Artificial Deadlock Detection in Process Networks for ECLIPSE

Bharath.N and S.K.Nandy
CAD Laboratory
Supercomputer Education & Research Centre
Indian Institute of Science
Bangalore, India -560 012
{bharath@cadl,nandy@serc}.iisc.ernet.in

Nagaraju Bussa
Philips Research India - Bangalore
Philips Innovation Campus
No.1 Murphy Road, Ulsoor
Bangalore, India - 560 008
nagaraju.bussa@philips.com

Abstract

Kahn Process Network (KPN) is a popular model of computation for describing streaming applications. In a KPN model, processes communicate through unbounded unidirectional FIFOs. When theoretically unbounded FIFOs are implemented using finite memory, artificial deadlocks can occur due to one or more FIFOs having insufficient sizes. Generally a system designer must be able to make a design time trade-off between execution time and memory usage, preferably using no more memory than required for obtaining a certain execution time. But it is practically impossible to decide at design time, FIFO sizes that are sufficient to run the application without any artificial deadlocks. Hence there is a need for runtime mechanism for handling the artificial deadlock situations in process networks. Existing mechanisms detect artificial deadlocks only after all KPN processes block. This results in excessive blocking of processes and an application that appears to 'hang'. In this paper we present an improved mechanism for early detection of artificial deadlocks and its implementation on ECLIPSE (Extended CPU Local Irregular Processing ArchitecturE), an application domain specific architecture.

1. Introduction

Media and signal-processing applications in consumer electronics and telecommunications are represented as a collection of cooperating sequential processes that communicate via streams of data. Typical examples of such applications are MPEG2 encoding and decoding [16]. Kahn Process Networks(KPN) [6] is a popular model of computation to represent such streaming applications. KPN constitutes a collection of sequential processes that communicate through unidirectional unbounded FIFOs. These FIFOs buffer data streams between processes and facilitate parallel computation. The major advantage of KPNs is that they explicitly specify (task-level) parallelism and communication in an application, thereby making it very suitable for execution on multi-processor architectures. KPNs are fully compositional and hence facilitate the reuse of application models, which is very essential for reducing time-to-market for new products.

Figure 1 represents an example KPN. Here, nodes denote the processes of the KPN and the edges represent the FIFO communication channels between the processes. The direction of the edge denotes the direction of communication between the processes.

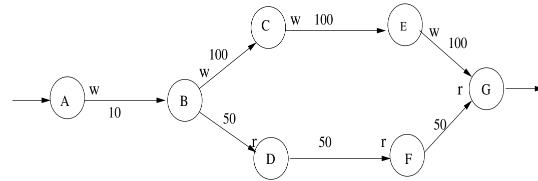

Figure 1. Example KPN.

An implementation of the KPN has to execute using finite memory. Following [2] we use the term process network (PN) to refer to a network of processes with bounded FIFO sizes. To effectively map and execute the KPNs on multiprocessor systems, two issues arise: (a) how to find an efficient schedule for the processes to execute and (b) how to allocate sufficient memory to the FIFOs to achieve a complete execution of the PN (output matches that produced by the unbounded KPN). These two issues are related - the amount of memory allocated to the FIFOs can possibly influence the execution order of the processes of the PN [2]. Insufficient FIFO size can cause processes to block on a write to a FIFO and lead to artificial deadlocks (all processes in the PN block and at least one process is blocked on write to a full FIFO) during the execution of PNs [11]. Figure 1 shows the example PN in artificial deadlock. The label on the arc denotes the FIFO size. Processes are blocked either on a write to a full FIFO, denoted by $'w'$; or a read to an empty FIFO, denoted by $'r'$.

Existing mechanisms [11] [2] detect artificial deadlocks

Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)
1063-6862/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

after all processes in the PN block. This leads to excessive blockings for the processes and limits the parallelism in execution. Also both [11] [2] require a centralized mechanism for monitoring the status of processes in a PN and cannot be implemented efficiently on a distributed template like ECLIPSE. In this paper we present a mechanism for the early detection of artificial deadlocks in Process Networks and its implementation on ECLIPSE (Extended CPU Local Irregular Processing ArchitecturE) [13], an application domain specific architecture. In our early detection mechanism all processes in the KPN can detect an artificial deadlock like situation, but only the reader and writer process of the target FIFO (the FIFO to be resized to resolve the deadlock) take action to resolve the deadlock.

2. ECLIPSE architecture

Martijn Rutten et. al present the details of the ECLIPSE architecture in [13]. Here we give a discussion of the ECLIPSE architecture. ECLIPSE is an architecture template for the design of high performance streaming media-processing SoC subsystems. ECLIPSE provides an architectural framework for on chip CPU and coprocessor communication, combining application configuration flexibility with the efficiency of function specific hardware (coprocessors).

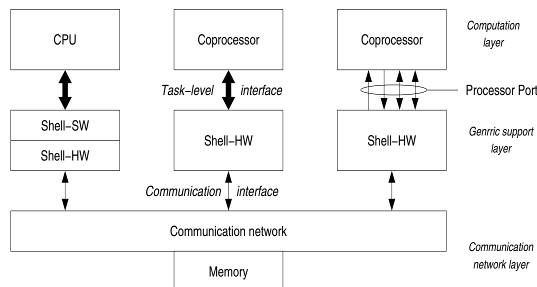

Figure 2. Coprocessor shell in ECLIPSE.

Martijn Rutten et. al introduce the concept of coprocessor shell to facilitate reuse of coprocessor designs over different ECLIPSE instances with different communication network characteristics [12]. Figure 2 depicts this hardware interface block that separates the computation hardware (coprocessors) from the communication hardware (buses, memory). The shell eases coprocessor design by absorbing many system-level issues, such as multitasking, stream synchronization, and data transport. Each coprocessor interacts with its shell through five generic interface primitives `GetTask, Read, Write, GetSpace, PutSpace`. `GetTask` allows a task switch to another task mapped on the same coprocessor. `Read` and `Write` are primitives for accessing data in the stream buffer; `GetSpace` and `PutSpace` are synchronization primitives for accesses to data in the stream buffer. ECLIPSE implements these primitives in hardware using a master-slave handshake protocol with the corresponding argument and result passing between coprocessors and their shells.

All tasks ports map to one physical coprocessor-shell interface that handles `Read, Write, GetSpace/PutSpace`, and `GetTask` requests in parallel. Communicating a stream of data requires a FIFO buffer, which in ECLIPSE has a finite size. The shell applies a cyclic addressing mechanism for proper FIFO behavior in the linear memory address range, using the `n_bytes` and `offset` arguments of the `Read/Write` calls in addition to the current access point and the buffer size.

Media processing applications for ECLIPSE are specified as set of concurrently executing tasks that exchange information solely by unidirectional streams of data using a KPN [6] based modeling framework called YAPI (Y-chart application programmer interface) [4]. Once a set of basic functions is defined as tasks, a multitude of applications can be configured by instantiating tasks and connecting them in a KPN graph structure.

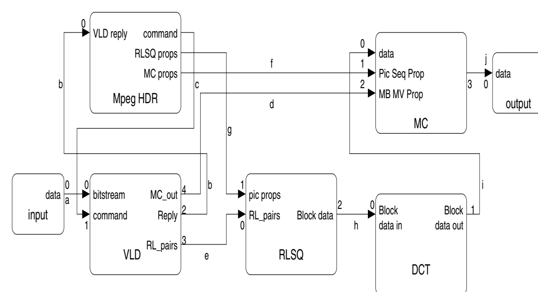

Figure 3. MPEG-2 decoder application graph in ECLIPSE.

Figure 3 shows the MPEG2 decoder application for ECLIPSE modeled as a PN. The processes are `MpegHdr` (header processing), `MC` (motion compensation), `DCT` (discrete cosine transform), `VLD` (variable length decoding), `RLSQ` (runlength decoding), `input` and `output`.

3. Artificial Deadlocks in ECLIPSE

In ECLIPSE task scheduling is distributed [12]; the tasks are scheduled independently by their respective shells. The coprocessors are loosely coupled, implying that within the time-scale that the buffer can bridge, scheduling of tasks on one coprocessor is independent of the instantaneous scheduling of tasks on other coprocessors. On a larger time scale, distributed unsynchronized scheduling results in a mismatch in the rate of production and consumption of tokens for a FIFO and causes the producer process to block on a full FIFO eventually leading to an artificial deadlock.

We analysed an ECLIPSE instance configured as a MPEG-2 decoder (Figure 3). We profiled the application

1063-6862/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

to arrive at the minimum FIFO sizes for the FIFOs (Table 1).

Table 1. Minimum FIFO sizes for MPEG-2 decoder application in ECLIPSE

Fifo and Port Name	Size(in bytes)
IDCT_MC /block data out	512
RLSQ_IDCT/block data	256
VLD_RLSQ/rl_pairs	128
VLD_MC/MC out	256
MPEGHDR_RLSQ/rlsq_props	256
MPEGHDR_MC/MC props	32
INPUT_VLD/bit stream	256
MPEGHDR_VLD/command	64
VLD_MPEGHDR/reply	32

Table 2 describes the test input streams used to analyze the MPEG-2 decoder application. We execute the MPEG-2 application instance with the FIFO sizes assigned to their optimum. For inputs $train.m2v$ and $teeny.m2v$, the application deadlocks after decoding the first I frame; the application does not produce all the decoded output frames. For the other two input streams the application runs till end of the input stream and produces all the decoded output frames. Here we see that application behavior with respect to artificial deadlock is data dependent. By profiling an application with sample input streams the designer cannot decide on FIFO sizes that are sufficient to execute the application without artificial deadlocks for any input streams.

Table 2. Test Input streams for MPEG-2 decoder application in ECLIPSE

Input Stream Name	Type	Number of Frames	Frame Size	Frame Rate
Oslo.m2v	HD	30	1920X1088	25
Train.m2v	HD	30	1920X1152	25
Tennis.m2v	SD	8	704X526	25
Teeny.m2v	SD	19	720X576	25

4. Artificial Deadlock Detection in PNs

Parks' strategy [11] to deal with artificial deadlocks executes the process network until a global artificial deadlock occurs (all processes block and at least one of them is blocked on a write). Deadlock is resolved by increasing the size of the smallest full FIFO; this guarantees that all blocked writes in the PN are eventually removed [11]. In implementations of PNs based on Parks' scheduling algorithm [8] [9], a special thread is used to detect and initiate the deadlock resolution procedure. This strategy incurs runtime overhead to keep track of the number of processes blocked on a read or a write.

Basten and Hoogerbrugge [2] show that the smallest full FIFO in a process network is not necessarily the bottleneck causing the artificial deadlock. Increasing the smallest full FIFO will resolve the deadlock, but may lead to unnecessarily large FIFOs. The cause of a read or a write block in a deadlocked process network is always a unique other blocked process. Consequently it is possible to track the entire chain of causes (called causal chain) that lead to a blocked output process. If the PN has multiple output processes, there are multiple causal chains leading to each of the output processes. If these chains are disjoint, the bottlenecks in these chains are independent and the bottleneck FIFO in each chain may be enlarged. If multiple chains share a bottleneck FIFO, it is sufficient to enlarge a bottleneck FIFO only once.

Geilen and Basten [5] investigate the requirements for a runtime scheduler to achieve a correct and bounded execution of process networks. They distinguish between local and global deadlocks. A process network is in global deadlock if all the processes in a PN are blocked; if some processes are blocked and the rest of the network cannot initiate their progress then the PN is in a local deadlock.

In Parks' approach, artificial deadlock is said to have occurred when all processes block, some of them on a write to a full FIFO. Parks' method detects and responds to global artificial deadlocks only. This approach ensures that execution does not terminate due to artificial deadlocks, but does not guarantee output completeness (production of all output required by Kahn semantics).

Under the assumption of effective KPNs (every token written to a channel is eventually read), Geilen and Basten [5] show that chain of dependencies leading to a local deadlock is cyclic. Their scheduling strategy is as follows: execute the process network in a data driven fashion until an artificial local deadlock occurs; resolve artificial local deadlocks by increasing the smallest full FIFO involved in the local deadlock by a finite amount.

These methods are unsuitable for implementation on a distributed template like ECLIPSE due to their inherent centralized mechanism for deadlock detection. Also these methods detect the artificial deadlock after all processes in the PN block leading to excessive blocking of the processes; this limits the parallelism that can be achieved during execution of KPNs. In this paper we present a mechanism for early detection of artificial deadlocks and its implementation on ECLIPSE.

5. Early Detection of Artificial Deadlocks in PNs

Our methodology to detect artificial deadlocks builds on the result of [5] that the chain of causes leading to a local deadlock is cyclic. We present the following observations regarding artificial deadlocks in process networks.

The root cause of an artificial deadlock is a mutual wait-for dependency among processes during execution of the PN. The processes involved in a mutual wait-for situation

IEEE COMPUTER SOCIETY

are said to form a causal cycle. When the remaining processes of the PN end up blocking on some process involved in a causal cycle, a global artificial deadlock results. When the remaining processes of the PN continue execution despite the mutual wait causal cycle, a local artificial deadlock results. The key to detection of artificial deadlocks is the identification of the causal cycle during the execution of process networks.

We present a mechanism to detect causal cycles during the execution of the PN without waiting for all processes in the PN to block. The cyclic chain of causes (causal cycles) leading to a mutual wait dependency correspond to the cycles in the (undirected) graph of the PN. Given a PN graph, the cycles in it can be enumerated at compile time. We observe that causal cycles that correspond with the cycles in the PN graph have one of the patterns shown in Figure 4. This observation can be intuitively explained as follows: every blocked process A has a unique other blocked process B as the cause [2]. Hence a process A can be blocked waiting for (a) another process B to read from the connecting FIFO; here the process A is blocked on a write to a full channel or (b) another process B to write to the connecting FIFO; here the process A is blocked on a read from an empty channel.

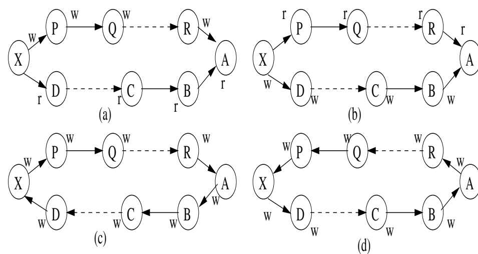

Figure 4. Possible causal patterns in a PN.

To summarize, the status of processes in a causal cycle as observed from processes X or A is one of the following (a) w^+r^+ (b) r^+w^+ (c) ww^+(rr^+ is not an artificial deadlock since there are no processes blocked on a full FIFO). We designate such processes as the monitor processes. While it is possible to enumerate the possible causal cycle patterns at compile time, it is not known which of these will actually lead to an artificial local deadlock. This is dependent on the input, the allocated FIFO capacity and the schedule of execution. Once the possible causal cycle patterns are known, deadlock detection reduces to monitoring the status of the PN for these causal patterns. To resolve the deadlock we increase the size of the smallest full FIFO in the causal cycle.

6. ECLIPSE Implementation

We extend the hardware shell of ECLIPSE to implement early deadlock detection as shown in Figure 5. We add hardware structures to aid in monitoring read-write blocks and causal cycle detection. The Block Monitor (BLK MONITOR) unit of the each shell maintains the instantaneous status of each port (whether the port is blocked or not blocked along with the time stamp information denoting the time instance of block) connected to its shell, at a globally accessible location. Deadlock Detection and Resolution (DD&R) unit implements monitoring for occurrence of causal cycles. Given an application modeled as a PN, the causal cycles in the PN graph are found using a modified version of the cycle finding algorithm [14] and listed in the application configuration file of ECLIPSE infrastructure. CPU configures the DD&R unit of each of the shells to monitor for occurrence of causal cycles for which the shell itself is a node. DD&R unit in each of the shells periodically accesses the global memory for finding the status of all ports and checks for formation of causal cycles.

The occurrence of a causal cycle is globally detected by all the shells and the target FIFO identified based on the following heuristic: (a) the smallest write blocked FIFO in the causal cycle is the target FIFO (b) if there are two write blocked FIFOs of the same size in the causal cycle, then the FIFO which got write blocked first is the target FIFO. The producer shell modifies the current write pointer in its Data Transport Write (DTW) unit and the consumer shell modifies the current read pointer in its Data Transport Read (DTR) unit to reflect the increased FIFO size. Also the memory controller checks for wrapping around in the target FIFO and accordingly moves data around when the target FIFO is resized to preserve the abstraction of an infinite FIFO to the processes. Other shells involved in the causal cycle also detect the artificial deadlock simultaneously but do not try to resolve since they neither read from nor write to the target FIFO. Deadlock detection and resolution is distributed and coprocessors can be programmed to independently monitor for causal cycles and resize the target FIFO upon artificial deadlocks.

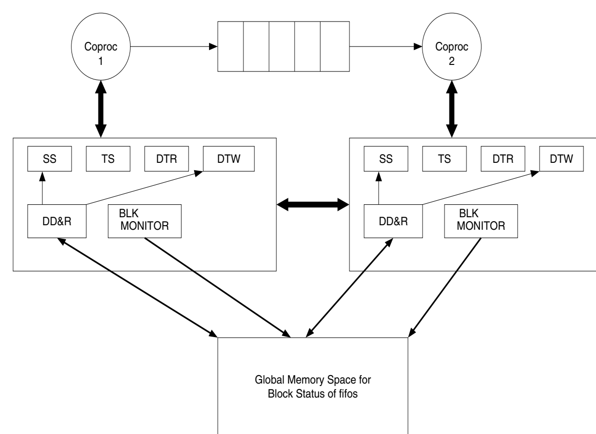

Figure 5. Block diagram of the ECLIPSE implementation.

Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)
1063-6862/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

We present simulation results for an ECLIPSE instance configured as MPEG-2 decoder running the test input stream $teeny.m2v$. We instrument the shells to sample the status of processes periodically during their execution and use a performance visualization tool to visualize the results. Figure 6 is a screen capture of the performance visualization tool showing the execution of processes in a native ECLIPSE instance with no deadlock detection. Each point on the X-axis denotes the sampling frequency and the Y-axis denotes the number of samples in the sampling interval for which the process was in execute/running state. We observe that the processes hang (deadlock) and execution stops after decoding the initial I frame. Processes `Mpeghdr` and `Vld` show execution activity since they are scheduled with the busy wait option. These processes do not relinquish the co-processor when a read or write block occurs, but instead keep polling till the block is resolved. Figure 7 shows the screen capture of the visualization tool for execution of processes in the modified ECLIPSE instance with early deadlock detection. The DD&R unit detects and resolves artificial deadlocks as and when they occur; all the tasks in the PN model continue to execute till all the decoded output frames are genearted.

7. Related work

The Compaan compiler automates the transformation of MATLAB code into KPN specifications for affine nested loop programs [7]. Out of order communication between the producer and consumer in the KPN model[1] requires buffering of data tokens on the FIFO channel to ensure functional correctness in KPN execution. Insufficient FIFO size can lead to write blocks for the producer process and result in artificial deadlocks. In Compaan, compile time tests determine whether the FIFO sizes are sufficient or additional memory needs to be allocated to the consumer process to buffer out-of-order data [15]. Such compile time analysis is possible only for KPNs with regular communication patterns. Our approach to artificial deadlocks in Process Networks is fundamentally different from Compaan. We use static analysis of the topology of the PN graph to determine the various possible artificial deadlock patterns during execution; runtime monitoring of the PN execution is required to detect the occurance of these causal patterns and resolve deadlocks.

In [1] Olson and Evans present a runtime approach to deadlock detection based on [10] algorithm. In this approach, exactly one process detects deadlock in a cycle of waiting processes. The deadlock detection scheme presented in this paper, is based on identifying and monitoring for occurence of causal patterns that lead to local artificial deadlocks. In our method every process in a causal chain (cycle of waiting processes) can simulatneously detect that it is involved in a deadlock. The way we gather information about the blocked processes is distributed and has some similarities with the Mitchell - Merrit method, except that we do this without maintaining labels in two steps, viz blocking step and transmit step. Instead our scheme is based on a finite state machine (FSM) to detect patterns that can possibly lead to artificial deadlocks [3].

[1] order in which data is written by the producer is different from the order in which consumer reads data

8. Conclusion

We presented a runtime mechanism for early detection of artificial deadlocks in process networks and described its implementation on ECLIPSE. Simulation results show that this method can correctly detect and resolve artificial deadlocks that arise during execution of process networks.

Acknowledgements

We thank Srinivas Gutta, Narendranath Udupa, Sainath Karlapalem, Martijn Rutten and Bart Vermeulen of Philips Research and the anonymous referees for their comments and suggestions.

References

[1] A.Olson and B.L.Evans. Deadlock detection for distributed process networks. In *Proceedings of the International Conference on Acoustics,Speech and Signal Processing*, 2005.

[2] T. Basten and J. Hoogerbrugge. Efficient execution of process networks. In *Proceedings of Communicating Process Architectures 2001*, 2001.

[3] Bharath.N and S.K.Nandy. A runtime mechanism for early detection of artificial deadlocks in process networks. In *Proceedings of the Special Session on System Design,MWSCAS - 2004*, 2004.

[4] E. de Kock et al. Yapi: Application modeling for signal processing systems. In *Proceedings of DAC - 2000*, 2000.

[5] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *Proceedings of the 12th European Symposium on Programming, ESOP 2003*, 2003.

[6] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 1974*, 1974.

[7] B. Kienhuis et al. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the eighth international workshop on Hardware/software codesign*, 2000.

[8] E. Lee. Overview of the ptolemy project. Technical Report Technical Memorandum UCB/ERL M01/11, University of California, EECS Dept., Berkeley, 2001.

[9] M.Goel. Process networks in ptolemy II. Master's thesis, University of California, EECS Dept., Berkeley, 1998.

[10] D. P. Mitchell and M. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, 1984.

[11] T. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, EECS Dept., Berkeley, CA, 1995.

1063-6862/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

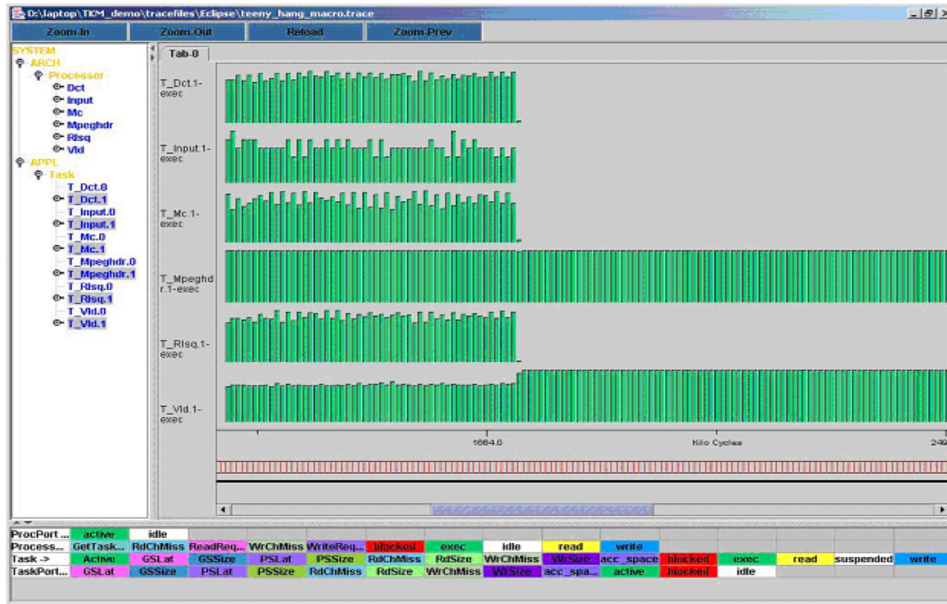

Figure 6. MPEG-2 decoder on ECLIPSE in artificial deadlock

Figure 7. Artificial deadlock detection in ECLIPSE for MPEG-2 decoder

[12] M. J. Rutten et al. Design of multi-tasking coprocessor control for eclipse. In *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002.

[13] M. J. Rutten et al. Eclipse: Heterogeneous multiprocessor architecture for flexible media processing. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, 2002.

[14] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, 1970.

[15] A. Turjan et al. A compile time based approach for solving out-of-order communication in kahn process networks. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, 2002.

[16] P. van der Wolf et al. An mpeg-2 decoder case study as a driver for a system level design methodology. In *Proceedings of the seventh international workshop on Hardware/software codesign*, 1999.

1063-6862/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY