Design and Analysis of a Many-Core Processor Architecture for Multimedia Applications

Jyu-Yuan Lai*, Po-Yu Chen*, Ting-Shuo Hsu†, Chih-Tsun Huang*, and Jing-Jia Liou†
* Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
E-mail: cthuang@cs.nthu.edu.tw
† Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan
E-mail: jjliou@ee.nthu.edu.tw

***Abstract*—We present a design of many-core processor architecture with superior cost-effectiveness to fulfill the rapid increasing demand of high-speed embedded multimedia applications. The prototype platform consists of sixteen processor cores and a 4-by-4 mesh-based duplex network interconnection with external memory. The hardware and software interface in a bare-metal environment, i.e., without an Operating System (OS), has been emphasized in our architecture. An on-chip communication library is developed for practical parallel applications. In addition, we propose two memory-based file handling approaches to manipulate files with the lack of file-system support by OS. Our file handling approach can effectively reduce the minimum requirement of local memory without page swapping for each core from 4 MB to 64 KB in a case study of JPEG encoding. Furthermore, the analysis of instruction and data caches is addressed for the trade-off between area and speed. The experimental result indicates that our many-core platform with its application development infrastructure is efficient in delivering cost-effective multimedia applications in a bare-metal environment.**

I. Introduction

With the increasing demand of high-performance multimedia applications in embedded systems, the emerging many-core processor architecture has been regarded as an attractive alternative to traditional centralized monolithic processor design, thanks to the ever shrinking process technology. Instead of stepping up the operating frequency, many-core processor architectures take the advantage of parallelism to achieve the throughput requirement with superior cost-effectiveness. In addition, Network-on-Chip (NoC), which adopts flexible and scalable packet-switched interconnection to replace conventional shared buses, is widely used in modern many-core processors to maintain efficient communication among the Processing Elements (PEs).

There have been numerous many-core processors published recently [1]–[8]. In [1], [2], the Raw microprocessor was built by replicating small and simple tiles. Each tile contains a RISC-like pipelined core which is tightly integrated with a programmable switch. The network interface is register mapped. And inter-tile communication is optimized as register access. In [3], the Asynchronous Array of simple Processors (AsAP) was geared to handle the task-level parallelism inherent in modern DSP applications. Because DSP applications usually exhibit fixed communication patterns between internal tasks, circuit-switched network with sophisticated task placement is adopted to manipulate the task communication. The program of each task can be written individually and linked either manually or using the AsAP tool chain. Moreover, the AsAP was extended to a many-core computational platform in [4] which is composed of 164 homogeneous processor cores and three dedicated-purpose accelerators. The TILE64 in [5] consists of 64 tile processors connected through mesh networks. The networks support five independent and distinct functions. Three out of the five networks are software-visible. The tile processors access the software-visible networks by using register mapped mechanism to reduce routing latency. In [6], the specialized Semi-Dynamic Flip-Flops (SDFFs) were utilized as critical registers in the 80-tile processor. Bisection bandwidth of the on-chip network can achieve 2 Terabits/s at the clock rate of 4 GHz. In [7], the 48-core processor was divided to 8 voltage and 28 frequency islands to support Dynamic Voltage and Frequency Scaling (DVFS) for higher performance per watt. Message passing between cores in the 48-core processor was accomplished by using on-chip shared Message Passing Buffer (MPB) [8].

In summary, most of the previous works emphasized hardware-driven optimization techniques, e.g., register-mapped network interface in [1], [2], [5], the circuit-switched network in [3], the specialized registers in [6], and the advanced power management in [7]. However, a robust, flexible, and efficient design framework is essential to modern many-core systems targeting a variety of practical embedded applications. Although the programs in the Raw microprocessor [1], [2], the AsAP [3], and the 48-core processor [7], [8] can be written in high-level programming language, either customized compilers or specialized API is mandatory. In addition, to the best of our knowledge, none of the publications mentioned about the manipulation of files for embedded many-core processors in a bare-metal environment, which is the crucial requirement for practical multimedia applications.

In this paper, we present a many-core processor architecture together with software development infrastructure for practical applications. A prototype platform composed of sixteen PEs and one external memory is implemented. Software applications can be executed on our platform directly without an OS. Therefore, overall performance of our many-core platform can be maximized for specific embedded applications such as multimedia and image processing, eliminating the overhead introduced by OS.

However, software development in such a bare-metal environment is challenging. There is no OS to support sophisticated file systems. In addition, accessing the parallel hardware architecture via high-level programming environment is a burden. So taking full advantage of a many-core platform at application level is not intuitive. On top of our many-core processor architecture, we develop a specific parallel programming library to support both robust message passing between the PEs and bulk data movement between memories. In addition, two memory-based file handling approaches are also proposed particularly for multimedia applications in a bare-metal environment. Realistic parallel applications on the many-core platform can be achieved easily in high-level C programming language using standard GNU C compiler (i.e., *gcc*) and its tool chain.

By using the proposed external memory-based file handling approach, the minimum requirement of local memory without page swapping for each PE can be reduced by 98.44% as compared with that using local memory-based approach, and the parallel processor using 16 PEs is 13 times faster than the serial one, in a case study of JPEG encoding. Finally, the trade-off between area and speed of the prototype platform is explored systematically. The experimental result and analysis show that our many-core processor architecture with the proposed parallel programming library and file handling approaches is an effective solution for developing realistic parallel multimedia applications in a bare-metal environment.

II. The Proposed Many-Core Platform

Fig. 1 shows the prototype platform of our many-core processor architecture. There are sixteen homogeneous PEs, 4-by-4 mesh-based network request/response switches and links, and external memory. For the flexibility, we adopt the standardized Open Core Protocol (OCP) [9] interface for the interconnection between the PEs and the switches. Heterogeneous hardware accelerators can also be integrated to the prototype platform as long as the accelerators are wrapped with OCP interface. By taking the advantage of network-based interconnection, the architecture can be scalable to support more PEs and external memories. In addition, a variety of realistic multimedia applications, such as JPEG encoding and object tracking, can be executed on the proposed many-core platform.

To enhance the efficiency of verification and development, the prototype platform is implemented in two different abstraction levels, i.e., Electronic System Level (ESL) and Register Transfer Level (RTL). The ESL platform can be utilized for rapid performance and bottleneck analysis, as well as to be a reference model for verification. On the other hand, the RTL one can provide realistic physical design parameters such as critical path delay, hardware area, and power consumption, etc.

Our PE consists of a single microprocessor, a local memory, a communication unit (including a DMA (direct memory access) unit and a PE-to-PE unit), and a local system bus which connects the components in a PE. The local memory of each PE contains both program instructions and private data variables. Note that for RTL implementation, the local memory is realized by a single-port SRAM and wrapped with the local system bus interface. The DMA is responsible for moving bulk data between the local memory and the external memory. The PE-to-PE unit adopts point-to-point message-passing mechanism for inter-PE communication.

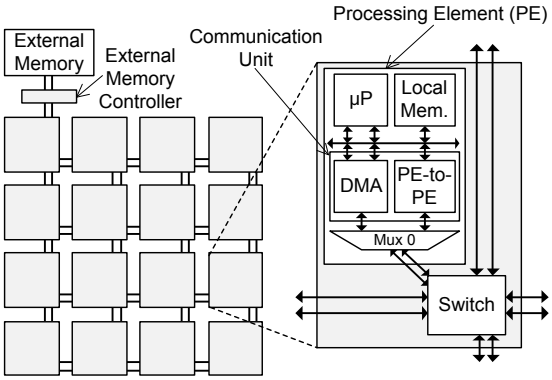

Fig. 1. The proposed many-core platform.

Note that the external memory in our platform is a shared memory for data storage and visible to all PEs. Each PE can access the external memory via the DMA unit. On the other hand, the local memory, which is the main memory of the microprocessor, is a private memory and only visible to the associated PE. A data variable in a parallel application (resided in the local memory of each PE) could have different value but the same address with respect to each PE. In addition, considering the address space in our platform, each PE accesses its own local memory starting from the address 0x0, and all PEs see the global external memory starting from the address 0x10000000.

A. The Microprocessor

Our microprocessor is based on OpenRISC CPU [10], because of its open source architecture and convenient GNU tool chain. The CPU is a five-stage pipelined 32-bit RISC architecture with separate instruction and data caches/MMUs (Memory Management Units). The sizes of instruction cache can be configured from 512 bytes to 32 KB, while the size of data cache ranges from 4 KB to 32 KB. Note that the cache line is 16-byte wide except the cache of 32 KB, which is 32-byte wide. All MMUs and caches can be removed to minimize the hardware area. The design exploration by using different sizes of instruction and data caches is addressed in Sec. IV in a case study of JPEG encoding.

B. The Proposed Communication Unit and Parallel Programming Library

Our communication unit comprises a DMA unit and a PE-to-PE unit. The DMA unit is based on the WISHBONE DMA IP core [11], and we replaced one of the interfaces with OCP master interface to connect with our network switch, as shown in Fig. 2(a). Each PE can read/write data from/to the external memory via its DMA unit to relieve the microprocessor from heavy memory routines. On the other hand, Fig. 2(b) shows the proposed PE-to-PE communication unit. There are four types of FIFOs in the PE-to-PE unit for our message-passing

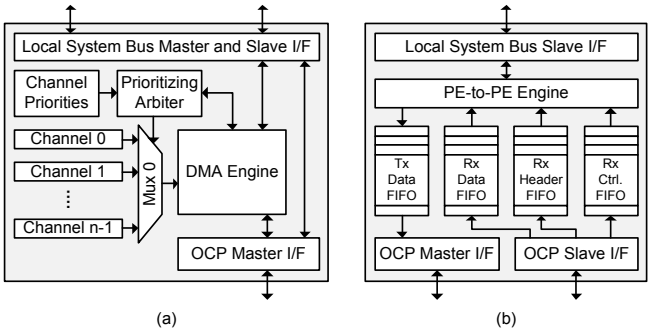

Fig. 2. (a) The DMA unit and (b) the PE-to-PE communication unit.

flow control protocol. Inter-PE message-passing is done by the PE-to-PE units in both sender and receiver PEs, instead of putting the message into the remote local memory of the receiver by the sender directly.

Both the DMA unit and the PE-to-PE unit can be programmed by using C-based primitive functions through memory-mapped I/O technique. To develop practical parallel applications, we build the parallel programming library to utilize the DMA unit and the PE-to-PE unit. By using the PE-to-PE unit, there are functions for point-to-point communication with and without robust flow control protocol between PEs. For example, $send()$ and $recv()$ are a pair of functions for transferring variable length of messages between two PEs with flow control protocol. In addition, utility functions for the DMA unit are also provided. As an example, the function $start\_dma()$, which is similar to the built-in function $memcpy()$, triggers the DMA unit to transfer bulk data from source address to destination address. The source and destination addresses can be both in the private local memory, both in the external memory, or one in the private local memory and the other in the external memory.

III. The Proposed File Handling Approaches in the Bare-Metal Many-Core Platform

Most of practical multimedia applications, such as image processing and 3D graphic rendering, involve frequently manipulation of massive files. Among the widespread file handling functions, $fopen()$, $fseek()$, $fread()$, and $fwrite()$ are the most indispensable functions for multimedia applications. However, the proposed many-core platform is a bare-metal application processor, i.e., there is no file-system support by OS. In addition, to implement a versatile file-system requires extra libraries, which increases the minimum requirement of local memory without page swapping for each PE. Note that once the minimum requirement of local memory without page swapping for each PE exceeds the size of local memory, either a larger local memory or extra timing overhead for page swapping is needed.

A. Local Memory-Based File Handling Approach

An eight-bit byte is usually the most atomic block of multimedia files. For example, a raw colored pixel is typically represented by three bytes such as Red (R), Green (G), and Blue (B), each color information is sampled by one byte.

```
1.  unsigned char lena512[] = {0xe9, 0x9a, 0x8f, ···
    ···, 0xc7, 0x5c, 0x64};
    ···
2.  int main(){
      ···
3.    char *ifile;
      ···
4.    ifile = lena512;
      ···
5.    return 0;
6.  }
```

Fig. 3. Open a simulated file by using the local memory-based approach.

```
1.  int main(){
      ···
2.    int seek_val, num;
      ···
3.    ifile += seek_val;
4.    memcpy((char *) dest, ifile, num);
      ···
5.    return 0;
6.  }
```

Fig. 4. Reposition the position indicator of a simulated file and read data from the simulated file by using the local memory-based approach.

Without file-system support by OS, we first decompose a multimedia file into an array of characters to represent the file. The preprocessing can be done by a host processor with OS.

By using our local memory-based file handling approach, we embed the resultant array in the program. As shown in Fig. 3, the array $lena512$ embodies a simulated file in a bare-metal environment. There can be multiple simulated files in a program. In addition, we substitute a pointer to a character for the pointer to a file. To open a simulated file, we just assign the address of the simulated file, i.e., the address of the first byte of the array, to the character pointer. As an example, in Fig. 3, we open a simulated file $lena512$ by assigning the address of $lena512$ to the character pointer $ifile$.

For parallel multimedia applications, to adjust the position indicator of an opened file, i.e., the function $fseek()$, is necessary for data partition among multiple processor cores. Since we use a character pointer to locate the simulated file, to reposition the position indicator can be accomplished by simply modifying the value of the character pointer, e.g., the line 3 of Fig. 4. In addition, the built-in function $memcpy()$ is used to replace the function $fread()$. As an example, in the line 4 of Fig. 4, $memcpy()$ copies the value of num bytes from the current position indicator of the simulated file $ifile$ to a certain memory block pointed by $dest$ in the local memory.

To write data to a simulated file, we first specify a reserved address for the output simulated file. The address is also located in the local memory. As shown in Fig. 5, the address OUT_FILE_BASE is reserved for an output simulated file. Again, there can be multiple output simulated files in the program, we use a pointer to a character to access the output simulated file (see line 4 of Fig. 5). We also utilize the function $memcpy()$ to move data to the position of the output simulated file. In addition, we use the function $sim\_file\_end()$ to set

```
1.  #define OUT_FILE_BASE    0x00300000
    ...
2.  int main(){
      ...
3.    char *ofile;
      ...
4.    ofile = (char *) OUT_FILE_BASE;
      ...
5.    memcpy(ofile, (char *) src, num);
6.    sim_file_end(OUT_FILE_BASE+num);
      ...
7.    return 0;
8.  }
```

Fig. 5. Write data to a simulated file by using the local memory-based approach.

```
1.  #define EXT_MEM_LENA512_BASE    0x10200000
    ...
2.  int main(){
      ...
3.    char *ifile;
4.    int seek_val, num;
      ...
5.    ifile = (char *) EXT_MEM_LENA512_BASE;
      ...
6.    ifile += seek_val;
7.    start_dma((char *) dest, ifile, num);
      // Transfers data of num bytes from ifile to dest
      ...
8.    return 0;
9.  }
```

Fig. 6. Open a simulated file, reposition the position indicator, and read data from the simulated file by using the external memory-based approach.

a special register in the PE for the address of the last byte of the output simulated file.

At the end of a parallel multimedia application, the host processor reads the special register in each PE to know the size of the output simulated file of each PE. The host processor then dumps fragments of the output simulated file in the local memories to form the resultant file.

The local memory-based file handling approach straightforwardly imitates the original file handling functions. This approach is simple and easy to implement. But this approach embeds the input files in the program, which will increases the minimum requirement of local memory without page swapping. In addition, by using the local memory-based approach, the input files can only be configured at compile-time, which limits the flexibility of the program.

B. External Memory-Based File Handling Approach

Instead of embedding input simulated files into the program, our second approach puts the files into the external memory. Only starting addresses of the input simulated files are specified in the program, e.g., the address $EXT\_MEM\_LENA512\_BASE$ in Fig. 6. Content of the files only needs to be placed into the external memory by the host processor before the require time of individual files. The manipulation of a pointer to a character is also used to realize linking to a file and modifying the position indicator of an opened file in a bare-metal environment (see line 5 and line 6 of Fig. 6).

Different from the local memory-based approach, the external memory-based file handling approach adopts the proposed DMA utility to read/write data from/to a remote file in the external memory. As shown in Fig. 6 and Fig. 7, the function $start\_dma()$ is utilized to substitute for both $fread()$ and $fwrite()$. Since there is only one external memory in current prototype platform, the PEs take turns to access the external memory, which will lead to extra performance overhead. In case of adopting multiple external memories, data partition among the memories should be handled by the host processor in advance.

Because the external memory is shared to all PEs, race condition may occur when multiple PEs intend to write data to the same output simulated file in the external memory. To guarantee every block of the file will not be modified by

```
1.  #define EXT_MEM_OUT_FILE_BASE    0x10300000
2.  #define EXT_MEM_OUT_FILE_END     0x10100000
    ...
3.  int main(){
      ...
4.    char *ofile;
5.    int ostart, num;
6.    if(ID == FIRST_CORE)
7.      ostart = EXT_MEM_OUT_FILE_BASE;
      ...
8.    if(ID != FIRST_CORE)
9.      recv(ID-1, &ostart, 4);
        // Gets data of 4 bytes from prior core and stores the data to ostart
10.   ofile = (char *) ostart;
      ...
11.   start_dma((char *) ofile, (char *) src, num);
12.   ostart += num;
13.   if(ID != LAST_CORE)
14.     send(ID+1, &ostart, 4);
        // Sends data of 4 bytes from ostart to next core
15.   else
16.     start_dma((char *) EXT_MEM_OUT_FILE_END, &ostart, 4);
      ...
17.   return 0;
18. }
```

Fig. 7. Write data to a simulated file by using the external memory-based approach.

more than one PE, we coordinate the PEs by passing the position indicator of the file between the PEs. The position indicator represents the address of the last valid byte of the output simulated file in the external memory at present. As shown in Fig. 7, the first PE starts with a specified beginning address of the output simulated file (see line 6 and line 7). Except the first PE, every PE cannot write data to the simulated file in the external memory until the PE receives the position indicator passed from its previous neighbor PE (see line 8 to line 11). Note that word alignment needs to be manipulated for word-based DMA unit when writing data to external memory, because the position indicator is not always an integral multiple of four bytes (i.e., one word). After writing data to the simulated file, the PE (except the last one) sends the current position indicator of the file to its adjacent PE. At the end, the last PE sends the address of the last byte of the file to the external memory for the host processor, e.g., line 16 in Fig. 7. Also, the host processor dumps the output simulated

file from the external memory to form the resultant file.

Compared with the local memory-based file handling approach, the external memory-based approach is more complicated. There will be additional timing overhead for coordinating the PEs and moving data between memories. However, by using the external memory-based approach, the input simulated files can be configured at run-time to improve the flexibility of the applications. In addition, the minimum requirement of local memory without page swapping can be significantly reduced.

IV. Design Analysis and Exploration

The prototype platform of our many-core architecture has been implemented in both Verilog HDL (RTL) and SystemC (ESL). The RTL one was synthesized by using TSMC 0.13-μm CMOS technology with operating frequency of 100MHz. Realistic parallel multimedia applications, such as JPEG encoding and object tracking, can be evaluated in our platform. In this section, the design exploration among area and speed is made in a case study of JPEG encoding. Because JPEG encoding is a typical and extensively used multimedia application. We assume the external memory is an off-chip/off-die memory, area of the external memory is excluded in our analysis. The characteristics of our many-core platform are extracted from the ASIC synthesis and simulation results. For the simplification, operation cycles discussed in the exploration do not include those for preparing input simulated file and dumping output simulated file by the host processor.

Table I compares the proposed two memory-based file handling approaches for JPEG encoding with one, two, four, eight, and sixteen working PEs. The third and the fifth columns denote the minimum requirement of local memory without page swapping for each PE. Different from the task-level parallelism in [3], we adopt data-level parallelism for JPEG encoding in our platform. A raw bitmap picture is partitioned into several independent parts, and these parts are concurrently manipulated by the PEs. The benchmark is a raw bitmap picture of 512×512 pixels. Considering the serial execution with one working PE, the number of cycles by using external memory-based approach is about 12.84% more than the one using local memory-based approach, because the overhead of moving data between the external memory and the PEs. Moreover, because the need of coordinating PEs to avoid race condition when writing data to external memory, the speed overhead of parallel execution using external memory-based approach is increased to between 14.17% and 15.38%.

However, the local memory-based file handling approach reserves memory blocks of about 3 MB in program to store input and output simulated files. In addition, all of the intermediate data during JPEG encoding are also stored in the local memory. Accordingly, each PE requires local memory of at least 4 MB in case of no page swapping. On the other hand, the external memory-based file handling approach keeps input and output simulated files in the external memory. Only a macroblock of 16×16 pixels (i.e., 768 bytes) is fetched into the local memory at a time. Intermediate data are also sent back to the external memory. The minimum requirement of local memory without page swapping for each PE by using the external memory-based approach can therefore be significantly reduced by about 98.44% as compared with that using local memory-based approach.

TABLE I
THE CYCLES AND MINIMUM REQUIREMENT OF LOCAL MEMORY WITHOUT PAGE SWAPPING FOR JPEG ENCODING BY USING PROPOSED LOCAL AND EXTERNAL MEMORY-BASED FILE HANDLING APPROACHES.

Number of Working PE	Local Mem.-Based		External Mem.-Based		Speed Overhead
	Cycles* (A)	Local Mem.†	Cycles* (B)	Local Mem.†	($\frac{B-A}{A} \times 100\%$)
1	83.31M	4 MB	94.01M	64 KB	12.84%
2	42.41M		48.42M		14.17%
4	21.73M		24.99M		14.97%
8	11.42M		13.17M		15.38%
16	6.29M		7.23M		14.94%

*No page swapping is occurred; †The minimum requirement of local memory without page swapping for each PE.

In addition, compared with the serial one, the parallel processor using 16 PEs with local memory-based file handling approach is 13.24 times faster in terms of cycles, and 13.00 times faster with external memory-based approach, respectively. The overhead of coordinating PEs and the external memory degrades the benefit of parallelism a little, but reduces the minimum requirement of local memory without page swapping for each PE from 4 MB to 64 KB in return. The comparison indicates that the external memory-based approach is effective to the many-core processor architecture with realistic multimedia applications in a bare-metal environment.

The operation cycles in Table I are obtained by configuring the microprocessor in each PE with instruction cache of 4 KB and without data cache. To further analyze the effectiveness of instruction and data caches, we explore the design space by using different sizes of instruction and data caches. In this work, we focus on the design space of parallel processors using 16 PEs with external memory-based file handling approach.

Fig. 8 plots the operation cycles of the parallel processors with different sizes of instruction cache for JPEG encoding. To stress the effectiveness of instruction cache, data cache is removed in this experiment. By using an instruction cache of 512 bytes, the operation cycles can be reduced by about 51.34% compared with that without cache. The number of cycles can be further reduced by increasing the size of instruction cache. However, configurations with instruction cache more than 8 KB cannot obtain more improvement.

The JPEG encoding iteratively performs a sequence of instructions, e.g., fetching a macroblock, converting the representation of color, DCT (Discrete Cosine Transform), quantization, and Huffman encoding. Using instruction cache can effectively reduce the number of cycles by storing those instructions in the cache. However, in our experiment, the size of the instructions iteratively used for JPEG encoding is about 6 KB. Using instruction cache more than 8 KB therefore cannot improve the performance further.

On the other hand, we also analyze the effectiveness of data cache for JPEG encoding. The operation cycles of the parallel processors with different sizes of data cache is shown

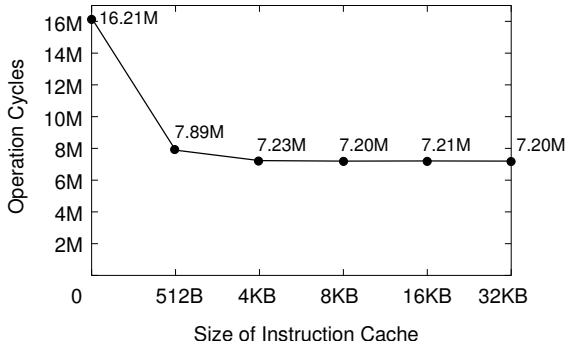

Fig. 8. The operation cycles of the parallel processors using 16 PEs with different sizes of instruction cache (without data cache) for JPEG encoding.

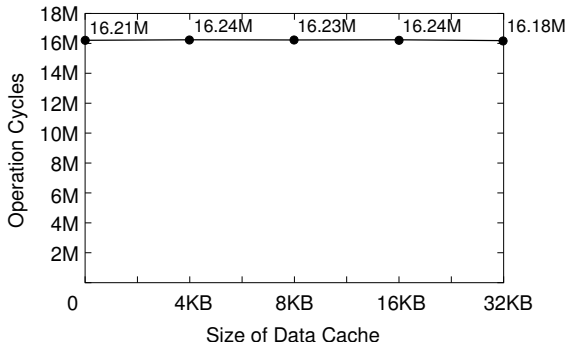

Fig. 9. The operation cycles of the parallel processors using 16 PEs with different sizes of data cache (without instruction cache) for JPEG encoding.

in Fig. 9. Similar to the above experiment, instruction cache is removed to focus on data cache. With data cache, the microprocessor tends to store copies of used data into the cache for future reference. However, as we mentioned before, the JPEG encoding iteratively fetches a new macroblock from external memory to local memory by utilizing the DMA unit. To guarantee the consistency between data cache and local memory, it involves extra overhead to invalidate the corresponding cache blocks. Consequently, using data cache for JPEG encoding does not always improve performance. The cache line in data cache of 32 KB is two times the size as compared with other configurations, which can reduce the overhead of ensuring data consistency.

The AT comparison of parallel processors with different sizes of instruction and data caches for JPEG encoding is shown in Fig. 10. All the AT products are normalized to the smallest one, i.e., AT of the configuration with instruction cache of 4 KB and without data cache. It is obvious that the processors without data cache are more AT efficient. The exploration of the trade-off between area and speed indicates that adopting more and larger caches does not always guarantee the improvement. Designers should optimize the system from the viewpoints of hardware and software altogether.

V. Conclusions

We have presented a design framework to accomplish realistic parallel multimedia applications in a bare-metal environment. The design framework consists of a many-core processor architecture, a parallel programming library, and two memory-based file handling approaches (i.e., local memory-based and external memory-based, respectively). A prototype many-core platform using network-based interconnection has been implemented with sixteen PEs and one external memory. The proposed external memory-based file handling approach can reduce the minimum requirement of local memory without page swapping for each PE by 98.44% with affordable speed overhead as compared with the local memory-based approach in a case study of JPEG encoding. The cost-effectiveness trade-off among a variety of design configurations has been intensively evaluated systematically. The implementation result and analysis justify the effectiveness of our design framework for practical multimedia applications in a bare-metal many-core environment.

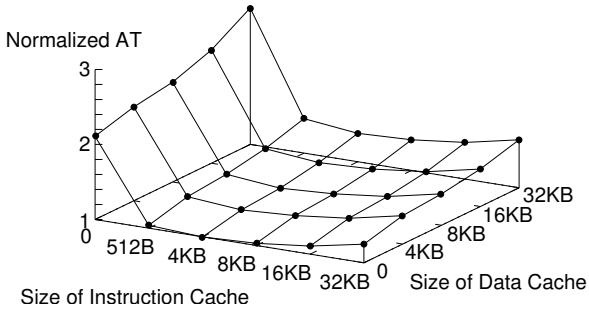

Fig. 10. AT comparison of the parallel processors using 16 PEs with different sizes of instruction and data caches for JPEG encoding.

Acknowledgement

This work was supported in part by National Science Council, Taiwan, under Contract NSC 100-2628-E-007-009 and NSC 101-2220-E-007-026 and by Industrial Technology Research Institute of Taiwan, ROC.

References

[1] E. Waingold *et al.*, "Baring it all to software: Raw machines," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, Sept. 1997.

[2] M. B. Taylor *et al.*, "The Raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar. 2002.

[3] B. Baas *et al.*, "AsAP: A fine-grained many-core platform for DSP applications," *IEEE Micro*, vol. 27, no. 2, pp. 34–45, Mar. 2007.

[4] D. N. Truong *et al.*, "A 167-processor computational platform in 65 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, pp. 1130–1144, Apr. 2009.

[5] S. Bell *et al.*, "TILE64^{TM}-processor: A 64-core SoC with mesh interconnect," in *Proc. IEEE Int. Solid-State Cir. Conf. (ISSCC)*, San Francisco, Feb. 2008, pp. 88–598.

[6] S. R. Vangal *et al.*, "An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, Jan. 2008.

[7] J. Howard *et al.*, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[8] T. G. Mattson *et al.*, "The 48-core SCC processor: the programmer's view," in *Proc. Int. Conf. High Performance Computing, Storage and Analysis (SC)*, Nov. 2010, pp. 1–11.

[9] OCP-IP, *Open Core Protocol Release 2.2*, OCP-IP Association, Redwood City, Jan. 2007.

[10] OpenCores, "OpenRISC 1000 architecture manual," http://opencores.org/or1k/Main_Page, June 2011.

[11] OpenCores, "WISHBONE DMA/bridge IP core," http://opencores.org/project,wb_dma, Mar. 2009.