leakage current of the cache significantly affects the TDP of the overall system [23], [24]. After all, when applying TC, the programmer must optimize the target application with these factors in mind. Fortunately, TC is a software based scheme, so programmers can easily do this by trial and error using the provided TC API. In addition, compared to the large power overhead of the CCL [25], [26], which may adversely affect the TDP, TC is advantageous for TDP because the CCL is not required.

B. LIMITATION DUE TO THE CACHE-LINE PROBLEM

In a function, shared data can be divided into N number of short-term private data and a short-term shared data, where N is the number of cores. Each private data can be temporarily cached by each core, and we refer to these data collectively as *TC data*, and define *TCx data* by attaching the index x of the corresponding core to each. The short-term shared data still accessed from main memory is called *non-TC data*. Then, noting that data transfer between cache and main memory is basically done on a cache-line unit, we can notice that a fatal problem can occur if two or more types of TCx data or non-TC data are on the same cache line. We call this problem the *cache-line problem* and continue its detailed analysis.

First, when TC1 data is transferred to the cache, non-TC data just located near the TC1 data can also be cached, resulting in the cache-line problem. In this case, if the core attempts to access non-TC data, the data will be accessed from the cache and not from the main memory. FIGURE 7 (a) shows a detailed example of this problem, which can eventually cause critical system errors.

Next, when two different TC data belonging to the same cache-line, another cache-line problem can also occur. FIGURE 7 (b) describes this case, where variable A and B are TC1 and TC2 data, respectively. If A and/or B are modified and written back to the main memory, the wrong value can be stored in the main memory, due to the unintentionally-cached data in each cache. This also can cause a fatal system error, and there is no existing solution to prevent it.

The cache line problem puts a big limit on the use of TC. For example, as shown in the 1st line of code in FIGURE 8 (a), array `a` is shared, so it should not be cached on the dual core platform without CCL basically. On the other hand, as seen in the next for-loop in both codes, `a` is actually used independently in each core, whereby `a[0~49]` and `a[50~99]` are executed on each core. Therefore, to improve performance, it is desired to apply TC for `a` in each code, which is described in FIGURE 8 (b). However, since there is a high possibility that some cache-lines of `a[0~49]` and `a[50~99]` overlap, the programmer must never apply TC as in the example in FIGURE 8 (b). In other words, the programmer must conservatively apply TC only to data that clearly does not share the cache-line, which is a huge constraint on the use of TC.

C. PLAUSIBLE SOLUTIONS

To overcome the limitation of using TC, one can come up with a method of using a lock mechanism with TC. For example,

```
1  int a[100];                          extern int a[100];
2  void core1_work()                    void core2_work()
3  {                                    {
4    int i;                               int i;
5    for(i=0; i<50; i++)                  for(i=50; i<100; i++)
6    {                                    {
7      a[i] = ... ;                         a[i] = ... ;
8    }                                    }
9  }                                    }
```

(a) Original code.

```
1  int a[100];                          extern int a[100];
2  void core1_work()                    void core2_work()
3  {                                    {
4    int i;                               int i;
5    int *x;                              int *x;
6    x = tc_malloc(a,50);                 x = tc_malloc(&a[50],50);
7    for(i=0; i<50; i++)                  for(i=0; i<50; i++)
8    {                                    {
9      x[i] = ... ;                         x[i] = ... ;
10   }                                    }
11   tc_free(x);                          tc_free(x);
12 }                                    }
```

(b) Code applying TC.

```
1  int a[100];                          extern int a[100];
2  void core1_work()                    void core2_work()
3  {                                    {
4    int i;                               int i;
5    lock();                              lock();
6    x = tc_malloc(a,50);                 x = tc_malloc(&a[50],50);
7    for(i=0; i<50; i++)                  for(i=50; i<100; i++)
8    {                                    {
9      x[i] = ... ;                         x[i] = ... ;
10   }                                    }
11   tc_free(x);                          tc_free(x);
12   unlock();                            unlock();
13 }                                    }
```

(c) Code applying TC with lock.

FIGURE 8. Example codes for the false sharing problem.

as shown in the 5^{th} and 12^{th} lines in FIGURE 8 (c), a programmer codes to lock and unlock before `tc_malloc` and after `tc_free`, respectively, so that `a[0~49]` and `a[50~99]` are cached, updated, and flushed independently. In this way, performance is improved in terms of data access speed due to TC, but since programs are sequentially processed by lock, significant performance loss may occur in terms of data parallel processing, which may lead to overall performance degradation.

Data allocation by the compiler is also a plausible approach to think about, but in the end it is not appropriate. For example, one might think that inserting the proper padding between `a[49]` and `a[50]` would avoid the conflict, but this is only possible if the address of the array elements is linear, which is not possible in reality. On the other hand, allocating one element per cache line size at compile time will definitely prevent the two TC groups from mixing into the cache line. However, this approach not only increases memory usage extremely, but also significantly reduces performance by removing the spatial locality of the cache.

We may also consider a way to fundamentally block two or more TCx data and/or non-TC data belonging to one cache-line by copying each TC data to new data and using it. However, this method violates the aim of TC to improve

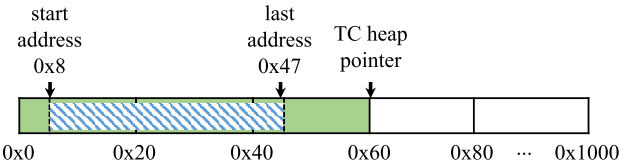


FIGURE 9. Example of assigning a new variable to TC Heap: the TC Heap is from 0 × 0 to 0xFFF, the cache-line size is 0 × 20, and the start address and size of the original variable are 0 × 10001068 and 0 × 40, respectively. The green part is the space allocated for the new variable, and the blue part is for the actual TC data.

performance, as it incurs memory resource waste and significant time overhead for copying data. Meanwhile, instead of the software-style approaches discussed above, we may think of a hardware-based solution that supports variables whose addresses are unaligned to the cache-line size. This method is ideally possible, but none of the existing core architectures, including the RISC-V cores, support this structure.

After all, all of the above solutions have fatal weaknesses. In particular, the problem is exacerbated by the inability to modify the RISC-V cores themselves. Under the conclusion that a solution based on software or hardware alone is difficult, we try to solve the cache-line problem through an approach that considers both software and hardware. Furthermore, to find the most practical solution, we considered the following issue in the software/hardware co-design approach: no matter how easily the developed software is available on the platform, if it is difficult to configure the platform using the necessary hardware with the software, this cannot be a practical solution. In the following sections, we will introduce our software/hardware co-design solution in detail and explain how to make this solution the most practical by implementing a way to automatically generate a multicore platform with the proposed hardware.

IV. TEMPORARY CACHING ARCHITECTURE

A. OVERVIEW

The cache-line problem can be solved by making the addresses of TC and non-TC data not consecutive. To do that, we introduce the concept of virtual addresses. The use of virtual addresses has the same effect as the copy-based solution discussed in Section III-C, which copies TC data to a new memory location, but the copy overhead can be avoided by mapping a virtual address to the original TC data. To realize this concept, it is necessary to develop system software that allocates virtual addresses and hardware that supports address translation, which is one of the major topics in this section.

Unfortunately, the introduction of virtual addresses alone cannot solve the cache-line problem because cache lines still have unintentionally-cached dummies. In other words, using a virtual address prevents the use of dummies, but cannot prevent them from being included in the cache-lines. There seems to be no problem because read/write does not directly take place on such unintentionally-cached dummies during

the program execution. However, the moment a write to TC data occurs, the entire cache line containing the TC data is changed to dirty, which means dummies can also be written to the memory during the write-back process, resulting in inconsistency. As a solution to this problem, we propose a method that uses a speculative approach to cache all data in the cache line containing TC data and ensure that these data work correctly when they are written back to memory, which is another major topic in this section.

B. VIRTUAL ADDRESS MAPPING

Virtual address mapping requires three components, the TC-MMU, TC heap, and TC APIs. First, the TC-MMU is a hardware unit that translates virtual address to physical address. It is similar to MMU for page table processing, but much simpler since direct translation is its only function. Next, the system software prepares the TC heap at compile time, which is a memory space in the cacheable region but does not contain actual data. It has a start address and an end address, but it does not include a compiled binary that is identical to the original heap used for dynamic memory allocation. Lastly, the TC API functions, `tc_malloc` and `tc_free`, are designed to perform the virtual address mapping internally. `tc_malloc` issues a virtual address when a physical address and the size of the target variable are given. When a programmer calls this function with the two parameters, the function allocates a specified amount of memory in the TC heap and returns its address. At the same time, it also configures the TC-MMU with the original address and new address by writing some registers. The description of these registers is given in Section IV-D, which presents detailed description of the hardware for TC.

During the heap allocation, the issued address must be aligned to the size of the cache line. This is to prevent problems that may occur due to cache-line overlap between different TC data in the TC heap. Moreover, we allocate memory in the way that the cache-line-offset of the virtual address is the same as that of the original physical one. The cache-line-offset refers to a value of the lower bits that are smaller than the size of the cache-line among all address bits. This approach will reduce the complexity of address translation logics in TC-MMU.

FIGURE 9 shows an example of assigning a new variable to the TC heap, where the start address and the size of the variable are 0×10001068 and 0×40, respectively, and the size of the cache-line is 0×20. As shown by the blue part in this figure, the start address of the new variable becomes 0×8 and the last address becomes 0×47, and the space allocated to the variable is larger than that, which is from 0×0 to 0×60 as shown in green in the figure.

When it is no longer possible to apply TC, the programmer executes the `tc_free` to flush the TC data in the cache and to release the allocated space for the variable in the TC heap. Since other variables can be allocated to the same address of current TC data in the future, `tc_free` must invalidate all the caches whether cache policy is write-back or write-through. The corresponding TC-MMU registers are also initialized, eliminating the mapping between the new variable and original data in the memory map. `tc_free` also contains a garbage collection process since the memory space for the TC heap is not infinite. The process can be implemented simply without any complex algorithm by tracking the number of alive TC variables in the two APIs; `tc_malloc` increases the number and `tc_free` decreases it. If the number becomes zero after the decrement, `tc_free` initializes a TC heap pointer, which is the variable to assign the next virtual address, to the start address of the TC heap. This initialization almost always takes place at the end of the function, so `tc_malloc` can now repeatedly issue the virtual address.

C. BYTE LEVEL MANAGEMENT

The basic idea to prevent the side effect of unintentionally-cached dummies is as follows: we stick with the traditional way that cache data is moved on a per-cache-line basis, but when data from the cache is written back to main memory, the dummies must not be written back to the main memory. For example, FIGURE 10 shows an example of the proposed method. In the figure, each TC data to is displayed in blue and orange, respectively, and the gray areas indicate the unintentionally-cached dummies. As seen in the figure, when writing back the data in main memory, only the blue and orange areas excluding the gray areas should be written.

To realize this idea, we propose a byte-level management method that exploits the byte enable signal, which is used to determine the validation of data in the conventional bus protocol [27]. In the bus protocol, data read or write is performed in units of data bus width, which is the size of data transferred per clock. If data smaller than the data bus width is transferred for the write operation, an unintended value may be written to the memory. To prevent this, a byte enable bit is placed for each byte in a byte lane to determine the validity of the data. Therefore, as many byte enable bits are used as the number of bytes of the data bus width. Meanwhile, in the case of a read operation, when reading data smaller than the data bus width, no byte enable bit is needed because there is no problem even if the data is not used except for the required part by taking the data as much as the data bus width.

In the proposed byte level management, only the byte enable bits of the part corresponding to the TC data are set to 1, and the rest are set to 0. For instance, in FIGURE 10, only the byte enable bit corresponding to the blue and orange data becomes 1, the rest becomes 0, and then only the part with the corresponding byte enable bits 1 is written back to main memory. Along with the byte-level management mechanism, we have designed optimized hardware for this, and thanks to this hardware support, the system itself ensures that there are no data inconsistency due to the unintentionally-cached dummies, allowing the programmer to actively and conveniently use TC without worrying about the cache-line problem. A detailed description of the designed hardware is provided in the following section.

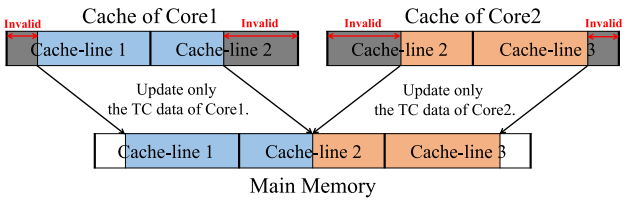


FIGURE 10. **Proposed solution to the problem of unintentionally-cached dummies.**

D. TEMPORARY CACHING UNIT

We have developed hardware that supports the virtual address mapping (i.e., TC-MMU) and byte level management functions, which we call a temporary caching unit (TCU). The TCU is written in Verilog hardware description language at register-transfer level, and is verified on a Xilinx FPGA. We first paid attention to the bus interface for communication between the core and the memory, and designed the TCU targeting the most commonly used AXI protocol [28], [29]. For reference, the AXI protocol consists of multiple channels of a read address (AR) channel, a write address (AW) channel, a read data (R) channel, a write data (W) channel, and a write response (B) channel, each of which works independently. Of the multiple AXI channels, the TCU is designed to implement the virtual address mapping function by controlling the signal of the AR/AW channel, and the byte level management function by controlling the signal of the W channel. The R and B channels in the AXI protocol are transmitted without any control from the TCU. The proposed architecture of the TCU is illustrated in FIGURE 11, and as shown in the figure, the TCU is largely composed of a block (on the left side of the figure) that takes the transfer of the AR/AW channel as an input, and a block (on the right side of the figure) that takes the transfer of the W channel as an input.

The block responsible for the virtual address mapping function of the TCU has dedicated hardware, called a *TC entry*, for mapping each variable to which TC is applied to its main memory address. The total number of TC entries in the block is the same as the number of variables to which TC can be applied at the same time, and platform developers can adjust this number as necessary. Each TC entry has two registers, each of which is to store the start address or the last address of the variable received from the TC API in order to determine whether the address accessed by the core through the AXI interface is the address of the variable to which TC is applied. Also, in the TC entry, there is a register to store the offset that is received from the TC API, which is used to convert the address of the TC variable to the address pointed by the original variable. Meanwhile, unlike TC entries, the mux-based control logic for each TC entry has the same configuration and function with each other, so we designed multiple TC entries to share one logic to reduce unnecessary overhead.

The main operation of the block being described is as follows. When an input address enters the TC entry from the AR/AW channel, the *Matched Decision Logic* in the TC entry compares the start address and the last address stored in the registers to the input address, and generates a *Matched signal* that can determine whether this address is the address of the variable to which TC is applied or not. At the same time, in the TC entry, the target address indicating the address of the main memory to which the TC variable is mapped is calculated by adding the value of the offset register to the input address. That is, the target address in the TC entry is calculated regardless of whether it is matched, but the target address finally becomes the output of the TC entry only when the Matched signal is 1 (cf. the Mux in the TC entry uses the Matched signal as a selecting signal), otherwise, the original input address will be the output of the TC entry. Then, a bitwise OR operation is performed on the Matched signals of all the TC entries, and the result is called the *TC signal*. This TC signal is used as the selecting signal of the next mux, which determines the final output address.

The right side block of the TCU in FIGURE 11 implements the byte level management function of the TCU. This block consists of a FIFO that stores the information received from the block on the left, a logic to create a burst address of TC data (we call this the *Burst Address Generator*), and a logic to control the byte enable signal (we call this the *WSTRB*[1] *Mask Generator*). The Burst Address Generator receives TC signal and AW channel information from FIFO as inputs (cf. FIFO Out1 in the figure), and generates the burst address as output. Along with this burst address, the WSTRB mask generator takes as input the TC signal, matched start addresses, matched last addresses, and matched signals from the FIFO (cf. FIFO Out2 in the figure), and outputs a WSTRB mask for byte level management.

The main operation of this block is as follows. According to the write operation of the AXI protocol, the write information generated by the AW channel (cf. AW channel info. in FIGURE 11) is stored in the FIFO of this block, and data is currently entering this block from the W channel once or several times in a certain size unit, depending on the data transfer mode determined based on the AW channel info. Then, when all addresses of the data correspond to TC data, i.e., when the TC signal is 1, the Burst Address Generator calculates the burst address of the corresponding data for each data transfer by using the AW channel information from the FIFO. Simultaneously, when the TC signal is 1, the WSTRB Mask Generator compares the burst address with the matched start address and matched last address to determine whether the transmitted data is TC data in byte levels. The WSTRB mask signal is then generated by setting the bit of the WSTRB mask corresponding to TC data to 1, otherwise, the bit of the WSTRB mask to 0. On the other hands, when the TC signal is 0, meaning that the data is non-TC data and its address belongs to the non-cacheable region, the WSTRB

[1] In the AXI protocol, the byte enable signal is called the WSTRB signal.

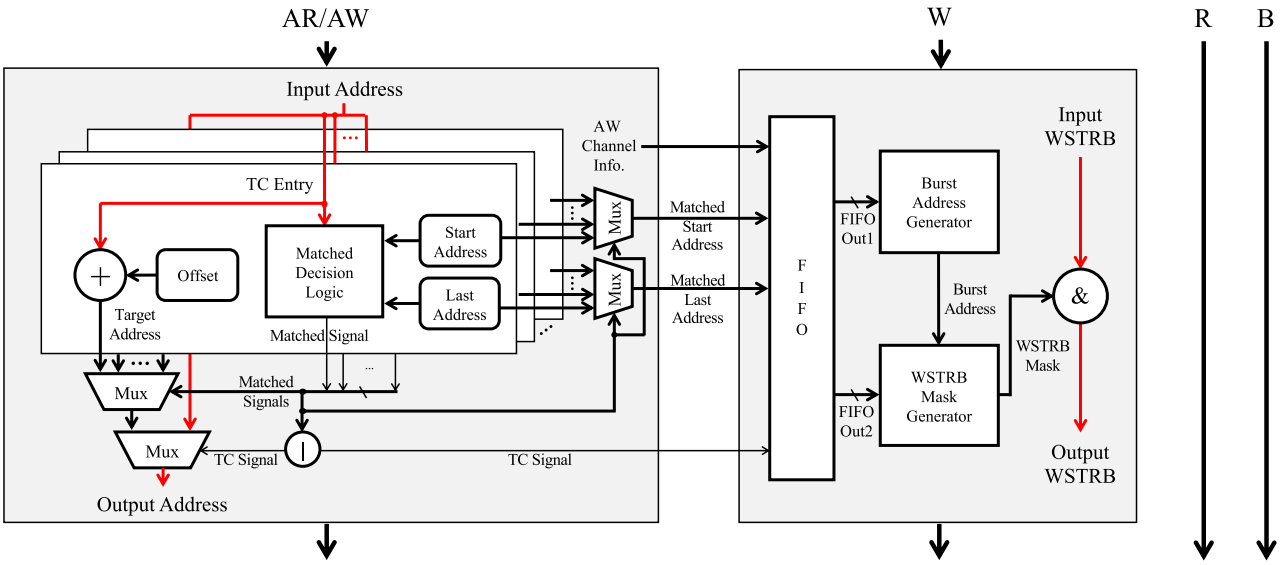


FIGURE 11. The proposed TCU architecture.

Mask Generator instantly sets all bits of the WSTRB mask to 1. Finally, a bitwise AND operation is performed between the generated WSTRB mask signal and the input WSTRB signal, and the converted output WSTRB signal is sent out from the TCU. Owing to the converted output WSTRB signal, only the TC data excluding the invalid portion of the data is written to the main memory as shown in FIGURE 10, so the cache-line problem due to the unintentionally-cached data does not occur.

V. EXPANSION OF TC CAPABILITY

A. EMBEDDING THE TCU INTO NETWORK-ON-CHIP

In order to answer question of where it is best to implement the developed TCU in a multicore platform, we focused on NoC, which plays a pivotal role of concurrent communication between IPs in the platform. Owing to the ability of NoC to overcome the limitations of the conventional bus-based system interconnects (e.g., unbearable increasing density and complexity induced by the system interconnect) [27], [30]–[32], NoC is commonly used in the state-of-the-art multicore platforms. FIGURE 12 (a) shows the conventional NoC architecture, and the processor core in the platform communicates with other IPs only through the dedicated network interface (NI) of NoC [28], [33]. Therefore, since the developed TCU operates independently between the core and the network, if it is embedded in NI, TC can be realized on the platform no matter what cores are used. In addition, as shown in FIGURE 12 (b), the design of placing the TCU inside the NoC does not require modification of the original internal structure of NI, so adding a TCU to NI can be easily designed without being limited to a specific NoC. In the end, we propose to embed the TCU in the core-dedicated

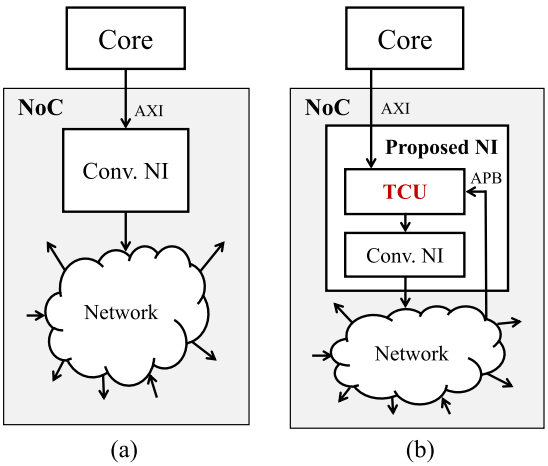


FIGURE 12. Architectures of (a) the conventional NoC and (b) the proposed NoC with the embedded TCU.

NI within the NoC as a general solution for TCU implementation.

In this paper, we implemented the TCU in our own NoC based on the presented architecture in [27], that is a compactly designed NoC that supports various types of IP interface conversion and has been silicon-proven in a fabricated SoC. To embed the TCU in the NoC, we first designed the TCU to have the advanced peripheral bus (APB) interface to configure the start address register, last address register, and the offset register in the TCU. This APB interface is connected to the NoC as shown in FIGURE 12 (b), so that the core can control the TCU using simple read/write memory operations. Next, we placed the TCU between the core and the existing NI, so that AXI data between the core and NI must be processed through the TCU.

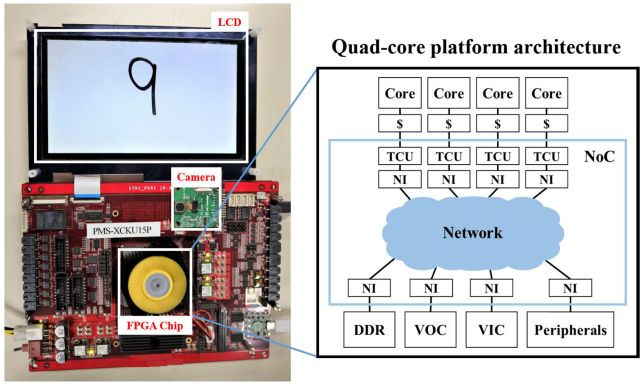


FIGURE 13. TC enabled multicore platform.

B. ENABLING DESIGN AUTOMATION OF RISC-V MULTICORE PLATFORMS WITH THE TCU

In a previously published paper [34], we introduced a new electronic design automation (EDA) tool, *RISC-V eXpress* (RVX), that allows the SoC developers to quickly and easily create SoC platforms using a variety of RISC-V cores. Indeed, there are many open source RISC-V cores and it is not difficult to acquire them, but the process of developing SoCs using such open source cores is very complex, which requires a lot of time and effort with high design skills and experience. To tackle this and ultimately accelerate SoC development, the RVX is developed to generate Verilog RTL codes, an FPGA prototype, and software development kit (SDK) for the target SoC, when the IPs to be integrated into the SoC are given using a high-level description.

In this paper, we have integrated the proposed TCU into RVX, so that RISC-V-based multicore SoCs equipped with a TCU can be automatically generated through the RVX. More specifically, we have implemented the TCU-embedded NoC in the RVX, allowing users to select this NoC as on-chip communication IP in the target SoC platform that connects the selected RISC-V cores and various necessary IPs. We also have upgraded the RVX to support an interface that enables users specify the number of TC entries per TC on the target platform. Finally, by using the upgraded RVX, we prototyped a TCU embedded RISC-V multicore platform and set up a software development environment. A detailed description of the experimental work performed using this is presented in the next section.

VI. EXPERIMENTAL WORK

A. PROTOTYPING A RISC-V MULTICORE PLATFORM

To verify the function and effectiveness of the proposed TC, we have implemented a complete verification system including the TC embedded multicore platform. Especially, the prototype platform was designed to have a quad-core, and for this, four Rocket [35] cores based on the RISC-V were implemented on the platform, each of which core was created as a single core without CCL. Additionally, this platform has a 512 MB DDR memory, video input/output controllers (VIC/VOC), and peripherals including UART, I2C, etc. Finally, all the IPs are interconnected with the developed TCU embedded NoC, that NoC has four NIs for each core, and each NI has a TCU with eight TC entries. Clocks of IPs are summarized in TABLE 2.

TABLE 2. Clock speed of each IP on the prototype multicore platform (MHz).

Cores	TCU	Peripherals	DDR
50	50	50	300

Video IPs	NoC
300, 200, 150, 50, 25	150

TABLE 3. Resource consumption on the FPGA.

	LUTs		FFs	
Cores (x4)	38,936	(34.4%)	16,060	(13.6%)
DDR Controller	23,144	(20.4%)	27,593	(23.3%)
Video IPs	8,137	(7.2%)	9,211	(7.8%)
Peripherals	10,787	(9.5%)	34,959	(29.6%)
TCUs (x4)	4,424	(3.9%)	3,644	(3.1%)
Conv. NoC	18,050	(15.9%)	20,271	(17.1%)
Etc	9,841	(8.7%)	6,500	(5.5%)
Total	109,807	(100%)	126,286	(100%)

To utilize video input/output, we designed a custom FPGA board. It contains a Xilinx FPGA chip (Virtex UltraScale+), DDR4 memories, a camera, and an LCD screen. FIGURE 13 shows the architecture of the developed platform, along with a picture of the actual implementation prototyped on the custom board.

The platform prototype was synthesized by using Xilinx Vivado [36], and resulting resource consumption of the TCU and the others are reported in TABLE 3. The four TCUs consume 4,424 look-up tables (LUTs) and 3,644 flip-flops (FFs), which takes only 3.9% and 3.1% in the entire platform.

B. DEVELOPING AN APPLICATION

We developed a handwriting recognition application based on camera input as an application to verify the validity of TC. In fact, the handwriting recognition applications are commonly used as a basic example of the deep neural network (DNN) [37], [38]. This basic handwriting recognition application recognizes an image in which one of the numbers 0 to 9 is handwritten, determines which number the image is, and shows the result. As the DDN architecture for this application, an architecture consisting of two convolution layers, two max pooling layers, and two fully connected layers was used, as shown in FIGURE 14. For DDN training, MNIST [39], the well-known handwritten image database, was used, and the parameters of the DNN trained in a Linux PC using TensorFlow [40], a deep learning framework, were applied to the handwriting recognition application.

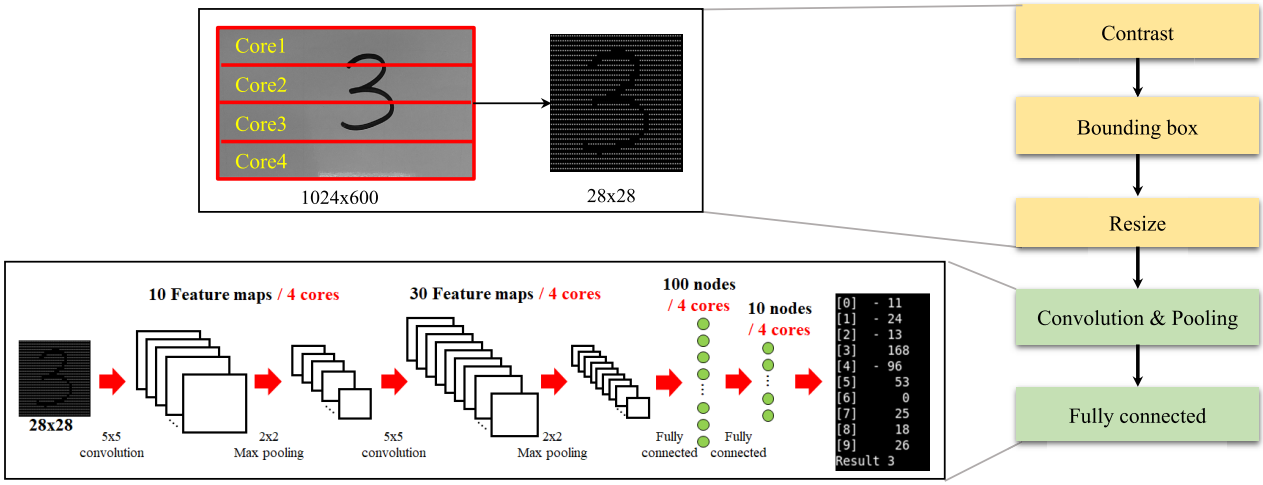


FIGURE 14. Multi-processing of the handwriting recognition application.

The MNIST database is composed of images of handwritten numbers, each image is in the format of 8 bits grayscale 28 × 28 pixels, and the handwritten numbers in the images are located at a certain size in the center of the image. In addition, since the MNIST database was used for training DNN, the input image of the DNN must be in the same format as the image of the MNIST database, and to improve the performance of DNN inference, the handwritten number should be located in the center of the image at a certain size, as do the images in the MNIST database.

In our target application that uses images taken by the camera connected to the multicore prototype, not only the format of the image obtained from the camera is different from that of the MNIST database, but the handwritten number may not be located in the center of the image. Therefore, to convert the image received from the camera into the image format of the MNIST database, and to place the number in the center of the image, we needed to implement image pre-processing such as contrast, bounding box, and resize (cf. yellow boxes in FIGURE 14). Finally, including this image pre-processing part, we have implemented the camera-input based handwriting recognition application by coding all of the inference parts of the DNN in C language.

Using the developed handwriting recognition application as a baseline, to experiment how much the performance improves when TC is applied, we wrote a testbench that applies TC to the baseline code that processes data in parallel. More specifically, by coding the testbench that applies TC to the image preprocessing process, we made the preprocessing performed in parallel on 4 cores. After the pre-processing process, the convolution layer and the max pooling layer of the application are distributed to 4 cores in unit of feature map, and multilayer perceptron (MLP) is distributed and processed in parallel through implementation.

The developed testbench and baseline application were run on a multicore platform consisting of four RISC-V cores prototyped on a custom FPGA, and the results of the experiment are reported in detail in the following subsection.

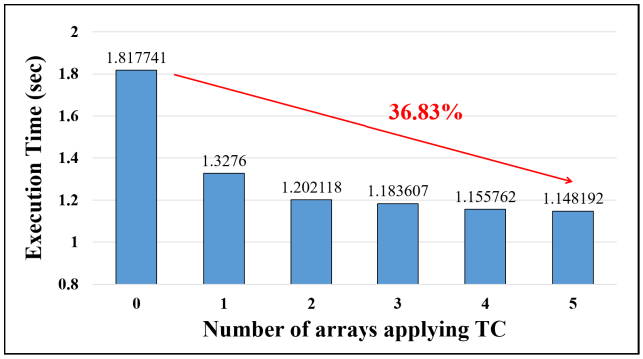


FIGURE 15. Measured execution time of the testbench running on the TC-enabled quad-core platform.

C. PERFORMANCE IMPROVEMENT RESULT

The testbenches are set by varying the number of variables to which TC is applied to the developed handwriting recognition application code. Then, the program execution times are measured by operating each testbench on the TC enabled quad-core platform (cf. FIGURE 13). More specifically, the number of variables to which TC is applied is 0 as the baseline (ie, TC is not applied), 1 to 5, and the measured execution time of each case is reported in FIGURE 15. As seen in the figure, the execution time continuously decreases as the number of variables to which TC is applied increases. As a result, when TC is applied to the five variables, its execution time is greatly shortened compared to the baseline, achieving a performance improvement of about 37%.

D. COMPARISON WITH THE OTHER APPROACHES

We evaluate how close the performance improvement of the proposed TC is to that of a multi-core platform using CCL.

TABLE 4. Comparison of quad-core platforms with different coherency schemes implemented on the FPGA.

Approach	HW	Practical-SW	TC
Shablack variable	Cached	Not-cached	Temporarily
Requiblack HW (Availability)	CCL (limited)	-	TCU (Provided)
Platform dev. difficulty	Hard	Easy	Easy
Requiblack SW (Availability)	-	-	API (Provided)
Application dev. difficulty	Easy	Easy	Medium
LUTs	48,698	38,936 (80.0%)	43,360 (89.0%)
FFs	18,663	16,060 (86.1%)	19,704 (105.6%)
Application exec. time (sec)	0.992	1.818 (183.3%)	1.148 (115.8%)

To this end, we used the Rocket [35] cores, which fortunately also offers a 4-core version with a dedicated CCL along with a single-core version. The results of comparative analysis for the hardware-based approach using CCL, the software-based approach that does not allow shared variable cache at all, and the proposed TC are reported in TABLE 4 as HW, practical-SW, and TC, respectively. As can be seen from the table, the HW approach often presents a very high level of difficulty in developing the CCL directly, so only limited platform development is practically possible using only the few types of cores that come with the CCL. On the other hand, practical-SW and TC approaches have low platform development challenges, no matter which core is used to develop a multi-core platform. Looking at the difficulties of developing applications that work on the developed platform, the TC approach provides an easy-to-use API, but it is still difficult compared to the HW or practical-SW approach. Meanwhile, in FPGA prototyping, the hardware resource consumption results of CCL and TCU show that the HW approach requires more hardware resources than the TC approach. In addition, for performance comparison, the HW approach has the shortest application execution time as expected, but the TC approach is also close. Of course, the performance of these two is far better than the practical SW approach. In the end, when developing multicore platforms using different types of RISC-V cores, the proposed TC approach may be the general solution, as it is flexible and easy, and the developed platform has good performance.

VII. CONCLUSION

Considering that when developing multicore platforms using various RISC-V cores, it is difficult to implement the dedicated CCL within each platform, resulting in a serious performance degradation problem. As a solution of this problem, we proposed the TC, a method that improves the performance of a multicore platform by enabling caching of data that are definite to not be shared for a certain period of time. Through a sophisticated operation mechanism, the proposed TC achieves performance improvement of the multicore platform while preventing the problem that can occur when TC data and non-TC data are on the same cache-line that can cause a fatal system error. To implement the proposed TC, we developed TC API for programmers and TC dedicated hardware, TCU, for platform developers, and detailed descriptions of each implementation method were provided in this paper. Especially, since the TCU operates independently of the core, it is possible to develop a TC-enabled multicore platform no matter what RISC-V cores are used. In addition, we proposed a method of embedding and implementing TCU in NoC in a multicore platform to facilitate the convenience of platform developers. Finally, in order to verify the effectiveness of the proposed TC, we implemented a quad-core platform equipped with TCU on the FPGA, and developed a handwriting recognition application with TC applied as a testbench. Through experimental work, we demonstrated that by applying TC, the performance of a multicore platform can be improved up to about 37% compared to the performance of a platform without TC.

ACKNOWLEDGMENT

(Hyeonguk Jang and Kyuseung Han contributed equally to this work.)

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.

[2] *RISC-V*. Accessed: Feb. 23, 2020. [Online]. Available: https://riscv.org/

[3] S. Greengard, "Will RISC-V revolutionize computing?" *Commun. ACM*, vol. 63, no. 5, pp. 30–32, Apr. 2020.

[4] D. Patterson, "50 years of computer architecture: From the mainframe CPU to the domain-specific tpu and the open RISC-V instruction set," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 27–31.

[5] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–8.

[6] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, "Parallelism via multithreaded and multicore CPUs," *Computer*, vol. 43, no. 3, pp. 24–32, Mar. 2010.

[7] Y. Kanehagi, D. Umeda, A. Hayashi, K. Kimura, and H. Kasahara, "Parallelization of automotive engine control software on embedded multicore processor using OSCAR compiler," in *Proc. 16th IEEE COOL Chips*, Apr. 2013, pp. 1–3.

[8] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, "The celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, Mar./Apr. 2018.

[9] M. Strobel and M. Radetzki, "Design-time memory subsystem optimization for low-power multi-core embedded systems," in *Proc. IEEE 13th Int. Symp. Embedded Multicore/Many-Core Syst.-Chip (MCSoC)*, Oct. 2019, pp. 347–353.

[10] M. Wang, T. Ta, L. Cheng, and C. Batten, "Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 173–186.
[11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 169–180.
[12] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012.
[13] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence domain restriction on large scale systems," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, New York, NY, USA, Dec. 2015, pp. 686–698.
[14] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *Proc. 25th Euromicro Conf. Real-Time Syst.*, Jul. 2013, pp. 80–89.
[15] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable cache coherence for multi-core real-time systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2017, pp. 235–246.
[16] S. Li and D. Guo, "Cache coherence scheme for HCS-based CMP and its system reliability analysis," *IEEE Access*, vol. 5, pp. 7205–7215, 2017.
[17] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta, "Reliability-aware data placement for heterogeneous memory architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 583–595.
[18] A. Ros, M. E. Acacio, and J. M. Garcia, "DiCo-CMP: Efficient cache coherency in tiled CMP architectures," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2008, pp. 1–11.
[19] I.-C. Lin and J.-N. Chiou, "High-endurance hybrid cache design in CMP architecture with cache partitioning and access-aware policies," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 10, pp. 2149–2161, Oct. 2015.
[20] U. Milic, A. Rico, P. Carpenter, and A. Ramirez, "Sharing the instruction cache among lean cores on an asymmetric CMP for HPC applications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2017, pp. 3–12.
[21] G. G. Shahidi, "Chip power scaling in recent CMOS technology nodes," *IEEE Access*, vol. 7, pp. 851–856, 2019.
[22] M. Ansari, M. Pasandideh, J. Saber-Latibari, and A. Ejlali, "Meeting thermal safe power in fault-tolerant heterogeneous embedded systems," *IEEE Embedded Syst. Lett.*, vol. 12, no. 1, pp. 29–32, Mar. 2020.
[23] S. Chakraborty and H. K. Kapoor, "Exploring the role of large centralised caches in thermal efficient chip design," *ACM Trans. Design Autom. Electron. Syst.*, vol. 24, no. 5, pp. 1–28, Oct. 2019.
[24] M. Rapp, M. Sagi, A. Pathania, A. Herkersdorf, and J. Henkel, "Power- and cache-aware task mapping with dynamic power budgeting for many-cores," *IEEE Trans. Comput.*, vol. 69, no. 1, pp. 1–13, Jan. 2020.
[25] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Oct. 2011, pp. 155–166.
[26] J. Cai and A. Shrivastava, "Software coherence management on non-coherent cache multi-cores," in *Proc. 29th Int. Conf. VLSI Design, 15th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2016, pp. 397–402.
[27] K. Han, J.-J. Lee, and W. Lee, "Converting interfaces on application-specific network-on-chip," *J. Semicond. Technol. Sci.*, vol. 17, no. 4, pp. 505–513, Aug. 2017.
[28] H. Jang, K. Han, S. Lee, J.-J. Lee, and W. Lee, "MMNoC: Embedding memory management units into network-on-chip for lightweight embedded systems," *IEEE Access*, vol. 7, pp. 80011–80019, 2019.
[29] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor, "BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, Jul. 2020.
[30] L. Chen, D. Zhu, M. Pedram, and T. M. Pinkston, "Power punch: Towards non-blocking power-gating of NoC routers," in *Proc. HPCA*, 2015, pp. 378–389.
[31] K. Han, J.-J. Lee, J. Lee, W. Lee, and M. Pedram, "TEI-NoC: Optimizing ultralow power NoCs exploiting the temperature effect inversion," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 2, pp. 458–471, Feb. 2018.
[32] K. Han, S. Lee, J.-J. Lee, W. Lee, and M. Pedram, "TIP: A temperature effect inversion-aware ultra-low power system-on-chip platform," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Design (ISLPED)*, Jul. 2019, pp. 1–6.
[33] M. Schoeberl, L. Pezzarossa, and J. Sparsø, "A minimal network interface for a simple network-on-chip," in *Architecture of Computing Systems*. Cham, Switzerland: Springer, 2019, pp. 295–307.
[34] K. Han, S. Lee, K.-I. Oh, Y. Bae, H. Jang, J.-J. Lee, W. Lee, and M. Pedram, "Developing TEI-aware ultralow-power SoC platforms for IoT end nodes," *IEEE Internet Things J.*, vol. 8, no. 6, pp. 4642–4656, Mar. 2021.
[35] K. Asanović *et al.*, "The rocket chip generator," Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[36] Xilinx. *Vivado 2016.4*. Accessed: Feb. 23, 2020. [Online]. Available: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2016-4.html
[37] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2012, pp. 3642–3649.
[38] B. Hutchinson, L. Deng, and D. Yu, "Tensor deep stacking networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1944–1957, Aug. 2013.
[39] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[40] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, Savannah, GA, USA, Nov. 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

HYEONGUK JANG received the B.S. and M.S. degrees in electrical engineering from Gyeongsang National University, Jinju, South Korea, in 2013 and 2015, respectively. He is currently pursuing the Ph.D. degree with the University of Science and Technology. He has been with the SoC Design Research Group, Electronics and Telecommunications Research Institute. His research interests include network-on-chip and system software in embedded systems.

KYUSEUNG HAN (Member, IEEE) received the B.S. and Ph.D. degrees in electrical engineering and computer science from Seoul National University (SNU), Seoul, South Korea, in 2008 and 2013, respectively. At SNU, he researched on computer architecture and design automation. Since 2014, he has been working with the Electronics and Telecommunications Research Institute (ETRI), Daejeon, South Korea. He currently belongs to the SoC Design Research Group as a Senior Researcher. His current research interests include reconfigurable architecture, network-on-chip, and ultra-low-power techniques in embedded systems.

SUKHO LEE received the Ph.D. degree in information communications engineering from Chungnam National University, Daejeon, South Korea, in 2010. He is currently a Principal Researcher with the SoC Design Research Group, Electronics and Telecommunications Research Institute, Daejeon. His current research interests include ultra-low-power system-on-chip design, embedded system design, video codec design, and video image processing.

JAE-JIN LEE received the B.S., M.S., and Ph.D. degrees in computer engineering from Chungbuk National University, in 2000, 2003, and 2007, respectively. He is currently a Group Leader with the SoC Design Research Group, Electronics and Telecommunications Research Institute, and also a Professor with the Department of ICT, University of Science and Technology. His research interests include processor and compiler designs in ultra-low-power embedded systems.

SEUNG-YEONG LEE (Student Member, IEEE) received the B.S. degree from Chung-Ang University, Seoul, South Korea, in 2020, where he is currently pursuing the M.S. degree in electrical and electronics engineering. He is currently a Beneficiary Student of the High-Potential Individuals Global Training Program. His research interests include low-power design, SoC architecture, and embedded systems.

JAE-HYOUNG LEE (Student Member, IEEE) received the B.S. degree from Myoungji University, Yong-In, South Korea, in 2020. He is currently pursuing the M.S. degree in electrical and electronics engineering with Chung-Ang University. He is currently a Beneficiary Student of the High-Potential Individuals Global Training Program. His research interests include low-power design, SoC architecture, and embedded systems.

WOOJOO LEE (Member, IEEE) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2007, and the M.S. and Ph.D. degrees in electrical engineering from the University of Southern California, Los Angeles, CA, USA, in 2010 and 2015, respectively. He was with the Electronics and Telecommunications Research Institute, from 2015 to 2016, as a Senior Researcher with the SoC Design Research Group, Department of Electrical Engineering, Myongji University, from 2017 to 2018, as an Assistant Professor. He is currently an Assistant Professor with the School of Electrical and Electronics Engineering, Chung-Ang University, Seoul. His research interests include ultra-low-power VLSI and SoC designs, embedded system designs, and system-level power and thermal management.