Data-path Synthesis of VLIW Video Signal Processors

Zhao Wu and Wayne Wolf
Dept. of Electrical Engineering, Princeton University
Princeton, NJ 08544, U.S.A.
Tel: (609) 258-4261, Fax: (609) 258-3745
Email: {zhaowu, wolf}@ee.princeton.edu

Abstract

This paper describes a methodology for synthesizing the data-path of a Very Long Instruction Word (VLIW) based Video Signal Processor (VSP). Offering both performance and programmability, VSPs are important for their roles in digital video applications, which are omnipresent in today's world. Among many different architectures, VLIW is becoming increasingly popular and widely used due to its efficiency in exploiting high degree of parallelism inherent in multimedia applications. While architectural syntheses of embedded systems have been studied in depth, little literature has addressed similar issues for VLIW-based VSPs. Using an MPEG-2 video encoder as a case study, in this paper we present a combined application of trace-driven simulation and performance estimation in the data-path synthesis of a VLIW VSP. Results show that our estimations are quite precise and helpful, let alone that they are orders of magnitude faster than simulation.

1. Introduction

Just as a wide range of applications and time-to-market pressures have driven audio-rate processing towards programmable DSPs, the need for greater functionality and short product cycles will push the video industry to programmable VSPs. In addition to offering great flexibility and video-rate performance at lower cost than multimedia-enhanced microprocessors (e.g. MAX2 for Hewlett Packard's PA-RISC [5]), programmable VSPs can also motivate the development of new types of multimedia applications by providing an efficient means to design, implement and test algorithms.

Because of the nature of video-rate signals, multimedia applications generate a considerable demand for real-time and computational-intensive processing. E.g., an MPEG-2 encoder requires from a few hundred million to several billion RISC-equivalent operations per frame [6]. On the other hand, VLIW architecture, while keeping hardware simple, combines a high degree of parallelism with the efficiency of statically scheduled instructions. The brevity makes the architecture easy to implement and attractive. The 90's have seen a great popularity of VLIW architecture in a number of commercial products such as Philips's TriMedia [11] and Chromatic Research's Mpact2 [7].

In this paper, we focus on the data-path synthesis of Princeton VSP, which is a VLIW-based microprocessor consisting of several clusters. We use trace-driven analysis to evaluate the performance of application on many different architectures with varying parameters such as register file size, number and type of functional units, et al. Since the trace length of a real application is in the order of 10^9 instructions or even more, trace-driven studies are very time-consuming, hence we cannot afford evaluating all the possible solutions in the design space. Instead, we use some statistical information collected at full-length simulation to estimate the performance without looking at the whole trace every time for each different data-path.

For a number of reasons, we use MPEG-2 encoder as a case study of data-path synthesis for our VSP. First, this is a comprehensive example which represents a category of typical video applications. Second, this application is fairly complicated and challenging for next-generation VSP — no programmable VSPs today can implement it yet. Third, the application is becoming very popular from home video to satellite broadcasting. Finally it is readily available.

The rest of this paper is organized as follows. We go over some related work in section 2, and then introduce the architecture of Princeton VSP in section 3. Section 4 discusses our methodology in detail. Section 5 shows the results. Finally conclusions are drawn in section 6 with future work.

2. Previous work

Micro-architectural synthesis is an overwhelmingly complicated task. An important case study of a VLIW processor based video system was done by Camposano et al. [1]. Using hardware/software co-design techniques, they synthesized a DCT-based video compression system. However, they did not model the VLIW processor precisely. Instead, they only analyzed the ratio of different instructions in a given application; dependencies among these instructions were neglected. Therefore their tool can only generate a lower bound on the resource numbers and an upper bound on the achievable performance.

Fisher and his colleagues at HP Labs practiced a more detailed exploration of VLIW VSPs [4]. They retargeted a VLIW compiler onto different architectures, generated object code from benchmark programs, ran the code trough a simulator to get the execution time, and compared the cycle counts along with area and clock speed obtained from estimation. Although this method can yield precise results, simulating the whole application would take such a long time that only experiments with small pieces of code are practical. Not surprisingly, as Fisher et al. have observed, the target architecture is very sensitive to the programs being executed — an architecture best tailored for one program may perform very poorly (up to six times worse) for another one. As a result, the overall performance of the entire application on the target architecture is still hard to predict, even though the performance of some kernel algorithms is known. Therefore evaluating the complete application is necessary.

In our previous work, using trace-driven architectural exploration, we analyzed the entire trace of several real video applications, including H.263, MPEG-2 and MPEG-4 CODECs, and studied their performance on different parallel architectures [10]. The results show us which architectural tradeoffs enhance the overall performance in the application domain and how we should balance processor resources among registers, functional units and memory units. By comparing the results, we can choose in the multi-dimensional design space the next point to evaluate and then analyze that architecture. This would save us tremendous time as compared to exhaustive search. However, the procedure requires a lot of involvement of human being — the designer has to enter the number and type of each hardware resource, and then decide the next set of architectural parameters based upon previous simulation results. Hence it would very helpful if some tools can automatically narrow the set of candidate architectures.

3. Architectural model

Figure 3.1 depicts the architecture of Princeton VSP. This is a VLIW-based microprocessor in which several clusters are integrated together on a single die to deliver high performance required for video signal processing. Inside each cluster, there are several functional units (ALUs, shifters and/or multipliers), a few memory units (for memory load and store operations), a local register file, and some local memory. Inter-cluster communication is provided by a global crossbar which is able to transfer any register from one cluster to another within a single cycle.

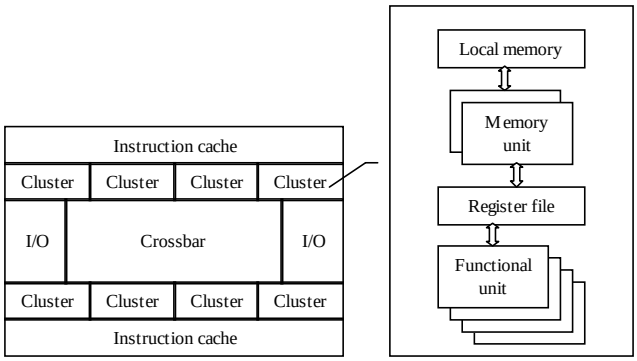

Figure 3.1: Architecture of Princeton VSP

Given a 32-port crossbar, there are a number of configurations. E.g. there could be 8 clusters each having four issue slots, or 16 clusters each having two issue slots, and so on. Depending on the applications being executed, the clusters could be homogenous or heterogeneous. Wolfe et al.'s study of five key VSP kernel algorithms shows that architectures with 16 clusters outperform 8 clusters [8].

In each cycle, our VLIW processor can issue 32 regular instructions (executed by the functional units and memory units) and one branch instruction (executed by the branch unit, which is not shown in Figure 3.1), achieving a high degree of parallelism.

Focusing on the data-path of the VSP, we are interested in finding out the optimal numbers of registers, functional units and memory units for a specific application, namely MPEG-2 encoder. Although knowing these numbers is still far away from the final chip, at least we can determine the basic architecture of the VSP and fix the programming model.

4. Methodology

To synthesize an architecture, we must perform several tasks. The first and most basic problem is to estimate the cost of the architecture; given that we are using library components such as SRAMs and data-paths, we can easily obtain accurate area estimates. We also need to evaluate the performance of the architecture. This step is much more challenging given the large sizes of the traces on which performance estimates are based; much of our work has gone into performance estimation.

In our problem, we limit the size of the design space by only allowing certain resource configurations. This not only makes the chip easy to implement (it is difficult to layout an architecture with arbitrary configuration), but also simplifies compiler design. Starting from the assumption of using 16 clusters, we outline our search space with following parameters:

- Total number of register: 256, 512, 1024, 2048
- Total number of memory units: 8, 16, 24, 32
- Total number of ALUs: 16, 24, 32
- Total number of shifters: 8, 16, 24, 32
- Total number of multipliers: 8, 16, 24, 32

Notice that the numbers listed above should be divided by 16 in order to get the configuration for each cluster. When the resource number is not a multiple of 16, the architecture becomes heterogeneous. E.g. if we are to use 8 shifters, then half of the 16 clusters will not have any, while the other half will have one each.

4.1 Performance and cost

We use total execution time of the application and chip area (not including instruction cache) as the metrics for performance and cost respectively. The total execution time is further defined as the product of cycle count and clock rate. We obtain the cycle count through trace-driven analysis, which will be explained in the next subsection.

Number of registers	Number of ports			
	3	6	9	12
16	0.8957	0.9498	0.9572	0.9620
64	0.9596	0.9648	0.9709	0.9763
256	0.9745	1.0006	1.3903	1.4845

Table 4.1.1: Delay (ns) for 16-bit register files

Number of registers	Number of ports			
	3	6	9	12
16	0.1098	0.2201	0.3975	0.5851
64	0.3782	0.7362	1.2615	1.8405
256	1.3913	2.6566	4.3892	6.3818

Table 4.1.2: Area (mm^2) for 16-bit register files

The other two factors, area and clock rate, are based on Dutta et al.'s work [3]. Using a 0.25μm CMOS technology, Dutta has designed parameterizable versions of key modules. Furthermore, he simulated all the components with AT&T's ADVICE circuit simulator and collected all the timing information. As an example, Table 4.1.1 and 4.1.2 show, respectively, the delay (cycle time) and area of a 16-bit multi-port local register file. Other details of Dutta's work can be found in his thesis [2].

4.2 Trace-driven analysis

Since the instruction set of our VLIW VSP has not yet been fully defined, it is very difficult to evaluate the performance of an architecture. To deal with this problem, we follow the same idea Fisher et al. practiced, i.e. to try to match the application being analyzed with the structure and size of the candidate architecture, rather than specific opcodes [4]. However, unlike Fisher et al., we do not have a sophisticated retargetable VLIW compiler, and thus we have to devise some other method.

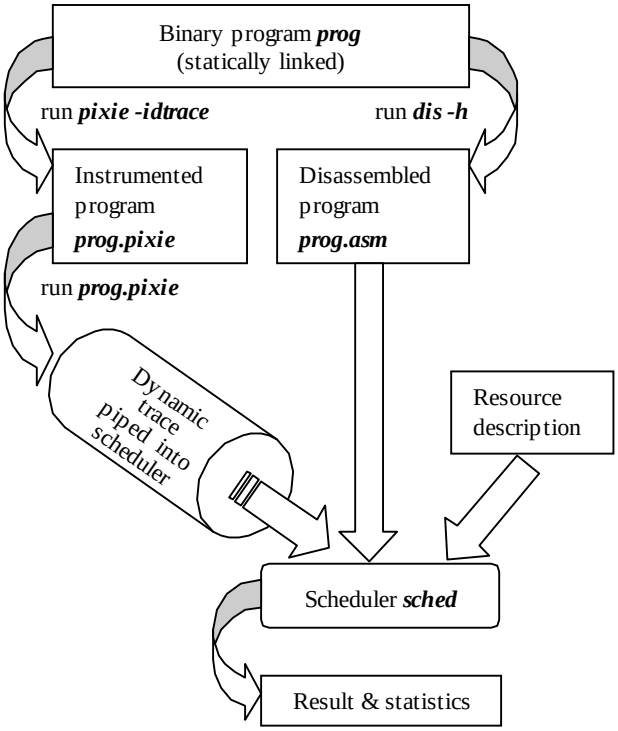

Figure 4.2.1: Flow chart of trace-driven analysis

Instead of doing static instruction scheduling at compilation time, which requires a highly optimized compiler, we schedule instructions dynamically at simulation time. Figure 4.2.1 illustrates the procedure of our experiments on SGI workstations. In addition to the dynamic trace, our scheduler also takes in the disassembled program (for dependency analysis) and a resource description file (which specifies the availability of all the hardware resources and their parameters). Since in our approach the application is instrumented and actually run on a processor rather than being interpreted by a program, the speed is orders of magnitude faster than a software simulator. Other highlights of our scheduler include moderate storage requirement, linearity in terms of input trace length, and using an extremely large search window (up to one billion instructions) to exploit

parallelism in a wide range from basic-block level to functional and loop level. Further information about the scheduling algorithm can be found in out previous work [9][10].

As for the precision of our methodology, of course there are always certain tradeoffs between accuracy and CPU time in performance analysis. Our experience from hand-coding five kernel algorithms in an MPEG-2 encoder [8] shows that the scheduler produces comparable results.

4.3 Performance estimation

Although our scheduling algorithm is quite efficient, it still takes a long time (5-10 hours) to simulate a whole application, just because the length of the dynamic trace is too long. Therefore we hope that we could get the cycle count without looking at the trace, but how?

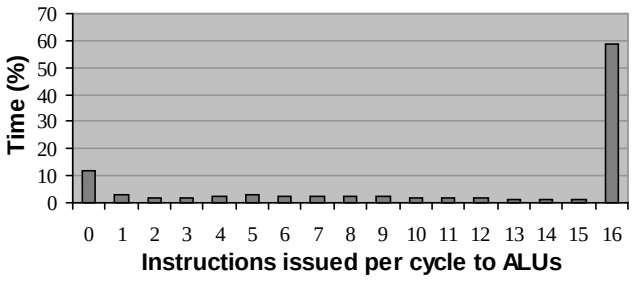

Figure 4.3.1: Histogram of ALU on (1024, 24, 16, 16, 16)

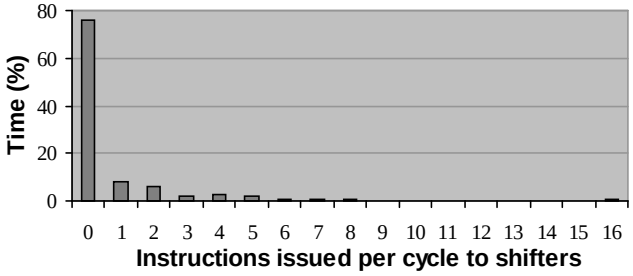

Figure 4.3.2: Histogram of shifter on (1024, 16, 32, 16, 24)

As pointed out in our early studies of the VSP architecture, by analyzing the statistical information accumulated at simulation, we can identify the bottleneck and figure out the most efficient way to use the resources [10]. E.g. Figure 4.3.1 shows the distribution of ALU instruction issue rate on a specific architecture. The quintuple (1024, 24, 16, 16, 16) denotes the architecture with 1024 registers, 24 memory units, 16 ALUs, 16 shifters, and 16 multipliers. Since over 50% of the total execution time a large number of instructions can be issued to ALUs, we would guess that increasing the number of ALUs is very likely to shorten the execution time. On the other hand, decreasing the number of ALUs would probably degrade the performance a lot. Similarly Figure 4.3.2 tells us that halving or doubling the number of shifters is not likely to affect the total execution time remarkably.

Obviously it is hard to crank out a precise cycle count just from estimation, but it is possible to get an upper bound and a lower bound by redistributing the bars in the histogram. E.g. if the number of ALUs is reduced from 24 to 16, then the bars corresponding to 17-24 ALU instructions (per cycle) have to be broken down (Figure 4.3.3). Remember that each cycle counted by the n^{th} bar indicates that n instructions can be issued simultaneously in that cycle, hence a 20-instruction cycle will be remapped to one 16-instruction cycle followed by one 4-instruction cycle, and so on. Notice that in the above case, some other (later issued) ALU instructions may be scheduled into the same cycle as the 4-instruction cycle, provided that all the dependencies have been cleared. Therefore the new cycle count from re-mapping is pessimistic.

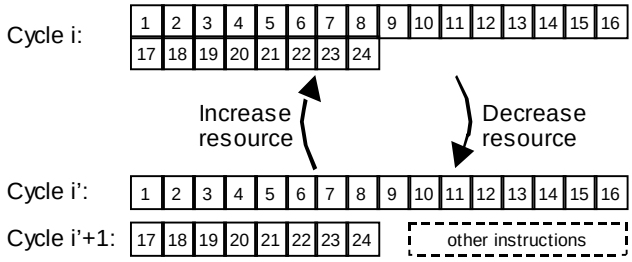

Figure 4.3.3: Increase and decrease resource

On the other hand, when we increase the number of units of a resource, we can redistribute the histogram the other way round. Nonetheless, we don't know which low-issue-rate cycles can be chosen to form a high-issue-rate cycle. Of course we can extract and save that information at simulation, but that would require a gigantic data structure with as many entries as the total cycle count. Moreover, manipulating such a big data structure would not be much faster than a complete simulation. Our solution to this problem is very simple: we just subtract from the total cycle count the number of cycles corresponding to the rightmost bar in the histogram. E.g. assuming that we are given 8 more ALUs to our already-simulated 16-ALU architecture, then the new cycle count would be cut down by the number of cycles corresponding to the 16^{th} bar. The reason can be explained as follows (refer to Figure 4.3.3). Imagine the situation on the old architecture when more than 16 instructions can be issued on the enlarged architecture. Since the scheduling algorithm is greedy, the old architecture must be able to

issue 16 instructions in that cycle, after which the rest instructions are issued in the following cycle. Not counting the 16-instruction cycle means to issue all the instructions in the following cycle. Apparently this simple estimation yields an optimistic cycle reduction for two reasons. First is that the following cycle may have more than 8 instructions (because the scheduler can squeeze in some other independent instructions), making it impossible to issue more than 24 instructions (as we only have 24 ALUs). The other reason is that a 16-instruction cycle cannot always be merged with another cycle — sometimes the parallelism is exactly 16.

So far we have only changed one resource parameter and considered the impact on the resource itself. However, when we alter the histogram of one resource, the others' could be changed as well due to correlation between different instruction types. To make our bounds tighter, we have to use another piece of information — the dependency array. This is a 3-D array in which element $dep[res_1][res_2][n]$ specifies the number of cycles when at least one res_1–instruction depends on n res_2–instructions and thus has to be issued later. By adding a few lines of code to the dependency analyzer in our scheduler, this 3-D array can be easily obtained as a byproduct of simulation.

When the number of one resource is reduced, the parallelism of other resources (which have dependency on the reduced resource) is also diminished, which can be reflected by the dependency array. E.g., when the number of ALUs is reduced from 32 to 24, in the worst case ($\sum_{24<i\leq 32} dep[multiplier][ALU][i]$) extra cycles will be introduced to multipliers. Similarly when the number of ALUs is increased from 24 to 32, same amount of stall cycles (indicated by the 0^{th} bar in the histogram) can be eliminated for multipliers in the best case.

As a summary, when we decrease the number of one resource, the total cycle count is going to increase, and we can get an upper bound. On the other hand, when we increase the number of one resource, we will get a lower bound on cycle count. However, since the cycle count adjustment is done individually for each resource, we might end up with four different upper/lower bounds for memory unit, ALU, shifter and multiplier respectively. In an extreme case, the real cycle count increase (or decrease) could be the sum of the increase (or decrease) of all the four resource types. Since such a bound is too loose and the situation is too odd to happen, we use the average of the four numbers as a semi-bound. The results from simulation show that our semi-bounds are always true bounds, no matter whether they are upper or lower bounds.

All the discussions we have made previously are based on a very important fact, i.e. the number of registers does not change. We find that it is very difficult to come up with some quantitative estimation about the impact of register numbers. Therefore we have to do the same experiments four times, each for a different register file size.

4.4 Algorithm

The main flow of the algorithm is outlined in Figure 4.4.1 below. We use two sets, *solutions* and *database*, to keep respectively all the promising solutions and results either simulated or estimated.

```
foreach reg ∈ {256, 512, 1024, 2048} {
  solutions = { eval(reg, 8, 16, 8, 8) };       // min arch
  database = { eval(reg, 32, 32, 32, 32) }; // max arch
  foreach arch ∈ (design space) {
    if (area(arch) < budget) {
      arch1 = least_smaller(arch);
      arch2 = least_bigger(arch);
      max_time = estimate(arch) based on arch2;
      min_time = estimate(arch) based on arch1;
      good = 1;
      foreach sol ∈ (solutions) {
        if ( area(sol) > area(arch) &&
             time(sol) > max_time ) {    // arch is better
          solutions = solutions \ {sol};
          database = database ∪ {sol};
          break;
        } else if ( arch(sol) < area(arch) &&
             time(sol) < min_time ) {     // sol is better
          good = 0;
        }                                  // else hard to judge
      }
      if (good)
        solutions = solutions ∪ {arch};
      else
        database = database ∪ {arch};
    }
  }
}
```

Figure 4.4.1: Synthesis algorithm

Depending on whether there is an increase or decrease of a resource, we can only get one bound at a time. So our synthesis algorithm starts from the minimum architecture (i.e. *reg* registers, 8 memory units, 16 ALUs, 8 shifters and 8 multipliers) and the maximum architecture, and then tries to approach the target data-path from both directions. While *least_smaller(arch)* returns the closest architecture (in *solutions* ∪ *database*) with less number of resources, *least_bigger(arch)* returns the closest one with more resources.

5. Results

Table 5.1 shows the solution pool with an area budget of 180 mm^2 (only including data-path and crossbar). The speedup is defined as the total execution time (cycle time × cycle count) of the application divided by the total execution time on the baseline architecture (256, 8, 16, 8, 8) (shown in the first line of Table 5.1). The simulated speedups are obtained from trace-driven analysis, while the estimated ones are calculated according to our estimation method. Notice that due to floating-point rounding, some numbers in the table appear the same but they are actually different. From the results we can see that our method does a good job.

Data-path architecture					Cost	Performance	
# of regs	# of mem	# of ALU	# of shfts	# of muls	Area (mm^2)	Simult'd speedup	Estmt'd speedup
256	8	16	8	8	149.6	1.00	1.00
256	16	16	8	8	151.2	1.03	1.08
512	8	16	8	8	152.6	1.37	1.37
512	16	16	8	8	154.2	1.48	1.50
512	16	24	8	8	157.4	1.54	1.52
1024	8	16	8	8	158.2	1.54	1.54
1024	16	16	8	8	159.8	1.82	1.92
1024	16	24	8	8	163.0	2.07	2.12
1024	16	32	8	8	166.2	2.12	2.19
1024	16	32	16	8	170.2	2.14	2.26
2048	16	24	8	8	175.0	2.30	2.35
2048	16	32	8	8	178.2	2.39	2.45

Table 5.1: Synthesis results

Being afraid that our synthesis algorithm may miss some solutions due to inaccuracy, we randomly picked out 100 architectures from the design space (which contains 768 points in total) and simulated the MPEG-2 encoder trace for these architectures. Fortunately our test passed — all the 100 solutions were discarded because they implied bigger area yet lower speedup than at least one of the winners in Table 5.1.

6. Conclusions and future work

In this paper, we use an MPEG-2 encoder as a case study of the data-path synthesis for a VLIW-based VSP. Our contributions include using a trace-driven scheduling method to evaluate the architecture and a quantitative estimation of the impact on cycle count when varying resource numbers. We apply the synthesis algorithm in our architectural exploration. Results show that our estimations are precise enough to be used when evaluating different architectures.

To simplify the problem, we limit the choice of some design parameters so as to reduce the search space. In general this may not be persuasive. Hence our future work will combine some search techniques (e.g. simulated annealing, genetic algorithm, et al.) to explore the space more smartly and efficiently.

Acknowledgements

This work was supported by the National Science Foundation under grant number MIP-9408462 and the New Jersey Center for Multimedia Research.

References

[1] R. Camposano and J. Wilberg, "Embedded system design", *Design Automation for Embedded Systems*, 1 (1-2), pp. 5-50, Kluwer Academic Publishers, Jan. 1996.

[2] S. Dutta, "VLSI issues and architectural tradeoffs in advanced video signal processors", *Ph.D. Thesis*, Princeton Univ., Nov. 1996.

[3] S. Dutta, A. Wolfe, and W. Wolf, "Design issue for a very-long-instruction-word VLSI video signal processor", *VLSI Signal Processing IX*, pp. 95-104, Oct. 1996.

[4] J. Fisher, P. Faraboschi, and G. Desoli, "Custom-fit processors: letting applications define architectures", *IEEE/ACM Proc. on 29th Int'l. Symp. on Microarchitecture*, pp. 324-335, Dec. 1996.

[5] R. Lee and J. Huck, "64-bit and multimedia extensions in the PA-RISC 2.0 architecture", *IEEE Proc. on Compcon*, pp. 25-28, Feb. 1996.

[6] H. Liao and A. Wolfe, "Available parallelism in video applications", *IEEE/ACM Proc. on 30th Int'l. Symp. on Microarchitecture*, pp. 321-329, Dec. 1997.

[7] S. Purcell, "Mpact2 media processor, balanced 2X performance", *Proc. SPIE Int'l. Symp. on Multimedia Hardware Architectures*, pp. 102-108, Feb. 1997.

[8] A. Wolfe, J. Fritts, S. Dutta, and E. Fernandes, "Datapath design for a VLIW video signal processor", *Proc. 3rd Int'l. Symp. on High-Performance Computer Architecture*, pp. 24-35, Feb. 1997.

[9] Z. Wu and W. Wolf, "Parallelism analysis of memory system in single-chip VLIW video signal processors", *Proc. SPIE Int'l. Symp. on Multimedia Hardware Architectures*, Jan. 1998.

[10] Z. Wu and W. Wolf, "Trace-driven studies of VLIW video signal processors", *Proc. 10th ACM Symp. on Parallel Algorithms and Architectures*, pp. 289-297, June 1998.

[11] http://www.trimedia.philips.com/