An Efficient Programming Paradigm for Shared-Memory Master-Worker Video Decoding on TILE64 Many-Core Platform

Xuan-Yi Lin[*] Kuan-Chou Lai[†] Shau-Yin Tseng[§] Kuan-Ching Li[‡] Yeh-Ching Chung[*]

[*] Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
{xylin, ychung}@cs.nthu.edu.tw

[†] Department of Computer Science and Information Science
National Taichung University of Education
Taichung, Taiwan
kclai@mail.ntcu.edu.tw

[§] Information & Communications Research Laboratories
Industrial Technology Research Institute
Hsinchu, Taiwan
tseng@itri.org.tw

[‡] Department of Computer Science and Information Engineering
Providence University
Taichung, Taiwan
kuancli@pu.edu.tw

***Abstract*— The ubiquity of many-core architectures brings challenges in making scalable application software, changing dramatically from the way applications are traditionally developed. Optimization of programs for many-core platforms is a multifaceted problem, where system and architectural factors should be taken into consideration. In this paper, we attack the problem on the aspect of programming paradigm. We propose a hybrid *producer-write plus consumer-read* shared-memory programming paradigm for implementation of a master-worker video decoder on the TILE64 many-core platform. To evaluate the scalability and performance benefits of different programing paradigms, a Motion JPEG decoder is parallelized using master-worker structure and implemented with combinations of *consumer-read programming* and *producer-write programming*. Experimental results show that the proposed implementation obtained competitive performance speedup, scaling well with number of available cores and up to 4 times performance improvement over other implementations on the decoding of a 1080P video.**

Keywords-many-core; producer-consumer; master-worker; shared memory; programming paradigm; TILE64

I. Introduction

With rapid industry development of many-core architectures, mass-produced processors now contain tens to hundreds of cores in a single chip [1]. While the trend of processor manufacturing is to increase the number of cores rather than clock frequency [2, 3], software developers can no longer rely on the so called "free lunch" [4] that automatically makes existing programs run faster on processors clocked at higher frequencies.

In order to make performance of a program scale well with the number of available cores on a many-core platform, existing software needs to be modified or re-written from ground up [5, 6, 7, 8, 9]. Efforts involving parallelization of an application are twofold, known as *design* and *implementation*. The former is about finding concurrency in the given application and to derive algorithms and program structures to make it run faster, while the latter is about utilization of available programming resources on the designated parallel platform to realize the designed algorithm and structure. The available programming resources include programming language, programming paradigm, APIs, among others.

Due to the flexibility of available options, there may be possible multiple implementations for a single design, so performance and scalability characteristics of completed applications may vary with different implementations. Thus, it is important to set guidelines for developers to follow in order to produce better programs on a given platform. The purpose of this paper is to discuss and demonstrate how programming paradigm correlates with issues in performance and scalability of software implementations on a many-core platform.

Master-worker structure is often adopted as design of an application when there is need to dynamically balance workloads among a set of available processors [10, 11]. There are two parts in a master-worker system where communications take place between master and worker processes. The former is *task distribution* and the latter is *result collection*. In the task distribution part, master process generates a set of workloads and distributes tasks to worker processes, here the master process can be seen as a *producer* process and worker processes can be seen as *consumer* processes. In the result collection part, the master process collects computation results made by worker processes, here the worker processes can be seen as *producer* processes and the master process can be seen as a *consumer* process. Efficient handling of the communications between master and worker processes is required to develop a high-performance system.

TILE64 is a family of general purpose many-core processors [12], containing 64 identical cores connected by an on-chip network. In their publication [13], Tilera suggests that programmers can implement applications in a way such that producer processes always write data directly into memory addresses shared by consumer processes to avoid unnecessary cache coherent traffics on the memory network.

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.65

There are also literatures discussing scalability issues on many-core processors featuring on-chip networks or multiple memory controllers [14, 15, 16]. In our previous work [17], we have shown that it is necessary to consider the memory hierarchy and on-chip networks in order to develop high performance applications on the TILE64 platform. We have also shown that program performance and scalability can be very different between two implementations of an equivalent functionality. The problem is how to choose better implementation options without going through a time-consuming trial and error sessions.

In this paper, we further explore the problem by defining two different styles of programming paradigms, *consumer-read programming* and *producer-write programming*, and to propose a hybrid *producer-write plus consumer-read* shared-memory programming paradigm for implementation of a master-worker video stream decoder on the TILE64 many-core platform. We implement task distribution and result collection in the master-worker system with combinations of producer-write programming and consumer-read programming. Experimental results show that for a Motion JPEG decoder, implementation based on *producer-write* task distribution and *consumer-read* result collection exhibits best performance and scalability for all given workloads with different video frame sizes. When decoding a 1080P video stream, the hybrid *producer-write plus consumer-read* decoder runs up to 4 times faster compared to other implementations.

This paper brings the following contributions. It identifies two shared-memory programming paradigms for a many-core platform, *consumer-read programming* (CRP) and *producer-write programming* (PWP), that shows the way a master-worker stream processing system can be implemented using CRP and PWP, as also detailed performance comparisons between implementations of a master-worker video decoder using CRP and PWP and suggests that the hybrid *producer-write plus consumer-read* paradigm best suits this application on the TILE64 platform.

The rest of this paper is organized as follows. Section II provides background knowledge of TILE64 processor architecture and the basics of how to implement shared-memory communication between two processes on TILE64. In Section III, a master-worker stream processing system is described. Section IV introduces the CRP and PWP and variations of shared-memory implementations of a master-worker stream processing system. In Section V, we implement a parallel Motion JPEG decoder with proposed programming paradigms and compare performance of the implementations. Concluding remarks of this work are given in Section VI.

II. PRELIMINARIES

A. The TILE64 Processor

The TILE64 processor is a 64-core many-core processor featured as an array of 64 identical processor cores (each referred to as a *tile*) interconnected via on-chip two-dimensional mesh networks [18]. The TILE64 is fully programmable using standard ANSI C under Linux environment, including a set of proprietary APIs called *iLib*. The iLib library supports two communication mechanisms, shared memory and distributed memory, for processes running on different cores to communicate with each other. So, software developers can make use of both communication primitives in an application program. In this paper, when we refer to a process, we mean a process that is bound to and running on a *tile*. A *tile* runs one process at any given time. A process bound to a *tile* at the initialization period will keep running on the same *tile* to its end of life. This fashion is similar to the execution of MPI programs.

Fig. 1 illustrates the architecture overview of a TILE64 processor. There are four memory controllers located at the four corners of a processor array, providing accesses to an external memory system that is accessible by all tiles. The interface to on-chip memory networks provides access both to L2 caches of other tiles and to external memory.

B. Shared Memory Communication on TILE64

Shared memory communication allows each process in a parallel application to load/store values from/to a globally visible region of memory. Each process in the application can access any object in shared memory at any time. Access to shared memory objects must be synchronized to prevent inconsistent states [19]. Data inconsistencies happen when multiple processes are storing values to identical memory address at the same time without proper synchronization.

Both the Linux and iLib programming environments provide tools for allocating and synchronizing accesses to the shared memory. Linux allows programs to allocate and synchronize using the standard Unix shared memory and pthreads APIs, while iLib supports a special function for shared memory allocation, *malloc_shard*() as well as an implementation of a pthreads-style mutex lock. To use iLib to implement shared memory mechanisms in a program, the process which shares information can call the *malloc_shard*() function to get an address pointing to a block of shared memory. Then the process notifies other processes the location of shared memory by sending them messages containing this address.

Fig. 2 shows an example on the use of iLib to create an

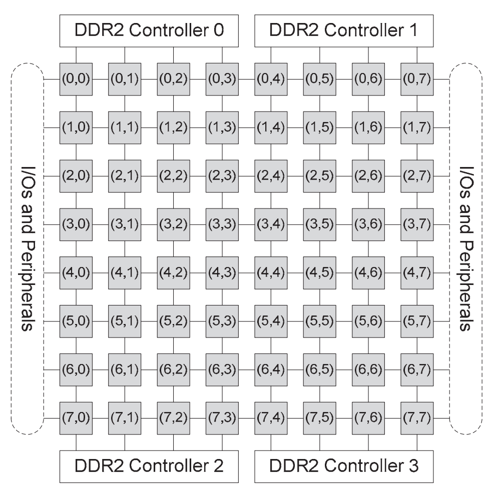


Figure 1. TILE64 processor architecture overview.

integer object shared between 2 processes, while Fig. 3 depicts the corresponding codes within processes 0 and 1.

- There are two cores, each of which executes one process,
- Process 0 allocates a region of memory to hold one integer using *malloc_shared*(),
- The *malloc_shared*() function returns a value x, which is the address of the shared integer. The value of x is stored in an integer pointer p in process 0,
- Process 0 sends content of p to process 1,
- Process 1 stores this address with integer pointer q.

After above initialization process, both processes 0 and 1 will be able to load from and store to this shared integer in the same way as normal variables. Any update to $*q$ made by process 1 can be seen by process 0 using $*p$, and back and forth is also valid.

Because the *malloc_shared*() function is called by process 0, the shared memory region starting at x is said to be *homed* on core 0.

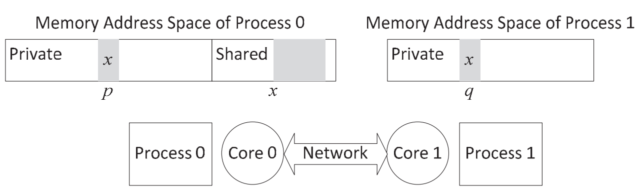


Figure 2. Sharing of an integer between two processes.

Process 0

```
int *p;
p=(int *)malloc_shared(sizeof (int));
ilib_msg_send(GROUP,            /* group */
              1,                /* rank */
              MESSAGE_TAG,      /* tag */
              &p,               /* buffer */
              sizeof(p));       /* size */
```

Process 1

```
int *q;
ilib_msg_receive(GROUP,         /* group */
                 0,             /* rank */
                 MESSAGE_TAG,   /* tag */
                 &q,            /* buffer */
                 sizeof(q),     /* size */
                 &status);      /* status */
```

Figure 3. Code snippets of process 0 and process 1 to create a shared integer.

III. MASTER-WORKER STREAM PROCESSING

A stream processing application is a program that takes a data stream as input, performs operations upon that input stream and then outputs another processed data stream [20, 21, 22, 23, 24]. Data streams might carry any kind of information, making there a huge diversity between stream processing applications. Video stream processing applications refer to those which data streams are used to carry video data. Some examples of such applications are video encoders, decoders, and transcoders. These applications transform video streams from one format to another. Other examples of video stream processing applications are image processing and pattern recognition ones, such as video labeling, object detection and object tracking applications. These applications retrieve information from input video streams then attach the information to output video streams.

Given a data stream to be processed by a stream processing application, assume that the stream can be divided into n sequenced fragments that can be independently processed and outputted. The input data stream can be represented as a set of sequenced data items, fi_1 to fi_n, and the output data stream is represented as fo_1 to fo_n. Assume that the application is run on a processor, each fragment takes time t_n to be processed from input format to output format. The total time needs to process all fragments in the stream would be:

$$\sum_1^n t_i$$

The ideal case of processing such data stream using p processors would be similar to the one shown in Fig. 4. In such ideal case, $t_1 = t_2 = \ldots = t_n$ and n is an exact multiple of p. This leads to a perfect speedup of p, though unfortunately barely impossible to existing real world applications. In reality, it may take variable amount of time to process different data fragments, and n is commonly not an exact multiple of p. In addition to that, even if the input data can be concurrently processed, the output data should be sequenced to guarantee the correctness of output stream.

One way to speed up data stream processing applications on multiple processors is to use a master-worker scheme as underlying parallelization structure. The master-worker scheme is a parallel skeleton for task pools with dynamic task distribution, what is particularly useful under the situation when there is a set of tasks to be done and completion times for each task are either unknown or vary a lot from task to task.

A master-worker system consists of a master process managing a set of worker processes. The master process distributes tasks to a set of subordinate worker processes and later collects computed results. There are two task pools in a master-worker system, the *pool of pending tasks* and the *pool of completed tasks*. Master process distributes tasks by filling data into the pool of pending tasks, and worker processes then fetch data from this pool to perform tasks. Once a worker finishes a task, the worker process fills the result to the pool of completed tasks. The master process then fetches results from the pool of completed tasks and outputs the results.

Fig. 5 illustrates a master-worker stream processing system that consists of one master process and 4 worker processes. The master process reads in the input stream and

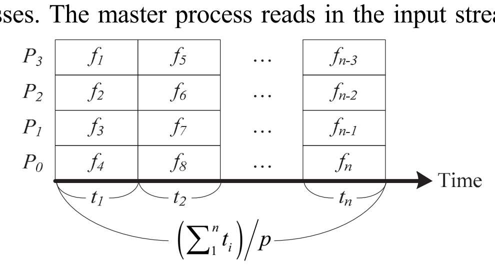


Figure 4. Perfect task scheduling of stream processing on 4 processors.

divides the input stream into smaller chunks of data that can be independently processed. Each of these chunks can be seen as a pending task that is then transferred into the pool of pending tasks by the master process. Once initialized, all worker processes in this system keep monitored the pool of pending tasks and see if there are present workloads. If such pool is not empty, any worker that is available can fetch (drain) a task from the pool and start to process it. Once completed such execution, it fills the pool of completed tasks with the results of current task. In the meanwhile, the master process monitors the pool of completed tasks and checks its status. Since completed tasks arrive in arbitrary order, master process keeps an output sequence counter. The counter is used to select the next completed task form the pool with correct sequence number to be output to the output stream.

Algorithm 1 shows the pseudo code of a master-worker stream processor.

A. Process Roles in a Master-Worker System

There are two parts in a master-worker system where communications take place between master and worker processes, namely *task distribution* and *result collection*. Each of these parts involves the handling of a task pool. In the task distribution part, master and worker processes work together to manipulate the pool of pending tasks, while in the result collection part, master and worker processes work together to manipulate the pool of completed tasks.

During the progress of task distribution, master process can be seen as a producer process and worker processes can be seen as consumer processes. This part is essentially one-to-many communication. On the other hand, in the progress of result collection, worker processes can be seen as producer processes and master process can be seen as a consumer process. This part is essentially many-to-one communication.

Fig. 6 shows the timing diagram of a master-worker stream processor featuring one master process and 4 worker processes. Note that synchronization overheads are introduced in both task distribution and result collection parts of the system. Programming paradigm used to implement the *fill*() and *drain*() functions have direct influences on these synchronization overheads, which further shapes the performance and scalability characteristics of the implemented system.

To focus on observation and comparison of performance impacts of shared memory programming paradigm, we use a flat master-worker structure rather than a hierarchical one. In this paper, the flat master-worker structure contains only one master process.

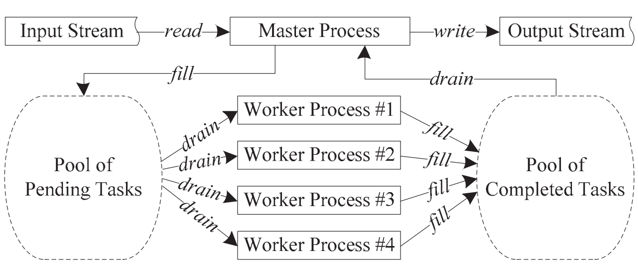


Figure 5. Illustration of a master-worker stream processor.

Algorithm 1: Master-Worker Stream Processor

Input : A data stream I consists of a sequence of data items of datatype α.
Output: A data stream$O$ consists of a sequence of data items of datatype β.
Data Types: α, β
Buffers: *pendingPool* $\longleftarrow \emptyset$, *completedPool*$\longleftarrow \emptyset$
Functions: `fill`(*buffer, item*), `drain`(*buffer*)

Process *master*: `/* one instance */`
Private Data: $F_\alpha \longleftarrow \emptyset, F_\beta \longleftarrow \emptyset$
begin
  `initialize` ($pendingPool$):
  `initialize` ($completedPool$):
  repeat
    `/* task distribution */`
    if $F_\alpha = \emptyset$ **then** $F_\alpha \longleftarrow$ `fetchNextItem`(I)
    if $F_\alpha \neq \emptyset$ **then**
      $result \longleftarrow$ `fill`($pendingPool, F_\alpha$)
      if $result = ture$ **then** $F_\alpha \longleftarrow \emptyset$
    `/* result collection */`
    $F_\beta \longleftarrow$ `drain`(*completedPool*)
    if $F_\beta \neq \emptyset$ **then**
      `output`(O, F_β)
      $F_\beta \longleftarrow \emptyset$
  until *program termination*

Process *worker*: `/* multiple instances */`
Private Data: $F_\alpha \longleftarrow \emptyset, F_\beta \longleftarrow \emptyset$
begin
  `initialize` ($pendingPool$):
  `initialize` ($completedPool$):
  repeat
    $F_\alpha \longleftarrow$ `drain`(*pendingPool*)
    if $F_\alpha \neq \emptyset$ **then**
      $F_\beta \longleftarrow$ `processItem`(F_α)
      `fill`(*completedPool,* F_β)
      $F_\alpha \longleftarrow \emptyset$
      $F_\beta \longleftarrow \emptyset$
  until *program termination*

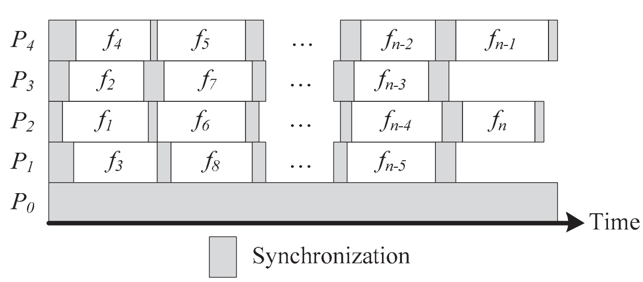


Figure 6. Synchronization overheads in a master-worker stream processing system.

IV. SHARED-MEMORY PROGRAMMING PARADIGMS FOR THE TILE64 PLATFORM

In this section, we introduce two shared-memory programming paradigms: the *consumer read programming* (CRP) and the *producer write programming* (PWP), as also show how CRP and PWP are used to implement shared-memory communication in a master-worker system on the TILE64 platform.

On the TILE64 platform, communication between two processes by using shared-memory mechanisms can be achieved by allowing a process to allocate a block of shared memory and then exchange the address of shared memory with another process. The steps involved in creating shared memory between processes are detailed in subsection II.B. All participating processes in the data communication are able to directly load value from or store value to the specified shared memory addresses, what provides flexibility of implementation.

By considering the scenario of implementing shared-memory communication between a producer process and a consumer process on the TILE64 platform, shared memory can be allocated by either producer process or consumer process. These two fundamentally different choices are the basis of CRP and PWP.

A. Consumer Read Programming

When producer process sends data to a consumer process, it writes the data into memory address shared by the producer process itself. Consumer process then reads the data from this shared address. The term *consumer read* implies the action of "consumer reads data from producer shared memory."

Fig. 7 depicts the initialization of CRP, where producer process allocates a region of shared memory to accommodate shared objects. Producer process then notifies consumer process the location of shared memory, so that producer checks and fills the shared memory if it is not full. Consumer keeps checking the content in the shared memory and consumes it if the shared memory is not empty.

B. Producer Write Programming

When a producer process sends data to a consumer process, it writes the data into memory address shared by the consumer process. The term *producer write* implies the action of "producer writes data to consumer shared memory."

In Fig. 8, consumer process allocates a region of shared memory to accommodate shared objects. Similarly to above discussion, consumer process then notifies producer process the location of shared memory, and producer checks and fills the shared memory if it is empty. Consumer keeps checking the content in the shared memory and consumes it if the content is valid.

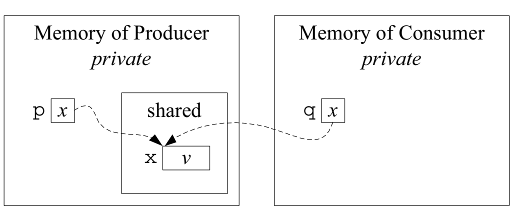


Figure 7. CRP illustration.

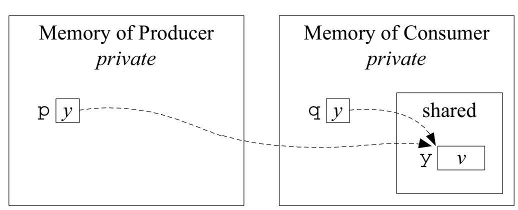


Figure 8. PWP illustration.

C. Implementation of Master-Worker System using CRP and PWP

There are multiple ways of using iLib shared-memory primitives to implement a master-worker stream processing system as described in Algorithm 1. The major difference is on the implementation of the two functions, *drain*() and *fill*(). These two functions are essential to the manipulation of the two task pools. Depending on the shared-memory programming paradigm used, the two pools of tasks can reside in memory addresses shared by either master process or worker processes.

The pool of pending tasks can be implemented using either CRP or PWP, so is the pool of completed tasks. The implementation algorithms are given in Algorithm 2 to 4. This gives us 4 master-worker system combinations:

1) CRP+CRP: Using CRP to implement both pools. The pool of pending tasks resides in memory shared by master process. And all of the worker shared memory combined together forms the pool of completed tasks. This combination is in fact implementation of a centralized pool of pending tasks and a distributed pool of completed tasks.

2) CRP+PWP: Using CRP to implement pool of pending tasks and using PWP to implement pool of completed tasks. Both pool of pending tasks and completed tasks reside in memory shared by master process. This combination is in fact implementation of a centralized pool of pending tasks and a centralized pool of completed tasks.

3) PWP+CRP: Using PWP to implement pool of pending tasks and using CRP to implement pool of completed tasks. Both pool of pending tasks and completed tasks are actually shared memory blocks distributed among all workers processes. This combination is in fact implementation of a distributed pool of pending tasks and distributed pool of completed tasks.

4) PWP+PWP: Using PWP to implement both pools. And all of the worker shared memory combined together forms the pool of pending tasks, and the pool of completed task resides in memory shared by master process. This combination is in fact implementation of a distributed pool of pending tasks and centralized pool of completed tasks.

V. EXPERIMENTAL RESULTS

We have modified an open source Motion JPEG decoder — *MJPEG Tools* [25], and made it a parallel decoder using

Algorithm 2: CRP Task Distribution

```
initialize (pendingPool):
begin
    switch role of the calling process do
        case master
            addr ⟵ mallocShared(α, size)
            broadcast addr to all worker
            pendingPool ⟵ addr
            break
        case worker
            receive addr broadcasted by master
            pendingPool ⟵ addr
            break

fill(pendingPool, F):        /* by master */
begin
    if notFull (pendingPool) then
        lock(pIn)
        pendingPool[pIn] ⟵ F
        pIn ⟵ (pIn + 1) modulo size
        unlock(pIn)
        return ture
    else
        return false

drain(pendingPool):        /* by workers */
begin
    if notEmpty (pendingPool) then
        lock(pOut)
        F_α ⟵ pendingPool[pOut]
        if CRP Result Collection is used then
            lock(OutputIndexQueue)
            enqueue myID to OutputIndexQueue
            unlock(OutputIndexQueue)
        pOut ⟵ (pOut + 1) modulo size
        unlock(pOut)
        return F_α
    else
        return ∅
```

Algorithm 3: PWP Task Distribution

```
initialize (pendingPool):
begin
    case master
        for i ← 0 to numOfWorkers−1 do
            receive addr from worker_i
            pendingPool[i] ⟵ addr
        break
    case worker
        addr ⟵ mallocShared(α, 1)
        send addr to master
        pendingPool ⟵ addr
        lock(AvailableWorkers)
        append myPID to AvailableWorkers
        unlock(AvailableWorkers)
        break

fill (pendingPool, F):        /* by master */
begin
    if AvailableWorkers ≠ ∅ then
        lock(AvailableWorkers)
        remove x from AvailableWorkers
        if CRP Result Collection is used then
            lock(OutputIndexQueue)
            enqueue x to OutputIndexQueue
            unlock(OutputIndexQueue)
        unlock(AvailableWorkers)
        pendingPool[x] ⟵ F
        return ture
    else
        return false

drain (pendingPool):        /* by workers */
begin
    if pendingPool ≠ ∅ then
        F_α ⟵ pendingPool
        return F_α
    else
        return ∅
```

master-worker structure as described in Section III. Then, we designed and instrumented the shared memory between master and worker processes using the following combinations: CRP+CRP (R+R), CRP+PWP (R+W), PWP+CRP (W+R) and PWP+PWP (W+W), as described in subsection IV.C.

A TILE64 hardware platform is used to conduct the performance evaluation. We ran the implemented decoders on a TILExpress-20G card, a TILE64 development platform featured with a TILE64 processor running at 700 MHz and 4 GBs of DDR2-800 memory.

Each of the decoders is setup to decode 4 videos files of different resolutions. Table I lists the video test files used. The files are placed in ram file system. Due to tiles located in the last row are reserved for system use and are not available to users when running programs on TILE64 hardware platform, the maximum number of tiles we used is 56 (8 columns by 7 rows.) We measure decoder performance from 2 tiles (1 master process and 1 worker process) to 56 tiles (1 master process and 55 worker processes) to obtain a total of 880 sets of timing data. We also collect 4 sets of sequential performance data to be the baseline for comparison. Table II shows the number of performance data sets collected between different configurations.

A. Speedup and Efficiency

Fig. 9 shows the speedup and efficiency results of the 4 decoders on different testing cases. These data are obtained by recording time spent on main decoding loop in the decoder and then compared to the same code segment in an

Algorithm 4: CRP Result Collection

```
initialize (completedPool):
switch role of the calling process do
    case master
        for i ← 0 to numOfWorkers−1 do
            receive addr from worker_i
            completedPool[i] ⟵ addr
        break
    case worker
        addr ⟵ mallocShared(β, 1)
        send addr to master
        completedPool ⟵ addr
        break

fill (completedPool, F):     /* by workers */
begin
    completedPool ⟵ F
    wait until get confirm message from master

drain (completedPool):        /* by master */
begin
    if OutputIndexQueue ≠ ∅ then
        lock (OutputIndexQueue)
        dequeue x from OutputIndexQueue
        unlock (OutputIndexQueue)
        F_β ⟵ completedPool[x]
        send confirm message to worker_i
        return F_β
    else
        return ∅
```

Algorithm 5: PWP Result Collection

```
initialize (completedPool):
switch role of the calling process do
    case master
        addr ⟵ mallocShared(β, size)
        broadcast addr to all worker
        completedPool ⟵ addr
        break
    case worker
        receive addr broadcasted by master
        completedPool ⟵ addr
        break

fill (completedPool, F):     /* by workers */
begin
    if notFull (completedPool) then
        lock(cIn)
        currentPosition ⟵ cIn
        cIn ⟵ (cIn + 1) modulo size
        unlock(cIn)
        completedPool[currentPosition] ⟵ F
        return ture
    else
        return false

drain (completedPool):        /* by master */
begin
    if notEmpty (completedPool) then
        F_β ⟵ completedPool[cOut]
        cOut ⟵ (cOut + 1) modulo size
        return F_β
    else
        return ∅
```

TABLE I. MOTION JPEG TEST FILES USED.

File Name	Format	Resolution	Frames
deadline	CIF	352×288	1374
city	4CIF	704×576	600
stockholm	720P	1280×720	604
factory	1080P	1920×1088	1339

TABLE II. PERFORMANCE DATA SETS OBTAINED.

Video Size	Sequential	Parallel			
	Vanilla	*R+R*	*R+W*	*W+R*	*W+W*
CIF	1	55	55	55	55
4CIF	1	55	55	55	55
720P	1	55	55	55	55
1080P	1	55	55	55	55

unmodified, sequential version of the decoder. Since parallel versions contain at least one master process and one worker process, the minimum number of cores required to run these parallel decoders is 2. When the parallel decoders are running using 2 cores, only the core that acts as worker process is responsible for the decoding job. Therefore, speedup and efficiency of the decoders on 2 cores would be close to 1 and 0.5 respectively.

The results show that the PWP+CRP implementation outperforms among all versions discussed in subsection IV.C. It can also be observed that the implementations can be separated into two groups by their speedup and efficiency characteristics. The R+R and W+R decoders, which are based on CRP result collection scales well when decoding 1080P videos. But the R+W and W+W decoders cannot scale beyond 16 workers.

B. Runtime Breakdown of Master Process

While speedup and efficiency charts shown in Fig. 9 provide overall performance summary, these two charts alone do not provide detailed information about processes themselves. Therefore, runtime breakdown charts are used to present these detailed information.

Due to running time of a master process decreases with increasing number of available worker processes, we use the percentage chart to better illustrate time spent by master process. For worker processes, we show the summation of total clock cycles spent by all worker processes. This enables us to ping-point program scalability issues by observing how much time have the worker processes actually spent on certain parts of the system.

Looking at Fig. 10, it is possible to identify the reasons why R+R and W+R do not scale well beyond 32 cores for

CIF video decoding. The worker processes drain the pool of pending tasks at higher speed than the rate master process fills the pool. Observing both Fig. 10 and 11, they show that for implementations based on PWP result collection, time spent by worker process on filling the pool of completed tasks grows linearly with number of participating worker processors in the system, degrading overall performance in these cases.

VI. Conclusion and Future Work

New generations of many-core processors bring higher performance within same or lower power envelope. This advantage comes with the price of complications to application programming. In this paper, we explore the design and implementation of a video decoder on the TILE64 platform. We design a master-worker structure for stream processing and propose two styles of shared memory programming paradigm—*consumer read programming* and *producer write programming*—for the TILE64 platform. Experimental results show that the CRP best suits implementation of result collection part in a master-worker Motion JPEG decoder while PWP performs better in the task distribution part.

We demonstrate that implementation choices for a given design on a many-core system will directly impact the performance and scalability of a program. We plan to further explore this topic by applying CRP and PWP onto more complicated designs such as hierarchical master-worker structures. And we would also like to see how CRP and PWP fit with applications of different data patterns such as those on video encoders.

References

[1] S. Borkar, "Thousand core chips: a technology perspective," *Proc. 44th Design Automation Conf. (DAC 07)*, 2007, pp. 746-749, DOI: 10.1145/1278480.1278667.

[2] J. Parkhurst, J. Darringer, and B. Grundmann, "From single core to multi-core: preparing for a new exponential," *Proc. IEEE/ACM Intl. Conf. Computer-Aided Design (ICCAD 06)*, Nov. 2006, pp. 67-72, DOI: 10.1145/1233501.1233516.

[3] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans, "Trends in multicore DSP platforms," *IEEE Signal Process. Mag,* vol. 26, pp. 38–49, 2009, DOI: 10.1109/MSP.2009.934113

[4] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," *Dr. Dobb's Journal,* vol. 30, pp. 202-210, Mar. 2005.

[5] G. Chen, F. Li, S. W. Son, and M. Kandemir, "Application mapping for chip multiprocessors," *Proc. 45th Design Automation Conference (DAC 08)*, June 2008, pp. 620-625, DOI: 10.1145/1391469.1391628.

[6] T. Scogland, P. Balaji, W. Feng, and G. Narayanaswamy, "Asymmetric interactions in symmetric multi-core systems: analysis, enhancements and evaluation," *Proc. Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC 08)*, Nov. 2008, pp. 1-12, DOI: 10.1109/SC.2008.5219748.

[7] B. So, A. Ghuloum, and Y. Wu, "Optimizing data parallel operations on many-core platforms," in *1st Workshop on Software Tools for Multi-Core Systems (STMCS 06)* 2006.

[8] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," *Proc. ACM SIGPLAN Conf. Programming language design and implementation (PLDI 08)*, 2008, vol. 43, pp. 114-124, DOI: 10.1145/1375581.1375596.

[9] G. Tan, N. Sun, and G. R. Gao, "A parallel dynamic programming algorithm on a multi-core architecture," *Proc. 19th ACM Symp. Parallel Algorithms and Architectures (SPAA 07)*, 2007, pp. 135-144, DOI: 10.1145/1248377.1248399.

[10] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe, "Hierarchical master-worker skeletons," in *Practical Aspects of Declarative Languages (PADL 08)*, *LNCS 4902*, P. Hudak and D. Warren, Eds.: Springer-Verlag, 2008, pp. 248-264, DOI: 10.1007/978-3-540-77442-6_17.

[11] A. Benoit, L. Marchal, J. F. Pineau, Y. Robert, and F. Vivien, "Scheduling concurrent bag-of-tasks applications on heterogeneous platforms," *IEEE Trans. Computers,* vol. 59, pp. 202-217, 2010, DOI: 10.1109/TC.2009.117.

[12] Tilera corporation, `http://www.tilera.com`.

[13] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," in *High Performance Embedded Architectures and Compilers*, *LNCS 5952*, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds.: Springer-Verlag, 2010, pp. 3-17, DOI: 10.1007/978-3-642-11515-8_3.

[14] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling," *Proc. 32nd International Symposium on Computer Architecture (ISCA 05)*, June, 2005, pp. 408-419, DOI: 10.1109/ISCA.2005.34.

[15] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," *Proc. 19th Intl. Conf. Parallel Architectures and Compilation Techniques (PACT 10)*, Sep. 2010, pp. 319-330, DOI: 10.1145/1854273.1854314.

[16] D. Abts, N. D. E. Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving predictable performance through better memory controller placement in many-core CMPs," *Proc. 36th Intl. Symp. Computer Architecture (ISCA 09)*, June 2009, pp. 451-461, DOI: 10.1145/1555754.1555810.

[17] X.-Y. Lin, C.-Y. Huang, P.-M. Yang, T.-W. Lung, S.-Y. Tseng, and Y.-C. Chung, "Parallelization of motion JPEG decoder on TILE64 many-core platform," in *Methods and Tools of Parallel Programming Multicomputers (MTPP 10)*, *LNCS 6083*, C.-H. Hsu and V. Malyshkin, Eds.: Springer-Verlag, 2011, pp. 59-68, DOI: 10.1007/978-3-642-14822-4_7.

[18] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, et al., "TILE64 processor: a 64-core SoC with mesh interconnect," *Proc. IEEE Intl. Solid-State Circuits Conf. (ISSCC 08)*, Feb, 2008, pp. 88-598, DOI: 10.1109/ISSCC.2008.4523070.

[19] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proc. 17th Intl. Symp. Computer Architecture (ISCA 90)*, 1990, pp. 15-26, DOI: 10.1145/325164.325102.

[20] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," *Proc. 19th Intl. Conf. Parallel Architectures and Compilation Techniques (PACT 10)*, Sep. 2010, pp. 365-376, DOI: 10.1145/1854273.1854319.

[21] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *Proc. 12th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 06)*, Oct. 2006, pp. 151-162, DOI: 10.1145/1168857.1168877.

[22] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," *Proc. 21st Symp. Principles of Database Systems (PODS 02)*, 2002, pp. 1-16, DOI: 10.1145/543613.543615.

[23] J. Park and W. J. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," *Proc. 22nd ACM Symp. Parallelism in Algorithms and Architectures (SPAA 10)*, June 2010, pp. 1-10, DOI: 10.1145/1810479.1810481.

[24] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, et al., "Scalable distributed stream processing," in *Conf. Innovative Data Systems Research (CIDR 03)*, Jan. 2003.

[25] MJPEG Tools, `http://mjpeg.sourceforge.net`.

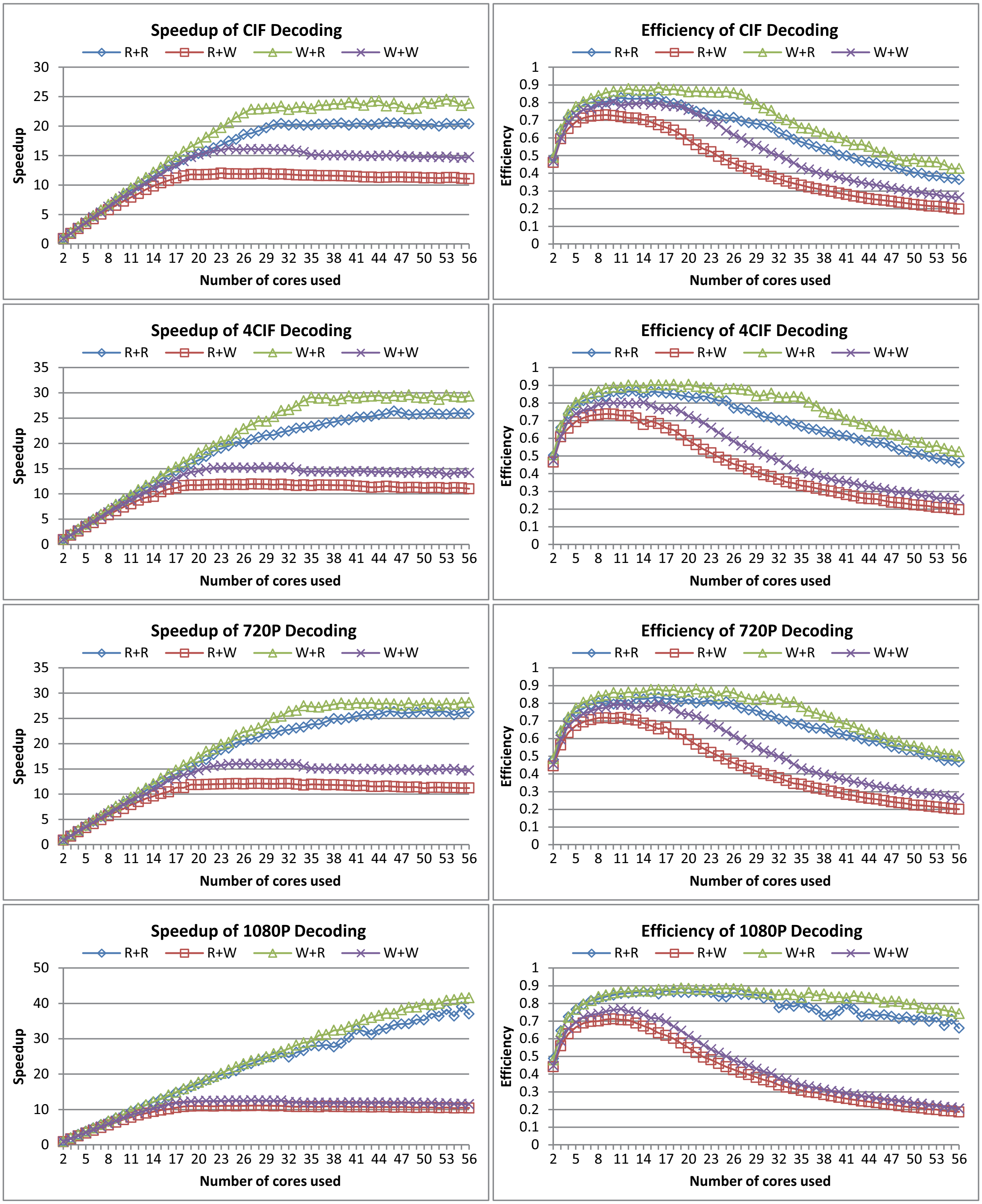
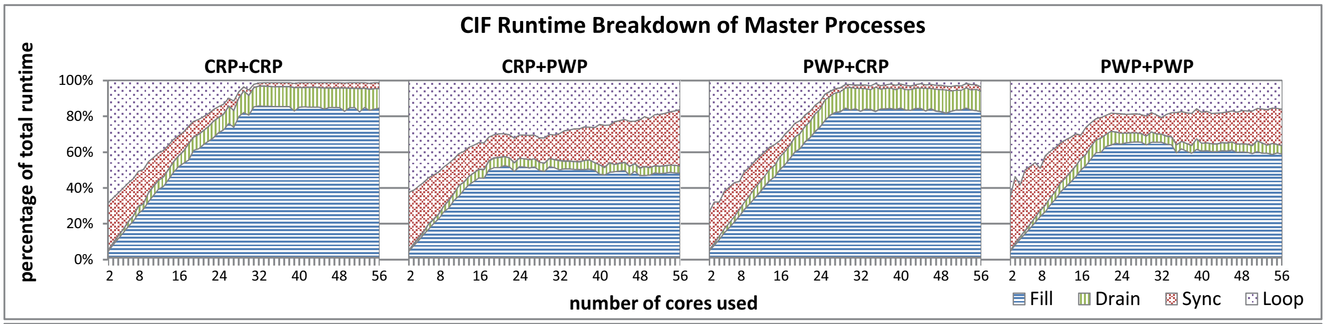


Figure 9. Speedup and efficiency results of the 4 implemented decoders on 4 different video frame sizes.



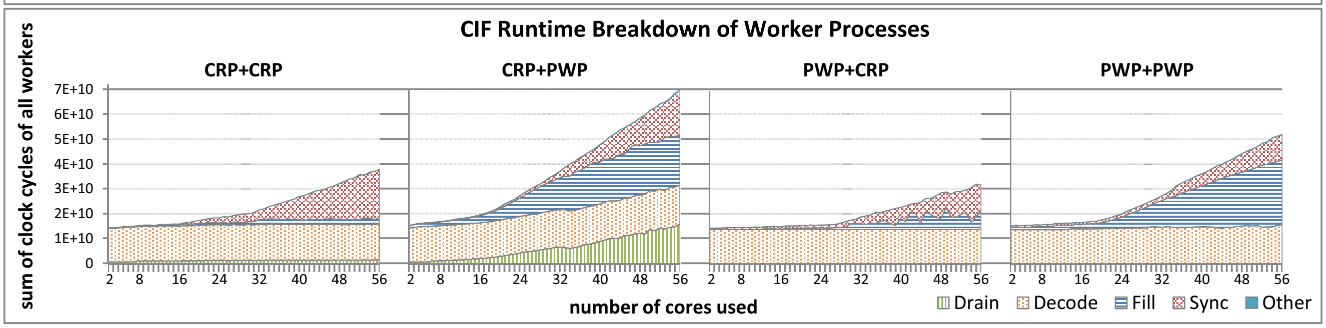
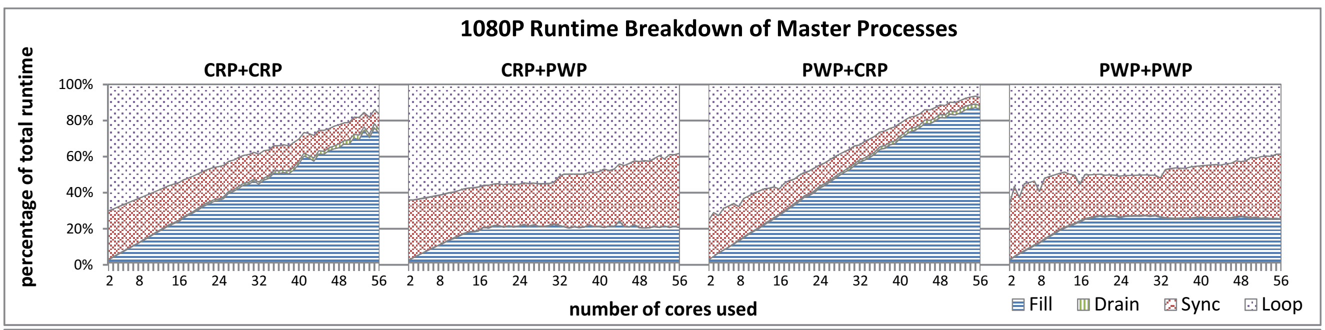


Figure 10. Runtime breakdown of CIF decoding.



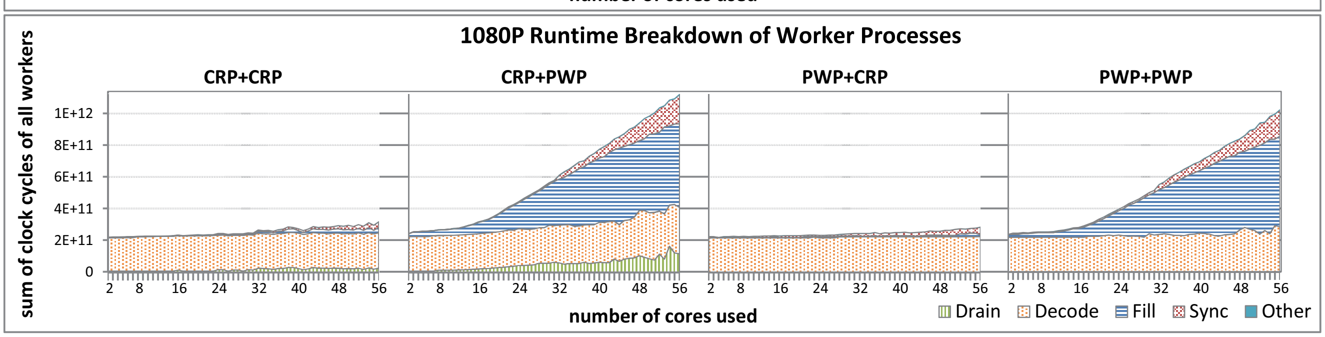


Figure 11. Runtime breakdown of 1080P decoding.