Hardware Context switching in a signal processing application for an FPGA Custom Computer

David Kearney and Raymond Kiefer

School of Computer and Information Science,
University of South Australia,
The Levels, South Australia 5095,
AUSTRALIA
E-mail: { kearney, kiefer}@cis.unisa.edu.au

Abstract. Context switching of hardware in an FPGA custom computer is evaluated using a signal processing application targeted at the SPACE2 architecture. It is found that both the throughput and area*time metrics of the implementaion can be dynamically varied. In an algorithm where the context switching is driven by the data input stream a form of hardware thrashing is possible. The results of the paper also show how the restriction of throughput of a CPU due to excessive interupt frequency can be removed by transferreing computing load to reconfigurable hardware.

1 Introduction

Context switching of hardware (also called run time reconfiguration [Luk et al, 1996] dynamic reconfiguration [Singh et al,1996] and swappable logic [Brebner,1997]) can be viewed as taking its inspiration from the paging mechanism in operating systems but extending the concept to hardware logic circuits. There are some important distinctions and qualifications that need to be made in extending the paging concept to hardware. These will be raised in the next section of this paper. However the reasons for introducing context switching with hardware units are most likely similar to the motivations for having paging. These are to allow expensive and therefore limited reconfigurable hardware resources (mostly routing resources) to be fully utilised for productive work, to allow logic circuits larger than available reconfigurable FPGA capacity to be implemented and to a make virtual context for the input and output connections to the circuit. In a custom computer the reconfigurable logic can be viewed as completely replacing the conventional CPU. However it seems unlikely that the mass produced processor will be made obsolete by reconfigurable logic. Rather the reconfigurable logic in some way will extend the resources available to a conventional processor to allow it to perform some types of operations faster just as hardware is now used today as a coprocessor for floating

point. A more telling reason to use hardware is to avoid software operations that severely impair performance of a standard von Neuman processors by placing them in hardware. The application discussed in this paper is targeted at architectures that use the reconfigurable logic as an adjunct to a standard processor and in which the hardware is primarily used to avoid operations which are inherently inefficient when executed on the standard processor.

It is not clear at present just which applications will benefit from the custom computing approach. The research discussed in this paper is intended to expose issues that might impinge on the right choice of applications by exploring the design space of custom computing using a signal processing application called peak detection. The study began by exploring various static choices as to which parts of the algorithm are best placed in hardware and which in software. This has given us some idea of the tradeoffs involved in the implementation of the algorithm including a range of actual measurements of the throughput and area (code size) associated with different static hardware/software partitions. The next step in the research was to introduce a context switched version of the implementation and explore the impact that context switching overhead might have on the speed and area of the design as compared with a static hardware software partition. We also have tried to evaluate how the performance of the partitions used in a context switching environment might be different to those associated with the static software/hardware case. Also explored briefly are the utility of the additional design options provided by hardware context switching in an application domain. In particular the application has highlighted the use of hardware context switching to provide for a graceful degradation of performance of the application when the processing requirements exceed the available capacity of the software based implementation.

The paper is organised as follows. In the first section we make some general comments on the differences between hardware context switching (which we also call hardware “paging”) and conventional software paging. We then introduce the peak detection algorithm and review some existing results of static hardware software partitions of the algorithm. Following this we examine the opportunities for the use of hardware context switching in the algorithm. The context switching performance of the SPACE2 platform is then evaluated and the likely impact on the algorithm’s performance of context switching overheads are evaluated. Context switching allows a wider range of tradeoffs between hardware and software and these are explore using data obtained from the context switching case. Finally we discuss the general implications that can be drawn for this research for the future design of custom computing hardware.

2 Hardware Context switches and Hardware Paging–How is it different from Software Paging

The fundamental difference between hardware context switching and paging arises from the nature of the objects being switched. In this section we want to point out this leads to some of the major differences between conventional software paging and hardware context switches not all of which seem yet to have been examined in the literature. One major issue about hardware context switching which has already been noted is that hardware resources are two dimensional as compared with RAM pages which are one dimensional. This means that we must decide in a hardware context switch whether we will deal with regular shaped circuits (say with square bounding boxes) or whether we will deal with irregular shapes and face the difficulty of mapping these into irregular spaces on the FPGA. It is interesting to note that the OS community has opted for fixed sized pages to reduce the overhead of a context switch and the possible extra management overhead that arises when memory fragmentation is possible. It seems very unlikely that the custom computing community will be able to afford irregular and different shaped hardware logic blocks when the time cost of a custom place and route calculation which is needed to connect irregular hardware blocks is vastly greater than the cost of managing a fragmented RAM which uses different sized pages.

In the next few sections we examine some of the implications that the differences between software page objects and hardware page objects will have on the design of a custom computer.

2.1 Hardware “pages” can be multi threaded

In paging the code being swapped between RAM and disk usually represents a single thread of control. In the hardware context the logic can be concurrent. This means that the replacement of a hardware “page” must take into account the concurrently operating threads of control in the “page”. For example it may not be possible to replace a hardware “ page” until all the threads of control have completed within it and it may not be necessary to bring into memory a new hardware “page” (i.e. a hardware “page fault”) until both threads feeding the new “page” are ready to supply data to the new “page” (new “page” has a join hardware synchronisation function)

2.2 Hardware “pages” begin computation immediately they are loaded into the FPGA

In software paging placing the code into RAM does not necessarily imply immediate execution whilst hardware will potentially run immediately if it is loaded into the FPGA. Thus a hardware context switch which brings in a new hardware “page” onto the FPGA must be sure that its inputs are stable before it begins its computation. There seem to be two ways to do this. First the input data could be brought in with

the hardware “page” as the contents of a register. In this case the page carries its own context and the transfer of data between hardware “pages” could be done in a similar way as a RAM transfer (maybe using software). The second way to synchronise the hardware context switches with cycles of the other hardware on the FPGA. For example all hardware “pages” might be designed to complete in n clock cycles and then any hardware “page” swaps could be restricted to the nth clock cycle.

2.3 Hardware “pages” do not need an external context.

Paged software is assigned and needs additional states or context (registers, stack pointer) when it begins to execute on the processor whilst the hardware page does not necessarily need this additional context to perform useful computation. This is another possible variation on the case above. Imagine a hardware block that is iterative and is swapped out in the middle of the iteration. If the iteration is supplied with data that is already loaded into registers inside the “page” the iteration can proceed immediately without any external context. The hardware paging interface could poll the independent hardware page for completion and then read the results and swap it out. Of course the detection of completions implies some output synchronisation between the “page” and the page swapping hardware/software.

2.4 Hardware “page” swaps can be implemented in reconfigurable logic

Just as in the management of paging in a normal OS the page tables that control virtual memory can themselves can be stored in virtual memory there seems no reason why the swapping of hardware “pages” should not itself be implemented in reconfigurable logic which is itself partly swapped in and out of the FPGA.

2.5 Is there a case for a “cache like hierarchy” for reconfigurable logic

There is obviously a penalty to pay in the implementation of an FPGA with a capacity for fast reconfiguration. Is there a case for a custom computer with two different types of reconfigurable logic one of small capacity with a fast reconfiguration rate and the other large capacity with a slower rate. It is clearly not necessarily meaningful to propose a “cache like” hierarchy of FPGA resources based just on reconfiguration rate. The reconfigurable logic cache also is likely to be influenced by the inverse relationship between execution speed and reconfiguration speed. Fast reconfiguration may slow down the processing rate of the hardware loaded into the FPGA because of tradeoffs related to interconnect. Thus we can view the “reconfigurable logic cache hierarchy” as the fastest logic and the slowest reconfiguration rate at the top (like registers) whilst the fastest reconfiguration rates and thus the slowest speed at the bottom (like dynamic RAM). We could image a cache management policy involving a sequence of data moves between these levels

maybe having the identical copies of a particular logic function existing on each level even. We could also have a sequence of hardware logic moves between these levels based on the needs and locality of the current application with the most often used logic moved into the fastest FPGA with the slowest reconfiguration rate. Into this already wide choice of options must be placed the existing memory products. Another option is to combine RAM cache with reconfigurable logic. This raises questions such as the following: When is it appropriate to transfer the configuration of the FPGA to static RAM as opposed to dynamic RAM or just directly between a slow reconfiguration rate FPGA and a fast one? Is there a benefit for a RAM cache for the configuration data on the FPGA chip itself to overcome the bottle neck observed and discussed later in this paper when data is transferred between the FPGA and off chip static RAM?

In the next section we report on the initial phases of our research into the issues listed above. The methodology has been to examine these options by extracting performance data from DSP functions implemented as the case studies. In this paper we present one of these studies involving a peak detector function.

3 Peak Detection - Non Context Switched Design

In this section we explain the implementation of a peak detector algorithm in a static fashion - that is without context switching. We first describe how we have developed the software hardware codesign and then discuss the functionality of the components. We make some comments on the relative performance of various partitions of the algorithm into hardware and software.

A peak detector takes as input a stream of samples from an analogue channel. A peak is defined as a value in the stream greater than some threshold. The output of the peak detector is the average number of peaks in a fixed interval of time. The peak detector can be divided into components as shown in Figure 1. The peak detector *component* is a comparator. The average peak count component stores the contents of the counter component when a regular event is issued by the period timer component.

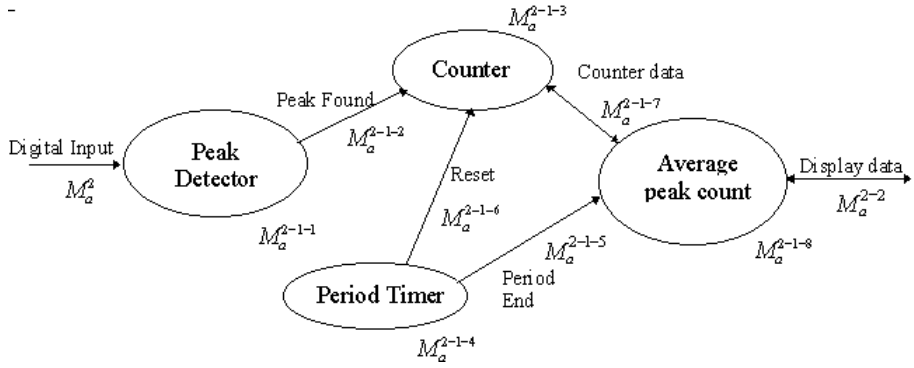

Fig. 1. 'Count the average number of peaks that exceed a preset threshold' HCFSM representation.

The peak detector signal processor application was implemented using a software hardware co-design technique. This is necessary since it is not clear at the start of the design just which parts of the algorithm will end up in hardware ad which in software. In the context switching case the same component may be running in software at one point and in hardware at another. Co-design techniques can be categorised into two broad areas. The first approach is the unified methodology [Themes, 1993],[Ismail, et.al., 1994a],[Ismail, et.al., 1994b] in which is a single language is used for top down design. The second approach is called heterogeneous [Kalavade A, 1994] where different design languages are used for each component.

This peak detector application described here was implemented using a unified methodology called Hierarchical Concurrent Finite State Machines (HCFSM) [Kiefer & Kearney, 1997. The first version of the implementation was targeted at a custom computer designed and built by the authors at the University of South Australia consisting of a XILINX 4000 FPGA and a DSP32c DSP chip. The co-design method consists of decomposing the specification into components and messages. The lowest level of the decomposed peak detector design is shown in Fig 1. All the components except the comparator (peak detector) were mapped into both software or hardware. The comparator was only implemented in hardware to ensure that the DSP interrupt was only activated on the arrival of a peak and not on every sample of the analogue signal. A partition is defined as a particular division of the algorithm between software and hardware. The "cost" of several implemented partitions was evaluated in terms of area and speed. The full results are reported elsewhere [Kiefer & Kearney, 1997] but a summary of the salient points are summarised below.

The overall speed of any partition was found to be significantly affected by the number of messages that were communicated between components across the partition boundary between hardware and software because the message passing was controlled using interrupts. As a result for some partitions the DSP processor spent much of its time servicing these interrupts which degraded the speed of the software

implemented components.

Component Label	Component Name	Area (bytes of configurable logic)
A	Peak detector (comparator)	30
B	Period timer	11
C	Peak Counter	24
D	Averaging Logic	8

Table 1 FPGA area cost of peak detector components

The FPGA resources used by a single channel peak detector are given in Table 1. The performance measured for the implementation in terms of throughput (peaks detected per second) and area (configuration bytes of reconfigurable logic) for three of the partitions studied are given in Table 2. The table also defines for each partition just which components are implemented in software and which in hardware. Table 2 also give a area*time metric for the partitions assuming where the are is just the FPGA resources and does not include the RAM resources consumed on the host by the software implemented components. In the published study these RAM resources are included . They have not been included in this study because the relative cost the small amount of dynamic RAM used on the alpha host for the SPACE2 is considered insignificant in comparison with the cost of FPGA area resources.

Partition Label	Components in hardware (FPGA)	FPGA area (bytes of configurable logic)	Components in software	Throughput of the partition (peaks detected per second)	Area/ Through -put
State 1 Hardware Only	A,B,C and D	71	None	1.92 MHz	36.9
State 2 Hardware /Software	A and B	41	D only	0.81 MHz	50.6
State 3 Mostly Software	A only	30	B,C and D	0.27 MHz	111.1

Table 2 Non context switched performance of partitions

4 Peak Detection–Context Switched Design

In this section we will describe how the peak detector application can be implemented with context switching. We then evaluate the performance of the algorithm under context switching on a typical custom computer.

The three possible partitions that take part in the context switching have been described above . The events initiating the context switch are related to the frequency of peaks occurring in the input stream. If the frequency of the peaks is high the algorithm can only be implemented with all components in hardware. If the frequency of the peaks is low all but the peak detector component (comparator) can

be placed in software. The third partition covers the intermediate case.

To assist in understanding the operation of context switching in the algorithm a state diagram can be defined showing the three partitions as states (nodes) joined by edges representing the events triggering context switches (dependant on the currently measure frequency of peaks). For each transition there is a associated cost of the context switch defined in terms of delay and for each state there is a given performance level in terms of maximum peak detection rate and associated FPGA area required to sustain this rate.. Given a particular input data pattern (which we call a work load) it is possible to derive the points when context switching will take place and hence the performance of the algorithm. We measure the performance in terms of the number of peaks processed per second and the area of FPGA hardware used weighted according to the time a particular partition using this area is loaded. We call the area measure a "rent" area by analogy with a rent paid for the use the FPGA floor space.

4.1 Types of Context switching

The simplest context switch which can be investigated is to swap all the components except the comparator between hardware and software. The criteria used being based on when the peak rate exceeds a threshold. Based on the states defined in Table 2, the fourth row of Table 3 shows this option together with the number of bytes required to be moved from FPGA to RAM for the context switch. The columns of Table 3 will be explained in due course.

Context switch state n to m	Swapped hardware components	Area swapped (bytes of reconfiguarbe logic)	Byte transfer time µs	Byte transfer time for 10 channels µs	Host transfer overhead delay µs	Total context switch time µs
1,2 2,1	C and D	30	3.6	36	10	46
2,3 3,2	B	11	1.32	13.2	10	23.2
1,3 3,1	B,C and D	43	5.16	51.6	10	61.6

Table 3 Context Switching times between states

The other rows of Table 3 shows the same information for the other possible context switches from state 1 to 2 and from state 2 to 3.

4.2 Context switching time

The time required to perform a context switch (as given in Table 3) has been obtained based on data from the SPACE2 [Gunther, 1997] custom computer. SPACE2 uses a array of dynamically reconfigurable Xilinx XC6216 FPGA's with closely coupled RAM connected to host computer via a 33MHz 64bit PCI bus. SPACE2 has local

static RAM directly connected to the XC6216 FPGA's as shown in Figure 2 which can be used for context switching.

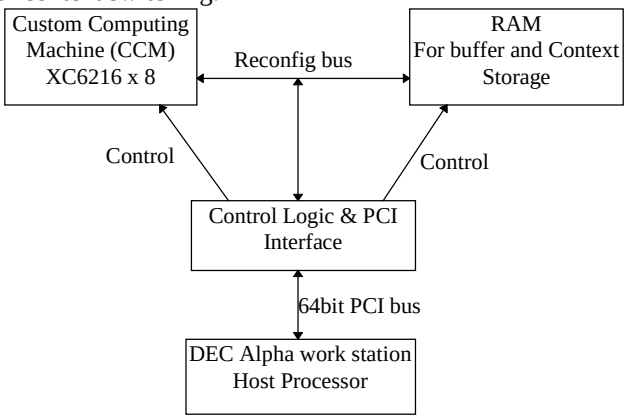

Figure 2: SPACE2 context switching hardware block diagram

SPACE2 also allows FPGA contexts to be stored in the dynamic RAM of the host with context switching via the PCI bus but this possibility is not considered in this paper because it is slower than using the directly connected fast static RAM.

The factors which determine the context switch time are number of reconfigurable logic configuration bytes to be transferred (approximately 120ns per byte) plus a fixed overhead of about 10μs.

Although the peak detector application was originally implemented on a 50MHz DSP it was retargetted to SPACE2 with its 266MHz DEC alpha host. The data in this paper refers to this retargetted design. A single channel of the peak detector occupies a relatively small area of the FPGA and the time required to transfer the hardware for 10 channels of the application is shown also in Table 3. It is assumed in this case that all channels will be transferred at the same time. It can be seen from Table 3 that the smallest context switch is for changing from the partition of state 2 to state 3. It is interesting to note that from Table 2 that a change from 3 to 2 will triple the throughput of the application.

4.3 Performance results

The performance of the application was obtained by applying three different workloads to it. Each work load consisted of a repetitive pattern of peaks with a cyclic variation in the peak frequency. The patterns are summarised in Table 4. Workloads 1. 2 and 3 from Table 4 were simply an alternating cycle of two different peak frequencies which exercise two of the partitions whilst workload 4. was a cycle of three peak frequencies that exercised all three of the hardware software partitions.

Work load label	Round robin cycle transitions	"Rent" Area (bytes)	Throughput formula MHz (ts is the time slice in µs)	Maximum Throughput MHz	"Rent" Area / Maximum Through-put
1.	1,2 2,1 ...	56	2.74/(2+92/ts)	1.37	40.8
2.	2,3 3,2 ...	35.5	1.08/(2+46.4/ts)	0.54	65.7
3.	3,1 1,3 ...	50.5	2.19/(2+123.2/ts)	1.09	46.3
4.	1,2 2,3 3,1 ...	47.1	3.0/(3+130.8/ts)	1.00	47.1

Table 4 Throughput and "rent" area for context switched logic with different input workloads

The "rent" area column of Table 3 is a average of the areas used by the partitions used in the cycle weighted by the time that each partition was active. The throughput of a workload is dependant on length of time each part of a cycle of different peak frequencies is maintained which is the same as the time a partition remains active on the FPGA between context switches. This is analogous to the time a particular job remains executing on a time shared CPU so we have used a variable called a "time slice" (ts) to represent this aspect of the workload. Of course if the input data induces context switching too often we would expect the throughput of the application to be damaged by the overhead of excessive context switching and as we will see this is in fact the case. Included for reference in Table 3 are the maximum throughputs and associated "rent" area*time metric (="rent" area/throughput) that can be obtained for each workload. These figures correspond to the case where the rate of context switching is so low that the overhead associated with it is negligible.

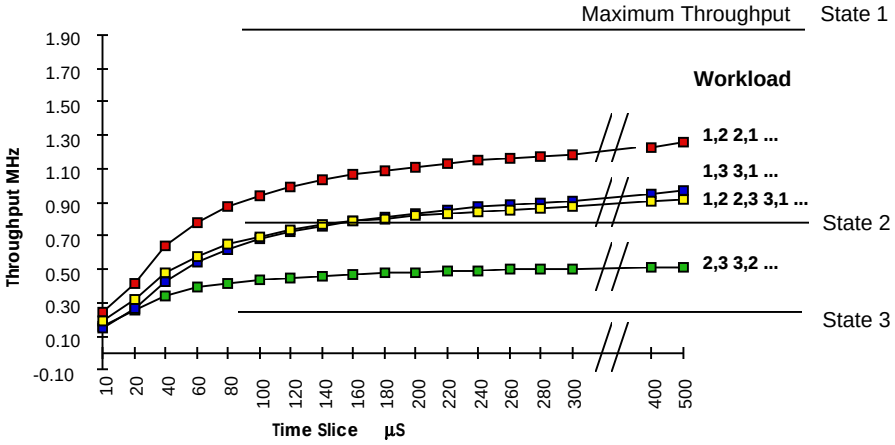

Figure 3 Throughput for context switched logic for different workloads and timeslices.

The throughput of the 10 channel peak detector application is shown in Figure 3 as a function of the size of the timeslice and the type of work load. The results in Figure 3 show the context switching overhead tends to make it unproductive to have too numerous context switches and the large time slice performance of each of the workloads approach the average of the performance over the partitions used. Figure 3 illustrates how context switching with a reasonable time slice can provide a method

of dynamically exchanging processing load between hardware and software depending of the throughput requirements of the moment.

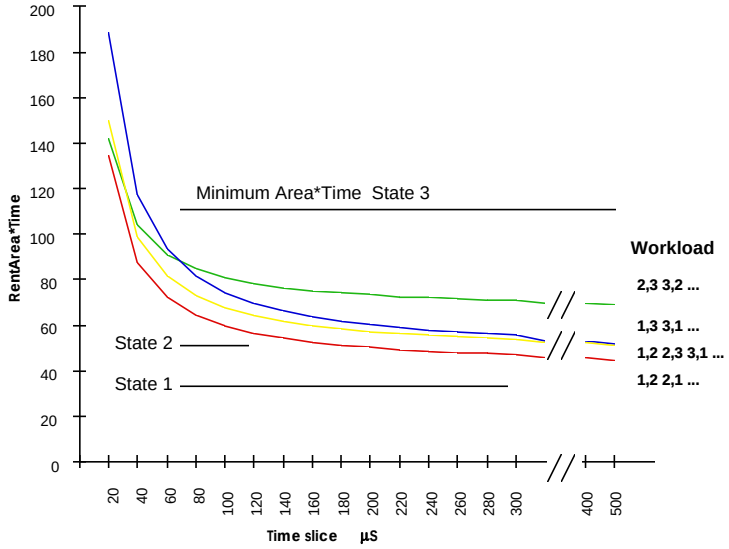

Figure 4 Rent Area*Time for context switched logic for different workloads and timeslices.

Figure 4 shows the “rent” area*time graphs for the various workloads and time slices it shows how the introduction of context switching allows the design to choose to operate the custom computer at an arbitrary area*time point by constraining the partitions that can be used in context switching. By choosing a context switching points for the peak detector algorithm based on peak frequency the design can dynamically vary the tradeoff between speed and area to suit the needs of the moment.

Conclusion

In this paper we have reported on some initial explorations of the possibilities of context switching in a signal processing application. The paper shows that custom computing using run time reconfigurable logic opens up a range of new design options for the implementation of signal processing algorithms as a combination of software and hardware partitions. We have defined some of these options and explored the quantitatively the implication of context switching hardware in a peak detecting application. The results show how context switching of hardware means that the designer can make trade offs between performance and logic area dynamically. The study has shown how there is a need to develop new metrics for assessing the implementations of signal processing algorithms when they are implemented on custom computers because many of the current metrics based on throughput and area time are dynamically selectable in a custom computer.

References

Luk W et al (1996) Modeling and Optimising Run-Time Reconfiguration Systems, Proc. IEEE Symposium of FPGAs for Custon Computing Machines, Napa Valley, California.

Shigh S et al Expessing Dynamic Reconfiguratoin by Partial Evaluation, Proc. IEEE Symposium of FPGAs for Custon Computing Machines, Napa Valley, California.

Brebner G (1997) The Swappable Logic Unit: A Paradigm for Vireual Hardware, Proc. IEEE Symposium of FPGAs for Custon Computing Machines, Napa Valley, California.

Kiefer RH & Kearney D (1997), Performance Comparison of Software / FPGA Hardware Partitions for a DSP Application, MICRO97 Melbourne, Australia Oct 1997

Gunther, Bernard (1997) SPACE2 as a Reconfigurable Stream Processor. 4th Ann. Australasian Conf. on Parallel & Real-Time Systems, 1997.

Themes, D.E, and Adams, J.K. and Schmit, H. (1993) 'A model and methodology for hardware-software CoDesign' IEEE Design & Test of Computers, Vol 10, Iss 3, pp6-15, Sept. 1993.

Ismail, T.B. Abid, M O'Brien, K. & Jerraya, A.(1994a) 'An approach for hardware-software CoDesign', The Fifth International Workshop on Rapid System Prototyping, Grenoble, France Cat. No.94TH0633-8, p. 73-80, IEEE Computer. Soc. Press, Los Alamitos, CA, USA, 1994.

Ismail, T.B. Abid, M & Jerraya, A.(1994b)'COSMOS: A CoDesign Approach for Communicating Systems', Third International Workshop on Hardware/Software CoDesign, Grenoble, France, Cat No 94TH0700-5, p17-24, IEEE Computing Society Press, Los Alamitos, California, USA, 1994

Kalavade A. & Lee, E.A.(1993) 'A hardware-software CoDesign methodology for DSP applications', IEEE Design & Test of Computers, Vol 10, Iss 3, pp16-28, Sept. 1993