UDispatch$^+$: A User Dispatching Tool with Automatic Binding

Tang-Hsun Tu, Yuan-Cheng Lee, and Chih-Wen Hsueh
Embedded System and Wireless Network Laboratory
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan 106, R.O.C.
E-mail: {d98944004, b94101, cwhsueh}@csie.ntu.edu.tw

Yi-Sheng Liu
Department of Information Management
National Chi Nan University
Nantou, Taiwan 545, R.O.C.
E-mail: s96105016@ncnu.edu.tw

***Abstract*—In multicore environment, multithreading is often used to improve application performance. However, there are some unexpected anomalies which degrade the performance of multithreading applications. Some of the unexpected anomalies come from inappropriate thread dispatching by operating system. To solve this problem, a novel User Dispatching Mechanism (UDispatch) was proposed. Through modification of application source codes with the UDispatch application programming interface (API), the application performance can be improved significantly. However, most of the time, the application source codes are not available, or it might not be suitable to do the modification of source codes. Therefore, we provide another tool to dispatch threads without any modification of application source codes, called User Dispatching Plus (UDispatch$^+$). It can bind the application threads automatically and dispatch to specific cores at the discretion of users. We experiment UDispatch$^+$ on multithreading multimedia applications. The results show that a more parallelized skip-line application can speed up to 171.8% on a 4-core machine, and a more dependent optimized H.264/AVC decoder can speed up to 20.1% on a 4-core machine.**

***Keywords*-Scheduling, Dispatching, Anomaly, Multithreading, Multicore, Virtual Device.**

I. Introduction

There are some unexpected scenarios, which users usually do not notice, in dispatching threads by operating system. For example, even if some cores are idle, the operating system might not dispatch any thread to the idle ones because it might not reach the load-balancing threshold or some threads finish too quick before the operating system can do any correct response. However, when users find out this situation, they still cannot do anything about thread dispatching if the operating system does not provide related control mechanisms.

Nowadays, there are some existing approaches that allow users to dispatch threads without any modification to source codes, e.g. taskset [1] and bindprocessor [2]. They can bind threads to a specific core through command line instructions. However, even if it is convenient to use, there are still not enough for multithreading applications. For example, if users only bind main thread to a specific core, the slave threads will inherit core affinity of the main thread and the workload will be still placed totally on the same core. Since taskset only can set a thread one at a time, to dispatch a multithreading application, users need to get process identifiers (pids) of all needed slave threads and execute this command many times. Moreover, sometimes, we need related system information to decide the dispatching. However, getting those information might incur extra overhead [3], especially for some timing critical applications. Another problem is that since users often do not have knowledge of the application, i.e. when the slave threads will be created, the later binding of the slave threads, the higher overhead that could be controlled. Therefore, we designed UDispatch [3] to provide the control mechanism and act as a bridge between user applications and kernel in the form of a virtual device [4], where the virtual device design further reduces the work of modifying kernel. Through this mechanism, users can dispatch threads of their applications to specify cores easily and efficiently.

Although UDispatch can improve the performance of applications remarkably, but it requires users to modify the source codes of their applications. In most cases, users do not have the source codes, and the specific application domain knowledge to modify the source codes. Therefore, extend UDispatch to UDispatch$^+$ by adding a shell command. It can manipulate kernel and UDispatch for thread binding thus requiring no modification to source codes with little patch to Linux kernel.

Our contributions can be summarized as follow:

- We present an automatic dispatching program, UDispatch$^+$, which can dispatch threads to specific cores so that the application performance can be improved on multicore systems without any modification of source code.
- We apply and verify UDispatch$^+$ through multimedia applications and sustain higher performance improvement than UDispatch.

The rest of this paper is organized as follows. In Section II, we present the framework and the usage of UDispatch$^+$. Section III shows the experiment results of UDispatch$^+$. Section IV shows the comparison between UDispatch$^+$ and UDispatch. This paper is concluded in Section V.

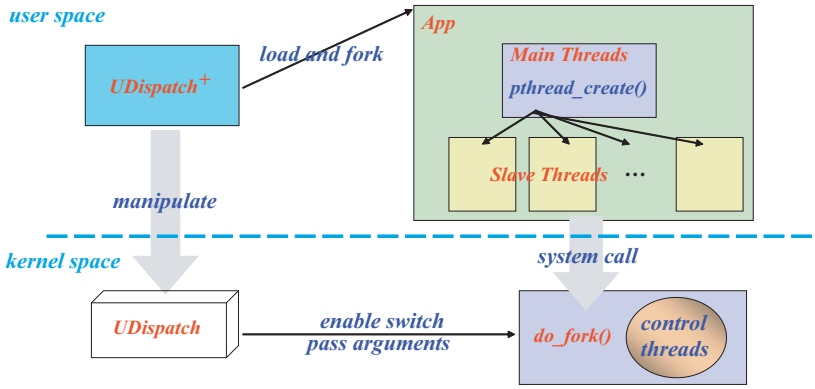

Figure 1: UDispatch+ architecture.

```
1  #include "UDispatch.h"
2
3  void UDPlus_main(int fd, pid_t ppid,
   pid_t cpid, unsigned char bm_flag){
4      // pass pids to UDispatch
5      // (UDispatch+ pid and leader pid)
6      set_UDPlus_pids(fd, ppid, cpid);
7      //bind main thread
8      if(bm_flag) bind_to_cpu(fd, cpid,
       find_least_load_core());
9      // enable the switch for slave
10     // threads
11     enable_UDPlus_switch(fd);
12     // wait specify application finishes
13     wait_app_done(cpid);
14     // disable the switch
15     disable_UDPlus_switch(fd);
16 }
```

Code 1: The default template of UDispatch+.

II. UDispatch+ Framework

To develop UDispatch+, the main concern is how the workflow executes, i.e. when to bind, where to bind, how to get the system information of the application and cores, and how to decrease the overhead of dispatching threads due to the system design as much as possible.

In Linux, a thread is treated as a process, and a process is created by the low-level kernel function *do_fork()*, i.e. system call *fork*. Therefore, to decide when to bind and where to bind, our idea is making a switch in the do_fork() function to automatically detect whether a newly created thread should be bound or not. As shown in Figure 1, an application, App, creates multiple slave threads, and the threads will be bound to specific cores. According to the mechanism in Linux, each thread will call the do_fork() to build its context in the creation process. When UDispatch+ starts, the application

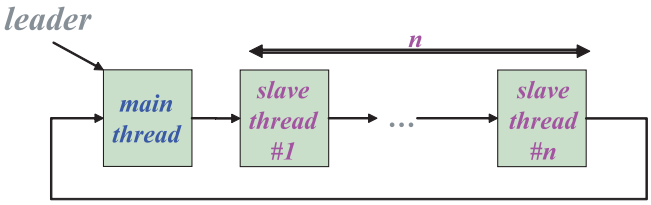

Figure 2: Multi-thread structure with identifiers.

will be loaded and forked by UDispatch+ as a child process. UDispatch+ then passes its pid and the pid of its child to UDispatch and enables the switch through UDispatch to start binding.

Since multiple threads might be created simultaneously, a simple switch mechanism in the do_fork() is not enough to identify whether a newly created thread needs to be bound or not. Therefore, another comparison is added to solve this problem. In Linux kernel, a multi-threading application is stored in a list structure as in Figure 2. The main thread is the group leader, and its slave threads are group members and linked sequentially. To ensure that a thread is the correct one for controlling, we make a comparison between the main thread pid of the application passed from UDispatch+ and the pid of a newly created thread. If the pids are matched, the newly created thread is a slave thread of the application and it will be bound to the specific core automatically; otherwise, nothing to be done.

UDispatch+ is implemented as a user application. To fit various applications, we provide a UDispatch+ template in C language as in Code 1, and users can customize this template easily. There are some related functions in UDispatch API as follows:

- ***set_UDPlus_pids***: Passing pids to UDispatch, i.e. pids of UDispatch+ and the application.
- ***enable_UDPlus_switch***: Turning on the switch to bind specific threads automatically in kernel function do_fork().
- ***disable_UDPlus_switch***: Turning off the switch to stop

the binding in kernel function do_fork().

```
1  void UDPlus_main(int fd, pid_t ppid,
   pid_t cpid, unsigned charbm_flag) {
2      int i, nr_slave_threads = 0, ret;
3      pid_t *pidary = NULL;
4      #define NR_THREADS 4
5      ...
6      // get threads' pids on a
7      // specific application
8      while(nr_slave_threads !=
       NR_THREADS)
9          nr_slave_threads =
           get_nr_threads(fd, cpid);
10     pidary = (pid_t *)
       malloc(nr_slave_threads *
       sizeof(pid_t));
11     get_pids_on_task(fd, pidary,
       nr_slave_threads, cpid);
12     // bind threads to specify cores
13     ret = bind_to_cpu(fd, pidary[1],
       find_least_load_core());
14     if(ret) ; // bind failure
15     bind_to_cpu(fd, pidary[3],
       find_least_load_core());
16     ...
17     free(pidary);
18 }
```

Code 2: An example of customized UDispatch$^+$.

As shown in Code 1, the entry point of UDispatch$^+$ is *UDPlus_main()*. The first parameter *fd* is the file descriptor of the virtual device of UDispatch. The second parameter *ppid* is the UDispatch$^+$ pid got by system call *getpid()*. The third parameter *cpid* is the pid of multi-threading application for controlling which is obtained by system call *fork()* and the last parameter *bm_flag* indicates whether to bind the main thread or not. The function *find_least_load_core()* returns the core with the least load and the function *wait_app_done()* is implemented by system call *wait()* to wait until the specified application finishes. When users want to use the switch mechanism in do_ork() function, *set_UDPlus_pids()* should be called first to pass pids for binding, and then the switch is turned on to start checking by calling *enable_UDPlus_switch()*. After the multi-threading application (cpid) finishes, *disable_UDPlus_switch()* is called to turn off the switch. To customize UDispatch$^+$, users only need to modify the entry function *UDPlus_main()*. An example is shown at Code 2, where it is customized to get identifiers of the application threads of more than the default number and assign one of the threads to a specific core.

Table I: Configuration of experiment machines.

	4-Core Machine
CPUs	Intel®Core™2 Quad CPU 2.4 GHz
Memory	1G DDR2 RAM
Hard Disk	160 G
OS	Ubuntu 7.04 - Linux-2.6.2.17-generic Upgraded to Linux-2.6.26.3

Table II: Skip-line arguments.

	VTL	**Range**	**Threshold**	**Image Size**
Value	Yes	7	3	1920x1080

III. Experiments

We apply the default template of UDispatch$^+$ listed in Code 1 to two multi-threading applications to measure its performance, one is the skip-line encoder and decoder [5], and the other is an optimized H.264 decoder. The experiments are conducted on a 4-core machine with the configuration in Table I. The experiment results are averaged from 1000 times and 500 times of execution with the same binding approach for skip-line application and H.264 decoder respectively.

For convenience, we use the following shorthand to represent different binding approaches:

- **NoB**: threads of user application are executed without core binding.
- **1-1**: slave threads are bound with one-to-one mapping.
- **M-L**: main thread are bound with the core of the least load and slave threads are bound with one-to-one mapping.
- **1-1**$^+$: using 1-1 in UDispatch$^+$.
- **M-L**$^+$: using M-L in UDispatch$^+$.

A. Skip-line Algorithms

Skip-line encoding is one of the lossy encoding techniques for binary image. It keeps a line using the run-length encoding, compares with the next line, and skips it until the line is not similar or fix number of lines has been skipped. Then, it keeps the dissimilar line or the next line to continue comparison. To get higher compression ratio while still keeping high recognizability, *Basic*, *Swing*, and *TOW* were three variants of skip-line algorithms with different thresholds [3].

Based on the skip-line algorithms with threshold, we can partition a whole image into some independent image strips as shown in Figure 3, where nr_skip_line is the number of lines encoded (or skipped plus one) from the original image. For each strip, we construct a thread to do encoding or decoding. Therefore, it is a very parallelized application. Since we focus on the behavior of threads, the

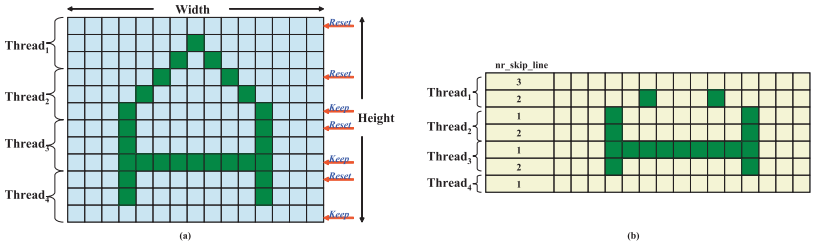

Figure 3: Skip-line (a) encoding (b) decoding with multiple threads.

Table III: Attributes and speed-up (%) of H.264 bitstreams.

Attributes				Speed-up		
No.	**Name**	**Resolution**	**# of Frames**	NoB M-L$^+$	1-1$^+$ M-L$^+$	NoB 1-1$^+$
1	Harbour	720P (1280x720)	300	12.9	0.4	12.5
2	Night	720P (1280x720)	300	14.4	0.2	14.2
3	Jets	720P (1280x720)	300	20.1	0.3	19.7
4	Harbour	480P (720x480)	300	12.0	0.3	11.6
5	Crew	480P (720x480)	300	15.9	0.6	15.2
6	Sailormen	480P (720x480)	300	12.9	0.1	12.9
7	Night	480P (720x480)	230	13.3	0.4	12.9
8	Mobile	CIF (352x288)	300	10.2	0.1	10.2
9	Football	CIF (352x288)	260	11.5	0.1	11.4
10	Bus	CIF (352x288)	150	13.4	0.1	13.3

Table IV: The speed-up (%) of skip-line applications.

	Skip-line Encoding			Skip-line Decoding		
	NoB M-L$^+$	1-1$^+$ M-L$^+$	NoB 1-1$^+$	NoB M-L$^+$	1-1$^+$ M-L$^+$	NoB 1-1$^+$
Basic	121.5	30.6	69.6	88.2	0.7	86.9
Swing	171.8	24.2	118.9	49.7	1.1	48.0
TOW	149.2	24.6	100.1	74.5	1.9	71.2

arguments of skip-line algorithm as described in Table II can be fixed, where VTL means that it also solves a vertical line problem [5] when encoding, $Range$ is the number of pixels to compare for each pixel, and $Threshold$ is the number of dissimilar pixels for skipping a line.

Table IV summarizes the performance gain using UDispatch$^+$ for skip-line encoding and decoding on the 4-core machine. The speed-up is defined by

$$\text{Speed-Up} = \left(\frac{\{\text{NoB} \mid \text{1-1}^+\}}{\{\text{M-L}^+ \mid \text{1-1}^+\}} - 1\right) * 100,$$

where the vertical bar stands for logical *OR*. As shown in Table IV, comparing to the original encoder without UDispatch$^+$, NoB, the improvement of M-L$^+$ can achieve up to 121.5%, 171.8%, and 149.2% for skip-line algorithm Basic, Swing, and TOW, respectively, and comparing to the original decoder without UDispatch$^+$, NoB, the improvement of M-L$^+$ can achieve up to 88.2%, 49.7%, and 74.5% for skip-line algorithm Basic, Swing, and TOW, respectively.

B. H.264 Decoder

H.264 or Advanced Video Coding (AVC) is a standard [6] for video compression. It is widely used in HD-DVD and Blu-ray Disc for its high quality and support of digital television broadcasting. Figure 4 shows the block diagrams of H.264 decoding. The main components are entropy decoding (ED), inverse quantization (IQ), inverse discrete cosine transform (IDCT), motion compensation (MC), intra prediction (IP), and deblocking filter (DF) [7]. Since the components are connected sequentially with lots of dependencies, except for deblocking filtering, it is difficult to apply the multi-threading technique to exploit parallelism and speed up its performance. Therefore, we focus on optimizing the multi-threading of deblocking filter. Based on the H.264 standard reference software [8], the H.264 decoder has been optimized with many advanced features [9], [10]. We adopt an efficient parallel deblocking filter algorithm [11] and apply UDispatch$^+$ on it. The deblocking filter is executed with four threads on the 4-core machine.

We use ten quite different H.264 bitstreams in the experiments. The performance gains compared to the ones without using UDispatch$^+$ is summarized in Table III. The speed-up is defined as

$$\text{Speed-Up} = \left(\frac{\{\text{NoB} \mid \text{1-1}^+\}}{\{\text{M-L}^+ \mid \text{1-1}^+\}} - 1\right) * 100.$$

Table III shows that it has 10.2% to 20.1% performance gain when using UDispatch$^+$. The main thread bound with

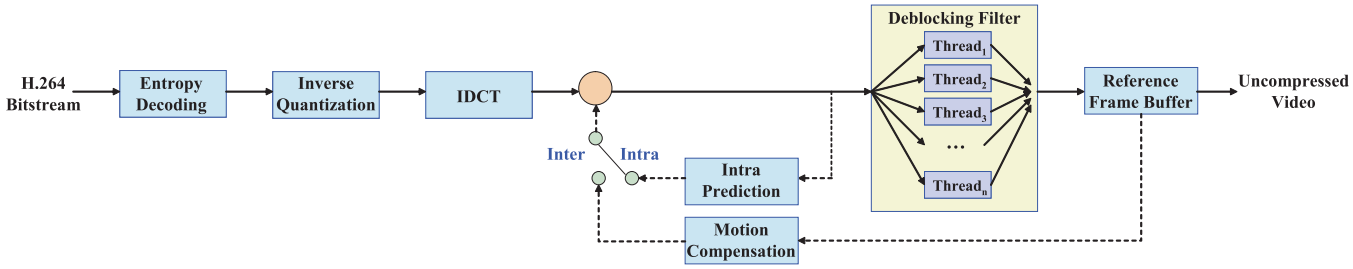

Figure 4: H.264 baseline decoding with multiple threads.

Table V: Speed-up (%) of H.264 decoder.

No.	NoB M-L	1-1 M-L	NoB 1-1	M-L M-L$^+$	1-1 1-1$^+$
1	12.9	0.7	12.1	0.1	0.4
2	14.4	0.5	13.7	0.1	0.3
3	19.9	0.1	19.0	0.7	0.6
4	11.3	0.2	11.1	0.6	0.4
5	15.1	0.3	14.7	0.7	0.5
6	12.8	0.1	12.6	0.2	0.2
7	13.0	0.4	12.6	0.2	0.2
8	10.0	0.1	10.0	0.2	0.4
9	11.2	0.1	11.1	0.3	0.3
10	13.0	0.1	12.8	0.4	0.2

the core of the least load has up to 0.6% performance gain than the one-to-one binding approach.

IV. Comparison between UDispatch$^+$ and UDispatch

In this section, we make a comparison between UDispatch$^+$ and UDispatch. First, we apply UDispatch to the same experiment applications and then compare the result with one from the earlier experiment. The results are listed in Table VII and Table V, and the speed-up is defined as

$$\text{Speed-Up} = \left(\frac{\{\text{NoB} \mid \text{1-1} \mid \text{M-L}\}}{\{\text{M-L} \mid \text{1-1} \mid \text{M-L}^+ \mid \text{1-1}^+\}} - 1\right) * 100.$$

As we presented earlier, UDispatch$^+$ employs a switch to control the thread binding in the low-level kernel function *do_fork()* directly. However, when using UDispatch, the thread will not be bound until the function *bind_to_cpu()* is called. So, before the binding, the thread might be initially scheduled on a heavily-loaded core. Also, a migration will occur, which causes undesired overhead, if the specified core is different from the initial one. Hence, in general, the performance of UDispatch$^+$ should be better than UDispatch.

As shown in Table VII, UDispatch$^+$ can achieve better performance than UDispatch in M-L and 1-1 bindings for both encoding and decoding of skip-line algorithms.

Table VI: Comparison between UDispatch and UDispatch$^+$

	UDispatch	UDispatch$^+$
Flexibility	More	Less
Convenience	Less	More
Performance	Good	Better
Extra Overhead	Little	More
Real-time	Better	Good

Although the best performance gain of 1-1 in skip-line encoding is closed to 1-1$^+$ in Figure 5 (only first 100 executions are shown because of the clear difference), UDispatch$^+$ shows a much more stable execution behavior than UDispatch. Actually, it has been shown in Table VII that UDispatch$^+$ can achieve higher performance gain using 1-1 approach, i.e. 33.1%, 50.9% and 48.8% respectively.

As Table VI shows, we make a summary of comparison between UDispatch$^+$ and UDispatch to help users choose which one is suitable for their applications.

With UDispatch$^+$, a thread will be bound when it is just created, but with UDispatch, users can control when the binding should be done through UDispatch API. Thus, UDispatch has more flexibility. From the view point of convenience, UDispatch$^+$ is preferred because users can execute an application without making any modification to the application. For UDispatch, applications must be modified by using the UDispatch API. As discussed earlier, for UDispatch, thread might be scheduled to a heavily-loaded core initially, and there might be an extra migration if the bound core is different from the original one. On the other hand, though UDispatch$^+$ has better performance, it incurs extra overhead to fork a specific application in the initialization and need to manage some extra data structures. As for real-time concern, since UDispatch$^+$ is a user process, it is scheduled together with other processes. In other words, we cannot guarantee the execution behavior of UDispatch$^+$ and thus it is hard to predict when the bound application will be executed. Hence it is not suitable for applications with critical real-time constraints. To deal with this issue, we can raise the priority of UDispatch$^+$ process to reduce the uncertainty.

Table VII: Speed-up (%) of skip-line algorithms.

	Skip-line Encoding					Skip-line Decoding				
	NoB M-L	1-1 M-L	NoB 1-1	M-L M-L$^+$	1-1 1-1$^+$	NoB M-L	1-1 M-L	NoB 1-1	M-L M-L$^+$	1-1 1-1$^+$
Basic	115.5	69.2	27.4	2.8	33.1	84.9	1.3	82.5	1.8	2.4
Swing	166.1	96.1	35.7	2.2	50.9	46.8	1.2	45.0	2.0	1.5
TOW	147.1	28.7	92.0	0.8	48.8	70.5	0.1	70.4	2.4	1.9

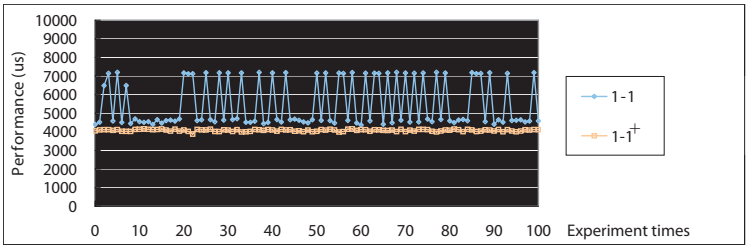

Figure 5: 1-1 versus 1-1$^+$ for skip-line encoding.

As described above, although under some circumstances, such as applications with critical real-time constraints, UDispatch$^+$ might not be able to outperform UDispatch. UDispatch$^+$ is in general a better choice, because its performance is better than UDispatch in most situations, and it can still work even if the source code is not available for modification.

V. Conclusion

We enhance a tool UDispatch to dispatch threads of application without any modification of application source codes, called User Dispatching Plus or UDispatch$^+$. We also provide an example template of UDispatch$^+$ for users to customize for various applications with different thread binding approaches. We conduct experiments for multimedia applications with UDispatch and UDispatch$^+$. The results show that UDispatch$^+$ has significant performance improvement, where a skip-line encoder can be improved up to 171.8% on a 4-core machine, and up to 20.1% enhancement for an optimized H.264 decoder in the same environment. We believe that UDispatch$^+$ provides convenient controllability so that users can easily and effectively dispatch threads for their applications to tune up the performance.

Acknowledgment

Supported in part by research grants from the ROC National Science Council, NSC 99-2628-E-002 -027 - and the Excellent Research Projects of National Taiwan University, 99R80304.

References

[1] "util-linux-ng," http://www.kernel.org/pub/linux/utils/util-linux-ng/, 2010.

[2] "IBM:bindprocessor," http://publib.boulder.ibm.com/infocenter/aix/v6r1/index.jsp, 2010.

[3] T.-H. Tu, C.-W. Hsueh, and R.-G. Chang, "A Portable and Efficient User Dispatching Mechanism for Multicore Systems," in *The 15th International Conference on Real-Time Computing Systems and Applications (RTCSA '09)*, Aug. 2009.

[4] "Virtual Device - Wikipedia, the free encyclopedia," http://en.wikipedia.org/wiki/Virtual_device, Sep. 2008.

[5] A. Moinuddin, E. Khan, and F.Ghani, "An Efficient Technique for Storage of Two-Tone Images," *IEEE Transactions on Consumer Electronics*, vol. 43, no. 4, pp. 1312–1319, Nov. 1997.

[6] ISO/IEC 14496-10, International Standard of Joint Video Specification, *Coding of Audiovisual Objects-Part 10: Advanced Video Coding*, ISO/IEC Std., 2003.

[7] I. E. Richardson, *H.264 and MPEG-4 Video Compression*, 1st ed. Wiley, Aug. 2003, iSBN 0-470-84837-5.

[8] "H.264/AVC JM Reference Software," http://iphome.hhi.de/suehring/tml, Jan. 2009.

[9] S.-W. Wang, Y.-T. Yang, C.-Y. Li, Y.-S. Tung, and J.-L. Wu, "The Optimization of H.264/AVC Baseline Decoder on Low-Cost TriMedia DSP Processor," *Proceeding of SPIE*, vol. 5558, 2004.

[10] X. Zhou, E. Q. Li, and Y.-K. Chen, "Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions," *Proceeding of SPIE Conference on Image and Video Communication and Processing*, vol. 5022, Jan. 2003.

[11] S.-S. Yang, S.-W. Wang, and J.-L. Wu, "A Parallel Algorithm for H.264/AVC Deblocking Filter Based on Limited Error Propagation Effect," in *IEEE International Conference on Multimedia and Expo*, Jul. 2007, pp. 1858–1861.