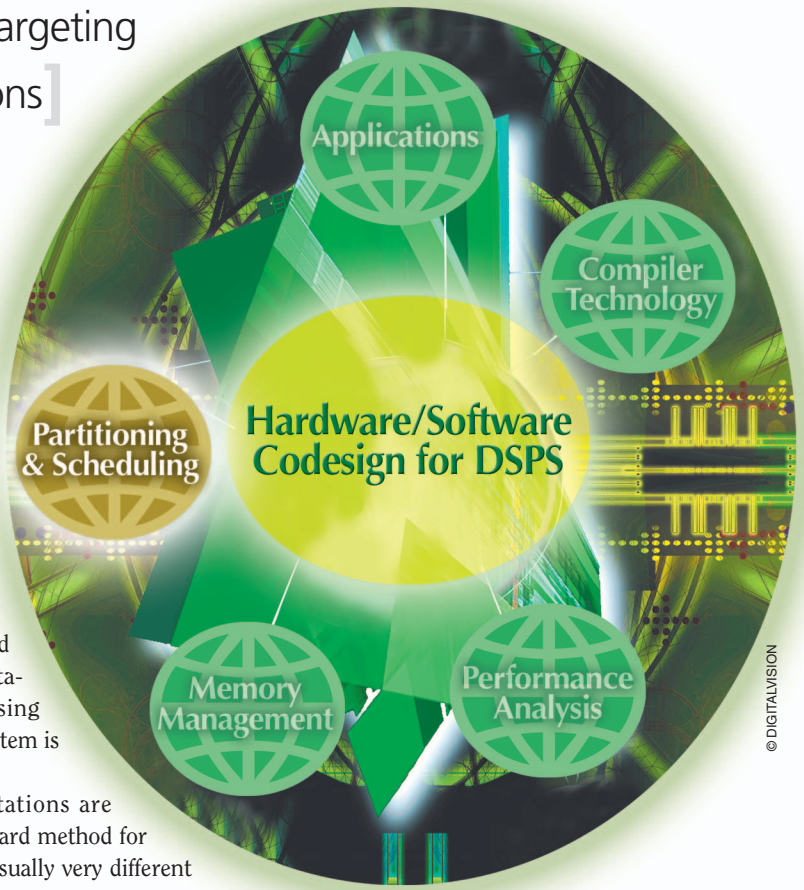
Theerayod Wiangtong, Peter Y.K. Cheung, and Wayne Luk

Hardware/Software Codesign

A systematic approach targeting data-intensive applications



Reconfigurable hardware has received increasing attention in the past decade due to its adaptable capability and short design time. Instead of using field-programmable gate arrays (FPGAs) simply as application-specific integrated circuit (ASIC) replacements, designers can combine reconfigurable hardware with conventional instruction processors in a codesign system, providing a flexible and powerful means of implementing computationally demanding digital signal processing (DSP) applications. This type of codesign system is the focus of this article.

Most traditional codesign implementations are application specific and do not have a standard method for implementing tasks. A hardware model is usually very different from those used in software. These distinctive views of hardware and software tasks can cause problems in the codesign process. For example, swapping tasks between hardware and software can result in a totally new structure in the control circuit. In addition, many design tools leave the designers to make their own decisions on task partitioning and scheduling, although these decisions dramatically affect the system performance and cost. For example, partitioning in [1] has to be done manually and there is no reconfiguration at run-time.

This article presents a systematic approach to hardware/software codesign targeting data-intensive applications. We focus on application processes that can be represented in directed acyclic graphs (DAGs) and use a synchronous dataflow (SDF) model, the popular form of dataflow employed in DSP systems [2], when running the processes. The codesign system is based on the UltraSONIC reconfigurable platform, a system designed jointly at Imperial College and the SONY Broadcast Laboratory. This system is modeled as a loosely coupled structure consisting of a single instruction processor and multiple reconfigurable hardware elements. We suggest a new method of constructing and handling system tasks for this real codesign system. Both hardware and software tasks are structured in an interchangeable manner without sacrificing the benefit of concurrency found in conventional hardware implementations.

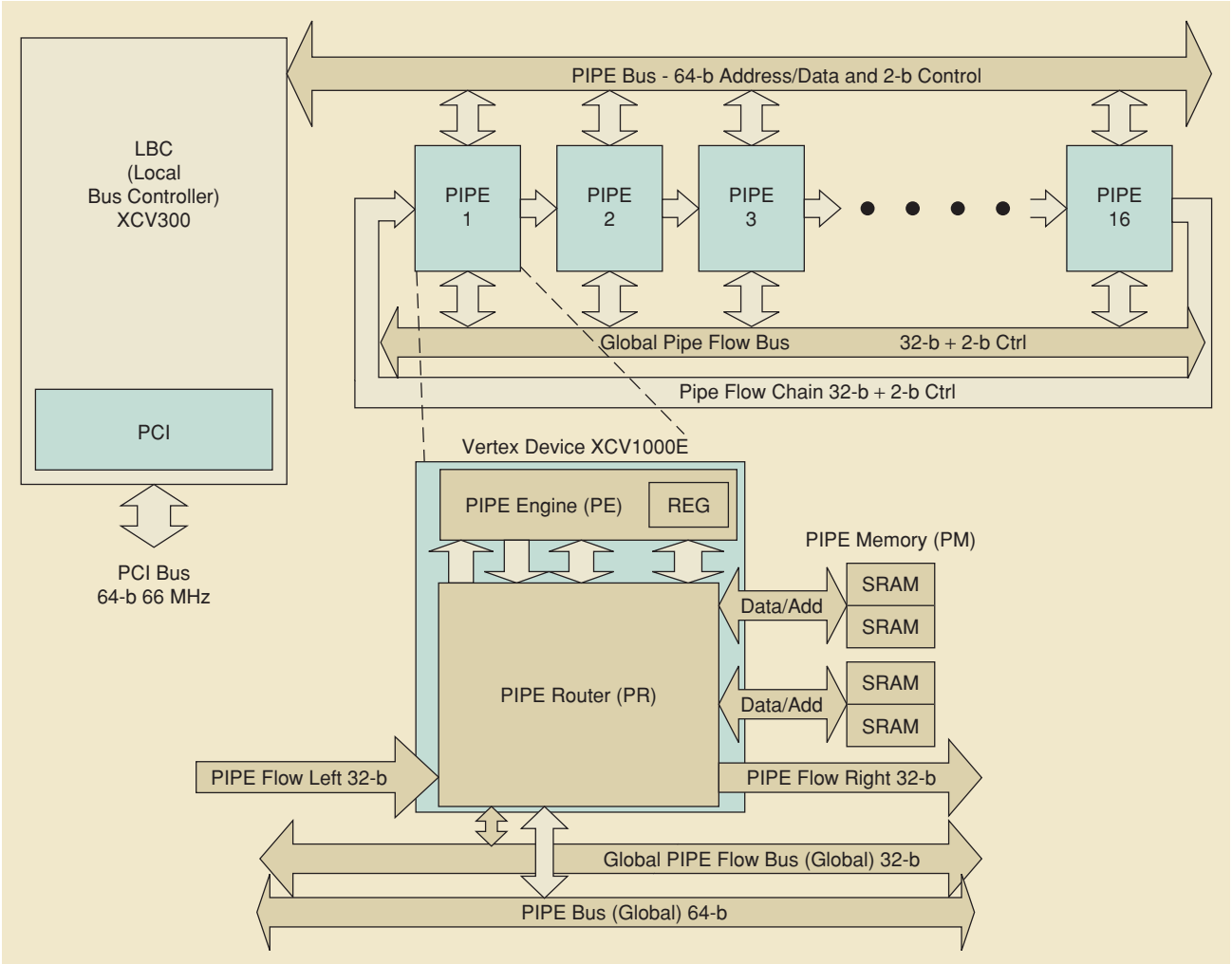
1053-5888/05/$20.00©2005IEEE

Our design environment involves an automated partitioning and scheduling algorithm to make a decision on where and when tasks are implemented and run. The CPS algorithm, collectively named for the three main steps of cluster, partition, schedule, is proposed to find the minimum processing time under a specified set of real-world conditions and constraints (communication time, memory conflict, and bus conflict), which are not addressed in [3]. (Note that processing time in this context is the time to process all tasks in the system. It has the same meaning as the maximum completion time, total production time, schedule length, or makespan that can be found in some literature.) Results from the CPS algorithm are used at the implementation stage, where the application-independent *infrastructure* is available to facilitate designers. This predesigned infrastructure, provided as standard library modules, is responsible for the control/interface mechanism in the codesign system. For example, the designed tasks must be encapsulated by the standard task wrapper to be able to collaborate with the rest of the system. To control operations such as task executions, run-time reconfigurations, and data transfers, an automatically generated *task manager* program is used. This design approach reduces design errors and supports system modularity, scalability, and manageability for run-time reconfiguration.

THE ULTRASONIC RECONFIGURABLE PLATFORM

Our codesign environment targets UltraSONIC [4], a reconfigurable computing system designed to cope with the computational power and the high data throughput demanded by real-time video applications. The architecture exploits the spatial and temporal parallelism in video processing algorithms. It also facilitates design reuse and supports the software plug-in methodology.

The structure of the board is shown in Figure 1. The system consists of plug-in processing element (PIPE) modules that perform the actual processing. The standard PIPE contains an XCV1000E FPGA and 4 × 2 MB SRAM. The PCI bus connects the UltraSONIC board to a host PC. Data transfers between the UltraSONIC and the host PC are performed over this PCI bus. (Note that the UltraSONIC main board is a universal PCI card, meaning that it can operate at 66 or 33 MHz in a 32-or 64-b PCI



[FIG1] The architecture of the UltraSONIC reconfigurable platform.

slot running at 5 or 3.3 V, controlled by the local bus controller.) On the board, there is one global bus, called the PIPE bus (PB), and two local buses, called the PIPE flow global (PFG) and PIPE flow chain (PFC).

Each PIPE consists of three main parts:

- PIPE engine: handles computations specified by the user. PE registers are used to receive parameter values from the host during computational processes.
- PIPE router: responsible for data movement. Routes are programmable via the internal PR registers. It consumes around 10% of resources in an XCV1000E device.
- PIPE memory: provides local buffering of data. Two independent data/address ports are provided for two equal memory blocks (4 MB each).

OUR ENVIRONMENT SUPPORTS AUTOMATIC PARTITIONING AND SCHEDULING BETWEEN A HOST PROCESSOR AND A NUMBER OF RECONFIGURABLE PROCESSORS.

SYSTEM SPECIFICATIONS AND MODELS

The reconfigurable hardware element of our codesign system is the programmable FPGAs. Run-time reconfiguration is supported in this reconfigurable platform, which does increase the complexity of our system. Features that we are concerned with in our reconfigurable system are as follows:

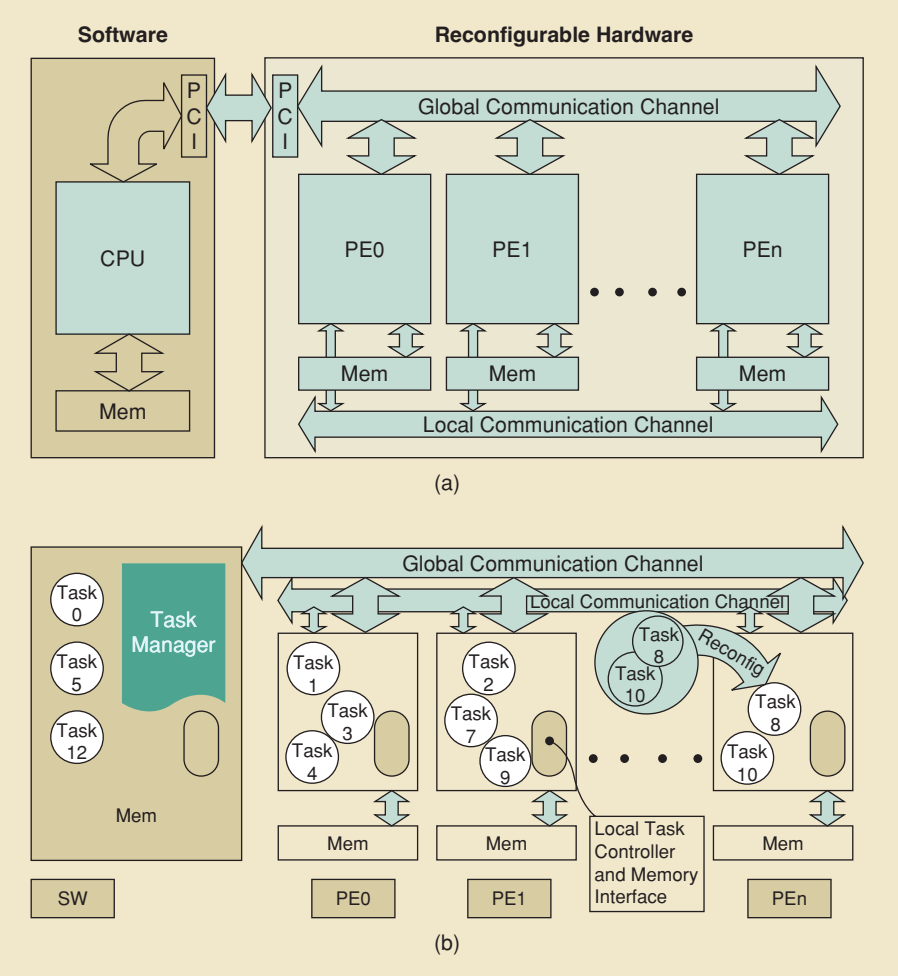
- There are many hardware processing elements (PEs) that are reconfigurable.
- Reconfiguration can be performed at run-time.
- All parameters such as the number of PEs, the number of gates on each PE, communication time, and configuration time are taken into account.
- On each PE where the number of logic gates is limited, hardware tasks may have to be divided into several temporal groups that will be reconfigured at run-time.
- Tasks must be scheduled without conflicts on shared resources, such as buses or memories.

SYSTEM MODEL

As shown in Figure 1, the target system is a loosely coupled model by nature, which means that there is no shared memory used as a medium for transferring data. There is only local memory on each PE. To transfer data between PEs, a communication channel is directly established between both ends that must communicate.

The target system can be modeled as a system consisting of a single software element and multiple reconfigurable hardware elements [see Figure 2(a)]. There are two main types of buses in this system: global and local buses. Configurations can be completed through the global bus. This global bus is also used as a communication channel between hardware and software. However, transferring data between the hardware PEs is done through the local bus. Each PE has its own local memory for storing input and output data for all internal tasks. In addition, there is a well-established mechanism for users to control PEs from the host processor based on a set of well-defined application program interface (API) functions.

Tasks in a system process are dynamically implemented and executed (once step for each task) either in the CPU or PEs, according to precedence relationships and priorities. All operations are controlled and



[FIG2] The UltraSONIC system. (a) The system model. (b) An example of task implementation.

initiated by a task manager program running in software. There exists a local controller in each PE to interact with the task manager and to be responsible for all local operations such as executing tasks, memory interfacing, and transferring data between PEs. An example of implementing tasks in the system is given in Figure 2(b).

TASK MODEL

The tasks that we implement in our system are assumed to conform to the following restrictions:

- Software and hardware tasks are built uniformly to be performed under the same control mechanism. This simplifies system management and task swapping.
- Tasks implemented in each hardware PE are *coarse-grain* tasks, which may consist of one or more functional tasks (blocks or loops).
- Communication between tasks is always through local single port memory used as buffers between tasks.
- Tasks for a PE may be dynamically swapped in and out using dynamic reconfiguration.

There are different types of tasks to be specified in this system: *normal*, *software-only*, *hardware-only*, and *dummy* tasks. A normal task is free to be partitioned and scheduled either in hardware or software resources. A software-only task is a task that users intentionally implement in software without exception. Similarly, a hardware-only task is implemented solely in hardware. A dummy task is either *source* or *sink* for inputting and outputting data, respectively, and is not involved in any computation. In our system, we assume that inputs and outputs are initially provided and written to the microprocessor memory; a dummy task is therefore a software task by default.

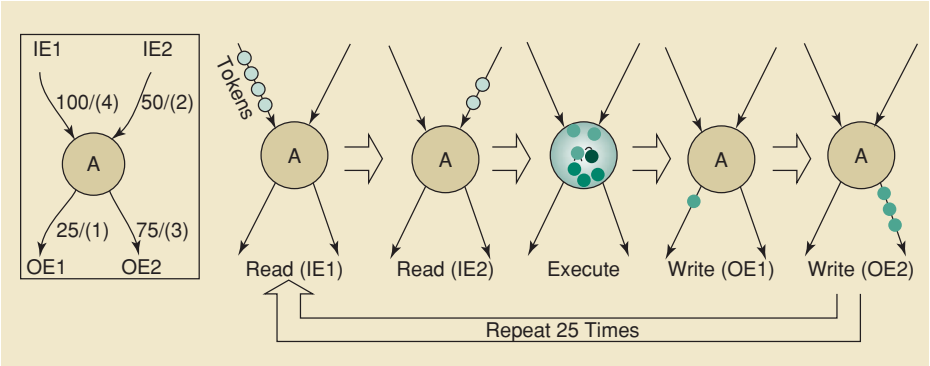
EXECUTION CRITERIA

After tasks have been loaded into a PE and are ready to be processed, they cannot be interrupted in the middle of the execution process; in other words, they are nonpreemptive. Task execution is completed in three consecutive steps: read input data, process the data, and write the results. This is done repeatedly until input data stored in memory is completely processed. Thus, the communication time between the local memory and the task (while executing) is considered to be a part of the task execution time [5]. Also, resource conflicts must be prevented, which means the same shared resource (such as memory or bus) is only available for use by one task.

The main restriction of this execution model is that exactly one task in a given PE is active at any one time. This is a direct consequence of the single-port memory restriction that allows one task to access the memory at any given time. However, multiple tasks can run concurrently when they are mapped to different PEs. This is an improvement over the model proposed by others in [6].

FIRING RULES

We adopt the rules by which data is processed by a task from the *synchronous data flow* (SDF) computational model [7]. However, in this work, our tasks are coarse-grained tasks represented in a DAG and we use the PE local memory as buffers between nodes, replacing the FIFO in [7] and [8]. As a result, tasks mapped to the same PE must be executed sequentially to avoid memory conflict. If a task requires a large amount of input data, the data must be sliced into sufficiently small units for processing to take place. The task is fired repeatedly as read-execute-write cycles until *all* data for the task has been processed.



[FIG3] Example of firing process.

An example of our task execution model is shown in Figure 3. In this example, task A, which has two incoming edges (IE1, IE2) and two outgoing edges (OE1, OE2), is a task to be fired. There are 100 tokens from IE1 and 50 tokens from IE2 to be executed. The number shown inside the parentheses on each edge is the number of data values needed for each firing iteration, representing the consuming rate or producing rate of a task. For instance, the consuming rate on IE1 of task A is four, while the producing rate on OE1 is one.

To process data, task A first reads four tokens from IE1 followed by two tokens from IE2. This must be performed sequentially because these inputs are stored in the same single-port local memory used as buffers on each edge. All of these input tokens are stored inside the node before being processed. When execution is completed, one token is written out to OE1, followed by three tokens to OE2. These steps are performed repeatedly until all data (shown as the first number on the edge) is processed.

CODESIGN ENVIRONMENT

Figure 4 depicts the codesign environment of the UltraSONIC system. It is divided into the front-end and the back-end stage. The front end is responsible for system specifications, input intermediate format, and system partitioning/scheduling. The back-end involves hardware/software task design and implementation, design verifications, control mechanisms, and system debugging.

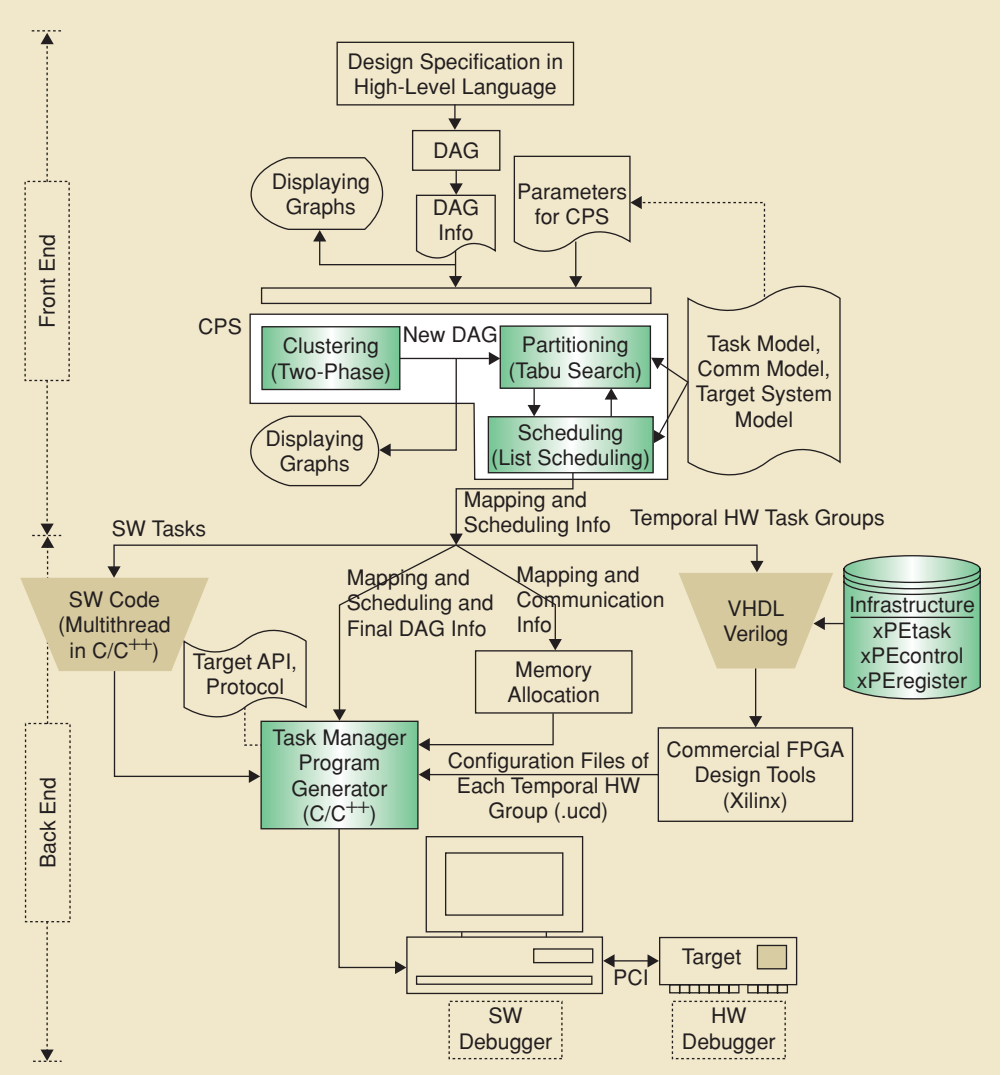
At the front end, the design to be implemented is assumed to be described in a suitable high-level language, which is then mapped to a DAG at coarse-grained level. Nodes and edges in the DAG represent tasks and data dependencies, respectively. The group of algorithms known as the *CPS* algorithm reads a textual input file that includes DAG information and parameters for the clustering, partitioning, and scheduling process. During this input stage, users can specify the type of tasks as normal, software-only, hardware-only, or dummy tasks.

After obtaining the results of the partitioning and scheduling process, which are the physical and temporal bindings for each task, we can start the back-end design implementation phase. In the case of hardware tasks, they may be divided into many temporal groups that can either be statically mapped to a hardware resource or dynamically configured during run-time.

We currently assume that software tasks are manually written in C/C++, while hardware tasks are designed manually in a hardware description language (such as Verilog in this work) using a library-based approach. Once all the hardware tasks for a given PE are available, they are wrapped in a predesigned circuit, called *xPEtask*, which is application independent. Commercially available synthesis and place-and-route tools are then used to produce the final configuration files for each hardware element. Each task in this implementation method requires some hardware overhead to implement the task frame wrapper circuit. Therefore, our system favors partitioning algorithms that generate coarse-grained tasks.

The results from the partitioning and scheduling process, the memory allocator, the task control protocol, the API functions, and the configuration files of hardware tasks are used to automatically generate the codes for the task manager program that controls all operations in this system (such as dynamic configuration, task execution, and data transfer). The resulting task manager is inherently multithreaded to ensure that tasks can run concurrently where possible.

In the following section, the important parts of the codesign environment, including the CPS algorithm, the task manager program, and the infrastructure, are described.



[FIG4] The proposed codesign environment.

THE CPS ALGORITHM

The CPS algorithm in the front-end stage plays an important role in this codesign system. It helps designers in task clustering, partitioning, and scheduling, which are known as intractable problems [9]. The CPS method is a combination of three heuristic algorithms: the two-phase clustering, the tabu search, and the list scheduling. This combination is designed to obtain a good result in a reasonable time frame.

The two-phase clustering algorithm [10] is used as a preprocessing step to modify the granularity of tasks and enable more task parallelism. On average, it has been shown to achieve 15% shorter processing time for different task granularities. A new, smaller DAG with coarser-grained tasks is then partitioned and scheduled in order to obtain the minimum processing time.

The heuristic algorithm, based on tabu

search, is used to partition tasks into software and hardware [11]. It has been modified for this real system, which has a search space of K^N (where K is the number of processing elements and N is the number of tasks). This exponentially increasing search space makes an exhaustive search impractical. Although heuristic search could yield a near-optimal solution, convergence speed could be greatly improved by using a good initial guess.

OUR DESIGN ENVIRONMENT INVOLVES AN AUTOMATED PARTITIONING AND SCHEDULING ALGORITHM TO MAKE A DECISION ON WHERE AND WHEN TASKS ARE IMPLEMENTED AND RUN.

The list scheduler is used to order tasks, without any shared resource conflicts, with regard to partitioning results, task precedence, and the target system model. Our scheduling process has a tight relationship with the partitioning process. It is used as a cost function to examine the processing time of the guessed solutions from the partitioner. After the processing time information is obtained, it is sent back to guide the partitioner to explore only promising regions. This iteration process between partitioning and scheduling to minimize processing time will terminate when the stop condition (such as the number of iterations specified by the designer) is met.

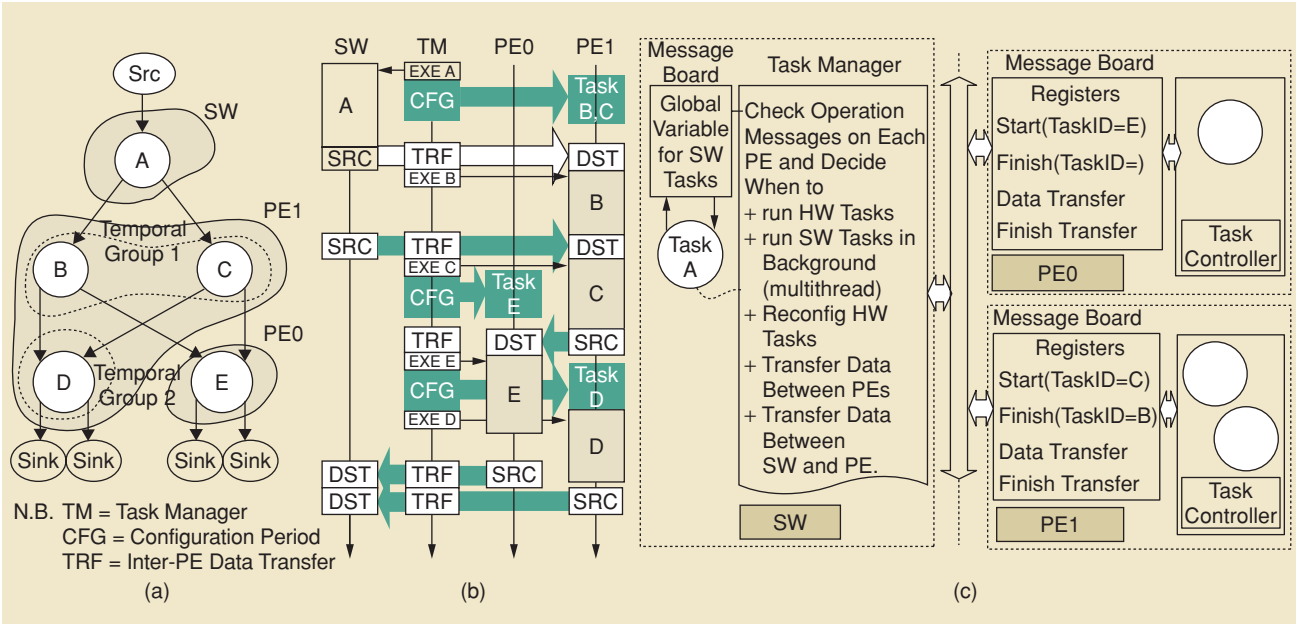
THE TASK MANAGER

Two main control methods in a distributed architecture are centralized control and distributed control [12]. In this work, we choose to implement the former due to its simplicity and good matching to the PC host processor in our system. The centralized control task manager program is used to orchestrate the sequencing of all hardware and software tasks, the transfer of data and the synchronization between them, and the dynamic reconfiguration of FPGAs in PEs when required. Because this program runs on the software processor (a host PC), the service time of which is uncertain and depends on several unpredictable factors, a time-triggered method is not suitable for real-time control actions. To properly synchronize the execution of the tasks and the communication between tasks, our task manager employs an event-triggered protocol when running an application. However, unlike a reactive codesign system [13], we do not regard external real-time events as triggers. Instead, we use the termination of each task execution or data transfer as event triggers, and the signaling of such events is done through dedicated registers.

With a single CPU model, the software processor must run simultaneously the task manager and the software tasks. A multithreaded programming technique is then employed to run these two types of processes concurrently. A mutex (short for mutual exclusion) is a way of communicating among threads that are executing asynchronously.

EXAMPLE

Figure 5(a) shows an example of a DAG. After the partitioning and scheduling process, operation sequences to run the DAG can be obtained as shown in Figure 5(b). This information is used to automatically generate the task manager program based on the message-based, event-triggered protocol as described earlier. In this example, the task manager first sends a message to execute task A in the processor (SW). Consequently, the configuration process runs to load the temporal group of tasks B and C into PE1. The task manager waits until task A is finished before initiating data transfer between SW and PE1, preparing



[FIG5] (a) DAG example, (b) operation sequences of the task manager, and (c) the task manager control view.

for task B to be executed next. This example also shows that there are two hardware temporal groups for PE1 that will be reconfigured at run-time.

TO CONTROL OPERATIONS SUCH AS TASK EXECUTIONS, RUN-TIME RECONFIGURATIONS, AND DATA TRANSFERS, AN AUTOMATICALLY GENERATED TASK MANAGER PROGRAM IS USED.

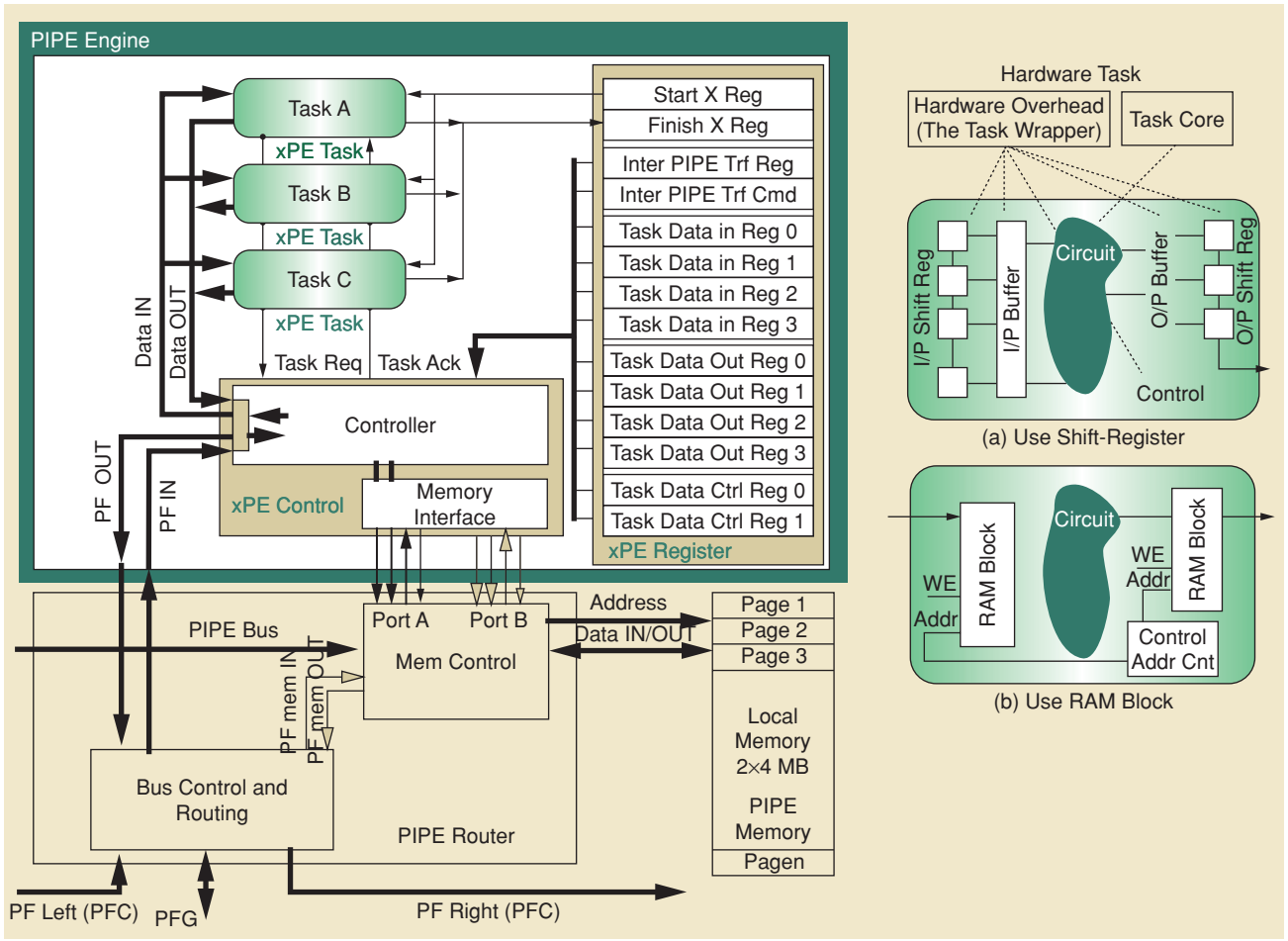
Figure 5(c) shows the conceptual control view of the task manager and its operations. The task manager communicates with a local task controller on each PE in order to assert control. A message board is used in each PE to receive commands from the task manager or to flag finishing status to the task manager. As can be seen, a message indicating execution completion from task B is posted to a specific register inside PE1. The task manager program polls this register, finds the message, and then proceeds to the next scheduled task (in this case, task C). Using this method, tasks on each PE run independently because the program operates asynchronously at the system level.

LIMITATIONS

At present, the task manager program, which is based on an asynchronous control protocol and runs in software, is not applicable to process streaming data that usually employs a pipelined structure. We cannot initiate several messages to run tasks in different pipeline stages simultaneously from the manager program, which is a sequential software code. However, one possibility to extend this work is to use special hardware nodes that can handle real-time pipelining to control operations, rather than the task manager program [14].

DESIGN INFRASTRUCTURE: THE TASK WRAPPER

Both hardware and software tasks are designed manually based on the results of partitioning. However, we provide an infrastructure to assist designers. A task core will reside in a standard predesigned structure called the *task wrapper,* which is responsible for cooperating with the task manager and local controllers. The software task wrapper is an automatically generated code that partly resides in the task manager program. The hardware task wrapper is a predesigned HDL code with generic parameters for different tasks with different consumption and production rates.



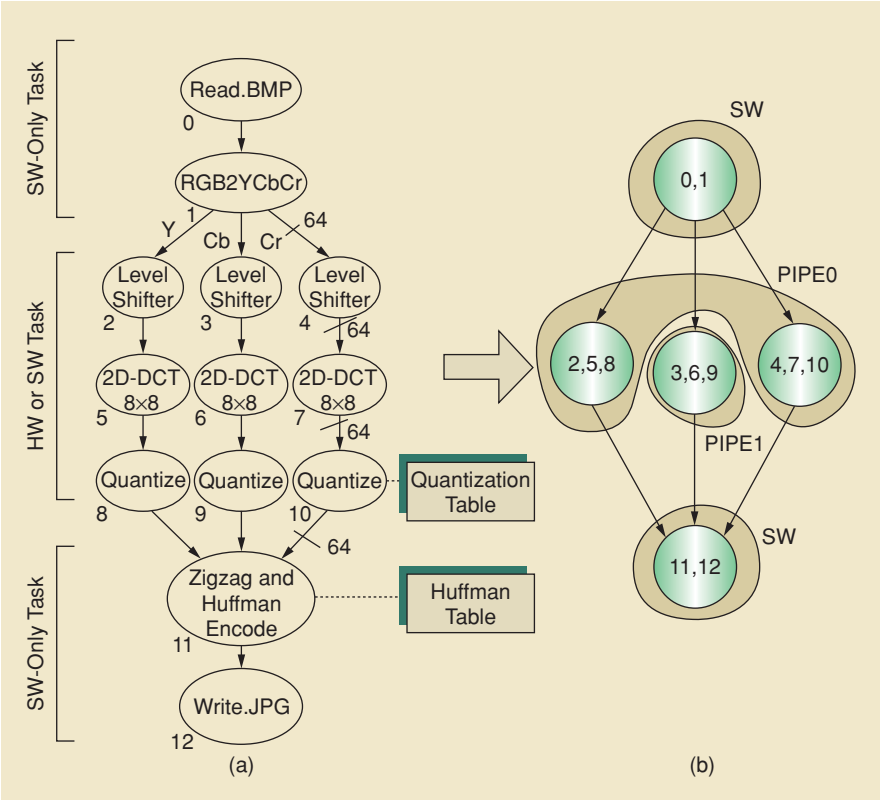
[FIG6] The hardware design structure in each PIPE.

By conforming to a set of design rules, designers can concentrate on constructing the task cores without having to worry about the interfacing or controlling mechanisms. This infrastructure facilitates fast design cycles and reduces error-prone aspects of the design process.

IMPLEMENTATION AND RESULTS

This section first provides implementation details of the proposed codesign framework for the UltraSONIC system. We then describe a case study for our approach based on JPEG compression and present some experimental results.

Figure 6 shows how the predesigned infrastructure for hardware tasks is implemented in the UltraSONIC PIPE. The infrastructure consists of three application-independent modules: *xPEcontrol, xPEregister,* and *xPEtask*. Based on the information on xPEregister, xPEcontrol can control operations of all hardware tasks resident in xPEtask wrappers. The total hardware overhead of this infrastructure is modest. It consumes around 10% of the XCV1000E FPGA on each PIPE.



[FIG7] (a) DAG of JPEG compression algorithm implemented in this work. (b) The results after clustering and partitioning.

CASE STUDY: JPEG COMPRESSION

JPEG compression has been used as a real codesign application [15]–[16] for grey scale images, using a block size of 4×4 pixels. In this work, however, the standard JPEG compression [17] with block size of 8×8 pixels for color images is implemented using the DCT baseline method with sequential encoding.

The DAG of the JPEG compression algorithm is illustrated in Figure 7. Tasks such as *Read.BMP, RGB2YCbCr, Encoder,* and *Write.JPG* are implemented in software for convenience. The other modules (*Level Shifters, 2D-DCTs,* and *Quantizers)* can be implemented in either software or hardware. For hardware, the designed tasks are wrapped by the xPEtask wrapper with the RAM structure. Note that we employ a one-dimensional fast DCT architecture for eight pixels [18]. To deal with two-dimensional data blocks with size of 8×8, the DCT block is applied first in the horizontal direction, then the vertical.

EXPERIMENTAL RESULTS

For the software-only solution, C++ codes of every task module are written and run under the control of the task manager program. Table 1 shows execution times of the software-only solution for different sizes of pictures. The values are averaged from 20 runs on the UltraSONIC host, a PC with a Pentium II processor running at 450 MHz.

When using hardware to alleviate computational jobs in this JPEG algorithm, we can substantially reduce the processing time by around 83% on average for given image sizes (see Table 2). To accomplish this, the tasks, including *Level Shifter, 2D-DCT,* and

[TABLE 1] EXECUTION TIMES OF TASKS RUNNING IN SOFTWARE.

		EXECUTION TIME (ms)							
IMAGES (PIXELS)	**BMP SIZE**	**READ .BMP**	**RGB2 YCbCr**	**LEVEL SHIFTER***	**2D-DCT***	**QT***	**ENCODER**	**WRITE JPEG**	**JPEG SIZE**
400 × 300	352 KB	35.5	130.2	12.7	834.9	67.4	193.9	4.0	11 KB
640 × 480	901 KB	90.1	360.5	32.1	2193.4	192.0	481.3	8.0	18 KB
800 × 600	1,407 KB	151.3	572.8	50.1	3437.3	299.1	835.1	8.0	55 KB
800 × 800	1,876 KB	186.1	758.0	63.7	4690.5	325.7	1121.7	11.0	63 KB
1,024 × 768	2,305 KB	257.3	862.1	86.8	5894.4	398.0	1402.0	15.0	70 KB
PERCENTAGE	–	*1.14%*	*4.24%*	*1.18%*	*80.90%*	*6.08%*	*6.38%*	*0.07%*	–

*The average time to process *one* color component.

[TABLE 2] COMPARISONS OF PROCESSING TIME OF JPEG COMPRESSION.

IMAGES (PIXELS)	SOFTWARE SOLUTION	HARDWARE/SOFTWARE SOLUTION*
400 × 300	3.15 s	0.57 s
640 × 480	8.23 s	1.34 s
800 × 600	12.96 s	2.04 s
800 × 800	17.38 s	2.71 s
1,024 × 768	21.71 s	3.26 s

*Using only two PIPEs available in the system.

Quantizer (which consume about 88% of processing time if all tasks are run in software), are all moved into hardware. This move results in substantial processing time reduction. The correct results of every image size are produced.

We also accomplish the implementation of the 8- and 16-point FFT algorithms, which contain 24 and 52 task nodes, respectively. The radix-2 butterfly node is represented as a task in the implementation. Unfortunately, the details cannot be exhibited in this article due to space constraints.

SUMMARY

This article introduces and demonstrates a task-based hardware/software codesign environment specialized for real-time video applications. Both the automated partitioning and scheduling environment (the predesigned infrastructure in the form of wrappers) and the task manager program help to provide a fast and robust route for supporting demanding applications in our codesign system. The UltraSONIC reconfigurable computer, which has been used for implementing many industrial-grade applications at SONY and Imperial College, allows us to develop a realistic system model. Many simplifying assumptions found in previous research, such as zero communication overhead and no possible resource conflicts, become unnecessary.

Current and future work includes improvement of our task execution model, which uses local memory as a shared buffer for hardware tasks on each PE. This limits the possible degree of concurrency within a PE. The task manager should also be improved for better concurrency. Furthermore, we plan to extend our ideas here to cover the SONIC-on-a-chip system [19], which would require improving our system model at different levels.

AUTHORS

Theerayod Wiangtong received the B.Eng. degree in electronic engineering from KMITL (King Mongkut's Institute of Technology Ladkrabang) in 1993, the M.Sc. degree in satellite communication from the University of Surrey in 1996, and the Ph.D. degree in digital system design (codesign) from Imperial College, London, in 2004. He is now working as a lecturer in the electronic department at Mahanakorn University of Technology, Thailand. His research areas include digital VLSI design, optimization techniques, hardware/software codesign, reconfigurable computing, and FPGA implementations for various applications.

Peter Y.K. Cheung is the deputy head of the Electrical and Electronic Engineering Department at Imperial College, University of London, where he is professor of digital systems. His research interests include VLSI architectures for DSP and video processing, reconfigurable computing, embedded systems, and high-level synthesis and optimization of digital systems, particularly those containing field-programmable logic.

Wayne Luk is a professor of computer engineering in the Department of Computing, Imperial College, London, and leads the Custom Computing Group. His research interests include theory and practice of customizing hardware and software for specific application domains, such as graphics and image processing, multimedia, and communications. Much of his current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays.

REFERENCES

[1] A. Kalavade and E.A. Lee, "A hardware/software codesign methodology for DSP applications," *IEEE Des. Test Comput.*, vol. 10, no. 3, pp. 16–28, Sept. 1993.

[2] S.S. Bhattacharyya, "Hardware/software co-synthesis of DSP systems," in *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, Y.H. Hu, Ed. New York: Marcel Dekker, 2001, pp. 333–378.

[3] H. Oh and S. Ha, "Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints," in *Proc. Int. Workshop Hardware/Software Codesign*, May 2002, pp. 133–138.

[4] S.D. Haynes, H.G. Epsom, R.J. Cooper, and P.L. McAlpine, "UltraSONIC: A reconfigurable architecture for video image processing," in *Proc. Field-Programmable Logic and Applications*, 2002, pp. 482–491.

[5] T. Pop, P. Eles, and Z. Peng, "Holistic scheduling and analysis of mixed time/event triggered distributed embedded systems," in *Proc. Hardware/Software Codesign*, 2002, pp. 187–192.

[6] J. Hou and W. Wolf, "Process partitioning for distributed embedded systems," in *Proc. Hardware/Software CoDesign*, 1996, pp. 70–76.

[7] E.A. Lee and D.G. Messerschmitt., "Synchronous dataflow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept. 1987.

[8] E.A. Lee and T.M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[9] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[10] T. Wiangtong, P.Y.K. Cheung, and W. Luk, "Cluster-driven hardware/software partitioning and scheduling approach for a reconfigurable computer system," in *Proc. Field-Programmable Logic and Applications,* 2003, pp. 1071–1074.

[11] T. Wiangtong, P.Y.K. Cheung, and W. Luk, "Comparing three heuristic search methods for functional partitioning in HW-SW codesign," in *Proc. Int. J. Design Automat. Embedded Syst.*, vol. 6, pp. 425–449, 2002.

[12] P. Chou, K. Hines, K. Partridge, and G. Borriello, "Control generation for embedded systems based on composition of modal processes," in *Proc. Computer-Aided Design*, 1998, pp. 46–53.

[13] G.D. Micheli, "Computer-aided hardware-software codesign," *IEEE Micro*, vol. 14, no. 4, pp. 10–16, Aug. 1994.

[14] M. Eisenring and M. Platzner, "An implementation framework for run-time reconfigurable systems," in *Proc. Workshop Engineering of Reconfigurable Hardware/Software Objects, ENREGLE*, 2000, pp. 151–157.

[15] K.S. Chatha and R. Vemuri, "Hardware-software partitioning and pipelined scheduling of transformative applications," *IEEE Trans. VLSI Syst.*, vol. 10, no. 3 pp. 193–208, June 2002.

[16] N. Narasimhan, V. Srinivasan, M. Vootukuru, J. Walrath, S. Govindarajan, and R. Vemuri, "Rapid prototyping of reconfigurable coprocessors," in *Proc. ASAP, Application Specific Systems, Architectures and Processors*, 1996, pp. 303–312.

[17] G.K. Wallace, "The JPEG still picture compression standard," *IEEE Trans. Consumer Electron.*, vol. 38, no. 1, pp. 18–34, 1992.

[18] B.G. Sherlock and D.M. Monro, "Fast discrete cosine transform," *ACM Trans. Math. Softw.*, vol. 21, no. 1, pp. 372–378, Dec. 1995.

[19] P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, and W. Luk, "A reconfigurable platform for real-time embedded video image processing," in *Proc. Field-Programmable Logic and Applications*. 2003, pp. 606–615.