Datapath Design for a VLIW Video Signal Processor

Andrew Wolfe, Jason Fritts, Santanu Dutta, and Edil S. T. Fernandes
Department of Electrical Engineering
Princeton University

Abstract

This paper represents a design study of the datapath for a very long instruction word (VLIW) video signal processor (VSP). VLIW architectures provide high parallelism and excellent high-level language programmability, but require careful attention to VLSI and compiler design. Flexible, high-bandwidth interconnect, high-connectivity register files, and fast cycle times are required to achieve real-time video signal processing. Parameterizable versions of key modules have been designed in a .25µ process, allowing us to explore tradeoffs in the VLIW VSP design space. The designs target 33 operations per cycle at clock rates exceeding 600Mhz. Various VLIW code scheduling techniques have been applied to 6 VSP kernels and evaluated on 7 different candidate datapath designs. The results of these simulations are used to indicate which architectural tradeoffs enhance overall performance in this application domain.

1. Introduction

Digital video is one of today's fastest growing information sources. Dedicated video signal processing chips are available now as CODECs and other specialized video functions. A wider range of applications and shorter product cycles have driven audio-rate processing increasingly to programmable DSPs and similarly new applications for video, along with increased development cost and time-to-market pressures, will push system designers to programmable video signal processors (VSPs). Since their architectures are tuned for multimedia applications, VSPs will offer video-rate performance at significantly lower cost than workstation microprocessor architectures.

Many interesting alternatives exist for programmable VSP architectures, but it is impractical to try to definitively determine the best architectural paradigm until the unique research and development issues for each class of architectures have been resolved. Our contribution to this process is an exploration of design issues for very-long-instruction-word (VLIW) VSPs. VLIW combines a high degree of parallelism with the efficiency of statically-scheduled instructions. Sophisticated VLIW compiler technology will allow users to develop real-time signal processing applications in high-level languages. A regular architecture will permit 32 or more operations per cycle while maintaining clock speeds in excess of 600 MHz.

The most accurate method for quantitative evaluation of architectural tradeoffs involves circuit-level timing simulation of full processor layout and cycle-level simulation of full applications based on optimized, compiled code. Unfortunately, the development of this detailed a simulation model is extremely expensive and can only be accomplished for a tiny subset of the design space. In an emerging application domain such as video signal processing, there is not even a body of existing processor designs, tools, and benchmarks to use as a starting point for design modifications. Early exploration methods are required both for technology-driven design parameters such as circuit performance and for instruction-level behavior. Our methodology for this initial investigation involves several steps:

1. After selecting the basic architectural paradigm, we performed detailed, parameterizable, transistor-level design of key modules including high-bandwidth interconnect, high-connectivity register files, and high-speed local memories. This detailed design is required in order to make rational design decisions using an aggressive implementation technology and an aggressive architecture.
2. Area and performance data from these simulations define a unique design space for this processor. Within this design space, candidate architectures are constructed based on the module cost and performance.
3. Key VSP kernels are hand scheduled onto each architecture. This provides early performance estimates and allows programmers to evaluate compilation strategies and designers to identify architectural bottlenecks.

The results of these experiments refine the design space and guide the implementation of the compiler. Eventually, a limited number of candidate designs must still be evaluated with real applications, a prototype compiler, and real application data.

Only recently has it been possible to implement a wide VLIW architecture on a single chip, thus many architectural structures are feasible for the first time [5]. In order to explore the new tradeoffs, we have designed and simulated several key components for a VLIW VSP in .25µ CMOS technology. These structures include large register files, crossbar interconnect networks, and high-speed local memories. These designs form a parameterizable megacell library for the construction of experimental VLIW VSP chips. The components were simulated using AT&T's ADVICE circuit simulator. The delay characteristics and area requirements of these components were jointly analyzed to determine the architectural balance points of a single-chip VLIW machine.

This paper shows that .25µ technology is sufficient to build a powerful, programmable VLIW VSP. In particular, our

This research was supported in part by NSF under MIPS-9408642

experiments show that interconnect bandwidth is not a problem in a first-generation VLIW VSP—the interconnection network occupies only a few percent of the chip area. However, register-file capacity is a significant problem. We construct a working architectural model based on the data from these component designs and analyze the computational capabilities on some key video compression kernels. We conclude by showing that a .25µ VSP can provide the communication, computation, and on-chip memory required to execute significant real-time video applications.

2. Why VLIW?

A great deal of research, as well as a growing amount of industrial implementation effort, has studied parallel architectures for video signal processing; however, few have studied VLIW architectures for VSP. Other approaches to VLSI video signal processing include array processors [11], pipelined architectures [13], and ASICs [14]. Unfortunately, none of these approaches are particularly amenable to high-level language (HLL) programming thus limiting them to a few mass-market applications. VLIWs are primarily designed for HLL programming and thus can serve a wider range of applications.

A VLIW machine exposes functional-unit parallelism directly in the instruction. Traditionally, a VLIW machine consists of a set of functional units connected to a central global register file [4]. A VLIW instruction is horizontally microcoded—the instruction independently specifies the operation performed on each functional unit during that cycle. VLIW architectures use compiler-based dependency analysis and static scheduling to enable high-performance parallel computations on a simple datapath implementation. Run-time arbitration for resources is never allowed, which permits both a very fast cycle time and a great deal of parallelism. In general-purpose VLIW designs, limited application parallelism prevents full utilization of a very large number of functional units, but in signal processing there is abundant parallelism in most applications. The challenge in adapting VLIW to video is to provide a very large number of functional units with enough interconnect resources to permit efficient compiler-based instruction scheduling while maintaining a very fast clock rate.

Previous VLIW architectures have been multi-chip designs because they could not implement sufficient resources in single-chip designs. Modern fabrication technologies make it feasible to implement single-chip VLIWs with significant amounts of parallelism; however, this requires a different approach to storage and interconnect design. Traditional VLIW architectures use a central register file to provide both operand storage and low-latency interconnect. This is ideal architecturally, but extremely difficult to implement. Two attempts to build 16-ported register files [7], [3], produced large, slow circuits. The 96-ported register file required to support a 32-issue VLIW is impractical even in .25µ technology.

Due to the difficulties of building register files with extremely large port counts, we chose an alternative architecture, similar to the one used in [6]. Clusters of functional units, each with a local register file, are connected by a global crossbar switch. Instructions are supplied to the machine through a distributed instruction cache. Local data memory is provided within each cluster. 16-bit integers are the only supported native data type. To maintain a consistent programming model, all clusters are identical. The number of clusters provided in the machine, the number of arithmetic and memory units per cluster, the number of registers per cluster, the number of ports per register file, the amount of local data memory per cluster, and the number of global crossbar ports per cluster are all architectural parameters to be determined by the results of the VLSI simulations and representative application analysis. Fig. 1 shows a floorplan of our VLIW VSP. Clusters of functional units surround the crossbar to form the datapath. An example cluster containing 7 functional units sharing 4 issue slots is illustrated.

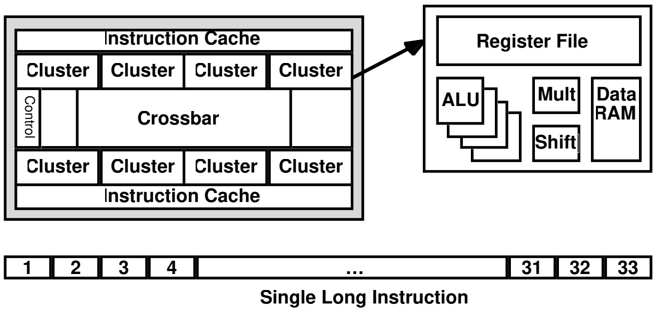

Fig. 1: Cluster-based VLIW architecture.

The code scheduling paradigm for VSP applications on this architecture tends to be quite different than on most research VLIW machines. The applications tend to have abundant parallelism but each cluster has extremely limited local resources. The number of registers is relatively small as compared to the operand bandwidth; the local data RAM is limited; and the local instruction cache is small. At the high clock rates involved, exceeding the capacity of any of these units can greatly reduce performance. Therefore, the compiler's primary job is effectively allocating limited resources rather than extracting maximum parallelism.

3. Design Methodology

We evaluated the VLSI feasibility of our architecture using the design methodologies previously developed for memory systems [2] and datapaths [12]. We first designed the unique components of the microarchitecture—global interconnect, register files and local memory—and measured their area as well as their speed. Next, we evaluated the application performance requirements by manually analyzing 6 different VSP algorithms. Using the performance of the components, along with the application characteristics, we then determined how to allocate chip area among the various subsystems of the microarchitecture. We also analyzed the power consumption and clock distribution for this chip to ensure that both were feasible. Though we do not have room for details, it was concluded that the chip's power consumption, although in the 50 W range, was low enough to be feasible and that the clock could be distributed at the required frequency.

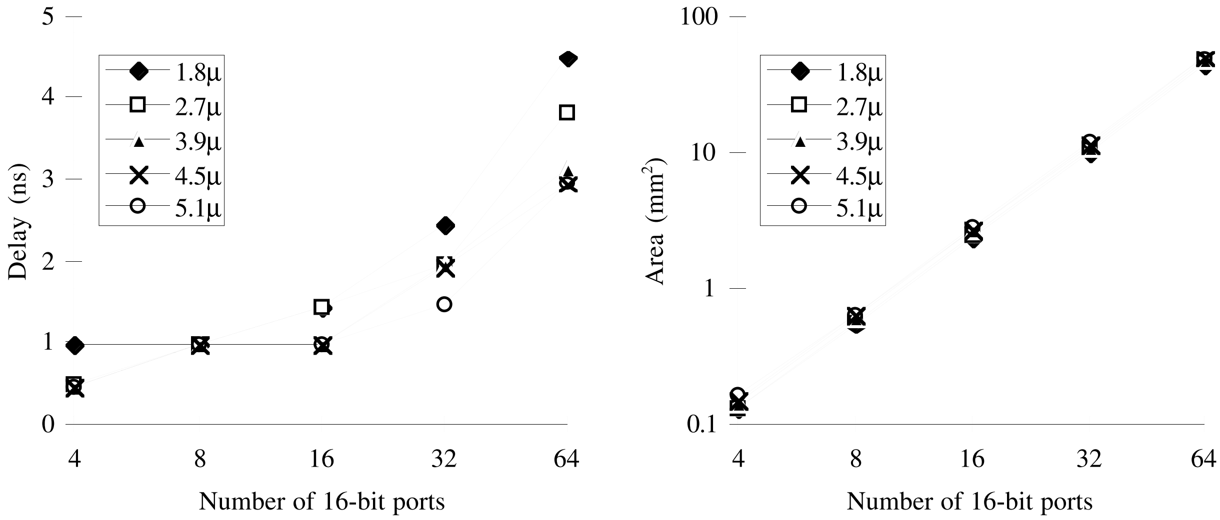

Fig. 2: Delay and Area for 16-bit Crossbar Switches.

3.1. Microarchitecture Component Analysis

As a first step to selecting design parameters for full architectural simulation, we chose to design, layout, and simulate the critical components for a VLIW VSP. We have used an experimental .25µ process for all designs. Circuit simulations were performed at 3.0V and nominal temperatures using AT&T's ADVICE simulator. We use only 2 levels of metal for module designs, reserving the upper layers for inter-module signals and power. The analysis provides area and cycle-time information that defines the reasonable architectural design space. This section describes the designs and characteristics of the major subsystems: global interconnect, register file, local memory, and ALUs. We present only a few of the major results of our layouts and simulations here, due to space limitations. A complete description of these designs and simulations is available in [1] and [15].

3.1.1. Global Interconnect

Since a single global register file is not a feasible solution for both operand storage and global interconnect, a simpler structure must be implemented. An earlier analysis [2], [10] has shown that within the range of parameters that are interesting for a single-chip VLIW architecture the area and speed required for a full crossbar is acceptable. A specialized routing scheme has been developed [10] that allows crossbar inputs and outputs to be routed into the crossbar switch from both sides.

Fig. 2 shows the delay and area of our preferred design with various driver sizes. Cycle times under 1ns can be supported with up to 16 ports, but drop off quickly to 1.5ns at 32 ports and 3ns at 64 ports. The area requirements for the crossbar are relatively insensitive to transistor size within the range of interest. The crossbars up to 32 ports require very little area for a key central architectural structure. In fact, they may be so small that additional routing is required to connect to the functional unit clusters. If so, the additional delay would need to be added into the crossbar delay, a functional unit output stage, or an additional pipeline stage. Considerable detail on the various tradeoffs in crossbar design appear in [10].

3.1.2. Local Register File

Local multi-ported register files are required in each cluster to supply the arithmetic units with operands. Register files with many ports and many registers are always preferable from a programming and compilation perspective, yet they must not be allowed to reduce performance by limiting clock speed or consuming too much space. VLIW architectures are typically designed to supply two operands and write one result back to the register file for each operation each cycle. This requires 3 ports per operation. We have therefore designed register files to support 1, 2, 3, or 4 operations per cycle. As is shown in Fig. 3, the register-file delay is only slightly dependent on the number of ports; however, the area requirements increase substantially both as a function of the number of ports and the number of registers. The optimal design point is a complex issue. As more ports are added to each register file, the number of issue slots that can be supported in that cluster increases, reducing the number of clusters required. Furthermore, the utilization of each issue slot is typically increased which may compensate for the lower register density of the highly multi-ported register files. Unfortunately, the number of registers required in each register file also increases with more issue slots, which increases the register file's cycle time. The data indicate that many register file sizes are reasonable depending on the overall system area and delay tradeoffs.

3.1.3. Local Memory

Like the register file, local memory must be available in each cluster to store video data since global memory cannot provide adequate bandwidth. An earlier analysis [2] of a different datapath showed that multi-ported memories could often eliminate enough access conflicts to reduce execution time despite increases in cycle time. Based on that result, we have designed a scaleable multi-ported memory. The performance of the minimum-size version of this design is shown in Fig. 4.

Note that the minimum-size cell transistor must increase as the number of ports increases, thus performance degrades slightly less than would be expected as the number of ports grows and area increases somewhat more than would be expected.

The basic design has been optimized for high performance with many ports and thus has rather low density. Although the performance of the multi-ported designs is excellent, the area penalty may be too severe for a VLIW design. Even the minimum-size design only allows about 400 bytes of 4-ported memory per mm^2.

Given that there are already two multi-ported interconnect structures on chip, the register file and the crossbar network, the additional performance of multi-ported memory will not compensate for the extra off-chip traffic caused by insufficient on-chip memory. The common practice of double-buffering data memory in signal processing further reduces available storage. In order to provide a high-density alternative, we specially designed 1- and 2-ported memories. Our improved design has a density of over 2600 bytes/mm^2 of single-ported memory or over 2200 bytes/mm^2 of two-ported memory.

3.1.4. Arithmetic Units

The arithmetic units for the VLIW VSP are essentially the same as those for other types of signal processing chips, with the provision that only small integer operations are required. This technology is well developed and we do not believe it will be a critical path; thus we have chosen to base design decisions on published results rather than custom designs. A 32-bit ALU in .25µ CMOS at 1.5ns has been described elsewhere [9]. That design required only 0.6mm^2. A full 54-bit multiplier requiring 12.8 mm^2 in .25µ CMOS [8] was measured to run at 4.4ns. An 8-bit multiplier should run much faster and require under 1 mm^2. A 16-bit multiplier should require under 3 mm^2.

3.2. Architectural Models

Based on the data from these module designs, we selected parameters for several working architectures. These have been used to test application kernels by measuring programmability and performance. Given the goals of this design, we initially chose to emphasize programmability and high functional unit utilization whenever possible. This indicates a design with relatively high connectivity. The key is to select parameters

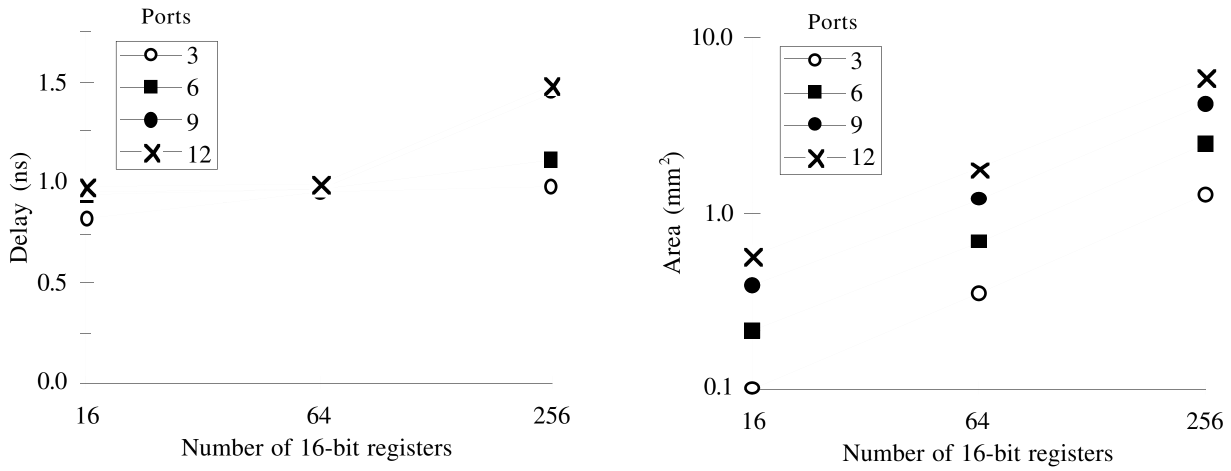

Fig. 3: Delay and Area for 16-bit multiported local register files.

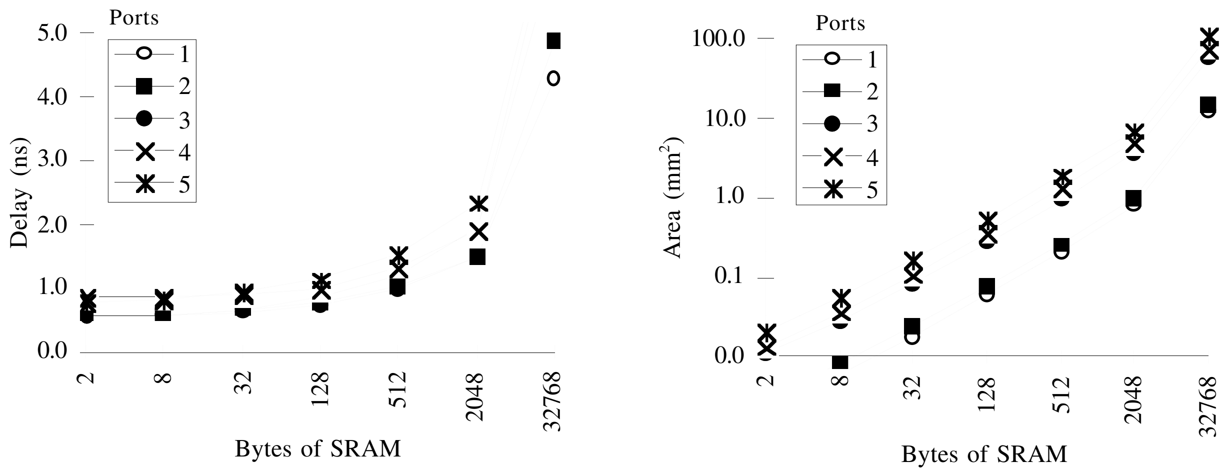

Fig. 4: Delay and Area for multiported high-speed SRAM.

that meet a target cycle time while allocating area wisely. For this initial model, we constructed a datapath with 8 clusters of functional units, each capable of issuing 4 operations per cycle for a total of 32 operations per cycle. In order to maximize connectivity, we provide a single 12-ported register file for each cluster and connect each issue slot to the global network using a full 32x32 central crossbar switch. Given these decisions and our simulation data, a target clock rate of 650MHz was selected for this configuration. Up to 256 registers can be included per cluster and still achieve this target clock rate, but our preliminary application studies indicate that 128 registers/cluster are adequate. Although a cluster is limited to 4 operations per cycle, it should have more than 4 functional units to keep utilization high. We propose 4 ALUs, one multiplier, one shifter, and one load/store unit configured so that each set of 3 register-file ports supports one ALU and up to one alternate function. All operations use 16-bit operands and execute in one cycle except for the multiplier which can only perform 8x8 multiplication. A four stage pipeline (instruction fetch, operand fetch, execute, write-back) is used in this initial configuration. At the target clock rate, this limits the complexity of load and store operations. Only direct or register-indirect addressing can be supported. The cluster is fully bypassed between the 7 functional units, requiring 10-input multiplexers in the operand bypass paths. Memory speed is a critical path for this and all of our configurations. Each cluster contains a single local data memory that is accessed via the load/store unit. The memory is word addressed and double buffered to enable concurrent processing and off-chip I/O.[1] In this configuration we can provide 32KB of single-ported local data RAM based on 16Kx1 modules. With careful design of buffers and pipeline registers, we can meet the target cycle time. Fig. 5 shows an estimate of the area for this datapath which we have labeled I4C8S4. Ten percent additional area has been allowed for local routing between subcomponents. This should be adequate given that there are 2-3 additional metal layers for routing over the subcomponents as well.

Restricted addressing modes can significantly increase the instruction count on some types of common array calculations. More powerful addressing modes such as indexed (register + register) and base-displacement can reduce cycle counts at the expense of datapath complexity. It is initially unclear as to whether or not this is a good tradeoff. We have introduced a variation of the datapath which includes these additional addressing modes. This model, I4C8S4C, must permit an address addition and a memory access within the same pipeline stage. This has a very significant impact on cycle time in this architecture and is unlikely to be beneficial, however, it can give us some understanding of the utility of these addressing modes. Another model, I4C8S5, is a more realistic way to add complex addressing modes. The pipeline has been extended into separate execute and memory stages as is done in most simple RISC processors. The negative impact of this adjustment is that a load-use delay of one cycle is now required. Furthermore in a VLIW cluster, 4 additional bypass paths are required. We have assumed that the more complex bypassing has a slight impact on cycle time. Table 1 summarizes the area requirement and relative clock speed estimates for these datapaths.

12-ported register file - 128 registers	3.0 mm^2
4 ALUs	0.4 mm^2 each
8-bit multiplier	1.0 mm^2
shifter	0.5 mm^2
32K Local RAM	12.9 mm^2
Bypass logic, pipeline registers, etc.	0.4 mm^2
Local Routing Overhead	1.9 mm^2
Cluster Area	21.3 mm^2
8 clusters + 32x32 crossbar	181.4 mm^2 datapath.

Fig. 5 - Area for Datapath I4C8S4

These 3 datapath models are quite similar, favoring programmability while maintaining a high clock rate and a large number of execution resources. As an alternative, we modified the datapath as necessary to further increase the clock rate into the 1GHz range. This requires a faster crossbar network, faster register files, faster RAM, and simpler bypassing as well as some modifications to the pipelining scheme. To accomplish these goals, we constructed a datapath with 16 small clusters rather than 8 moderately-sized clusters. Each cluster in the new datapath model, called I2C16S4, has only two issue slots. This allows a smaller 6-ported, 64-entry register file to suffice. Since there are twice as many clusters, each need only supply 16KB of local data RAM, however even this smaller RAM is too slow. Instead, two separate 8KB data memories are provided in each cluster. Also, the multiplier must now be pipelined to two stages. Each issue slot can now support either an ALU operation or a load/store operation to a specific one of the local memories. One of the issue slots can alternatively perform a multiply and the other can perform a shift. In order to increase the speed of the crossbar network, only 1 port to a 16x16 switch is provided to each cluster rather than one for each issue slot. Only simple addressing modes are supported and the smaller cluster size naturally reduces the bypass logic complexity. Despite all of these compromises, we estimate that only a 30% increase in clock rate is likely. Finally, a variation of this 16-cluster datapath was designed to support more flexible memory addressing. This model, I2C16S5, uses a 5-stage pipeline with a 1-cycle load-use delay. Also, rather than splitting the data RAM into two separate memory spaces, we have increased the cell size and moved the pipeline stage boundary to after the decode logic to allow a single 16KB memory to function at this speed. This produces a significant area penalty as seen in Table 1.

These datapath architectures serve as a reasonable initial exploration into the programming issues involved in this machine. The 30% clock speed improvement provided by the reduced-functionality clusters is substantial; however it involves numerous compromises in interconnect including an increased decentralization of storage resources. The conventional wisdom is that this greatly increases the

[1] Double buffering is a common memory strategy for signal processing systems. More advanced versions with several banks can improve memory utilization and may prove useful.

complexity of programming and generally results in less effective compilers. In our experiments, we attempt to quantify the impact of the additional complexity for hand-coded examples, providing some insight into the potential impact on final system performance although not fully addressing the issue of compilation.

Although the models under evaluation are intended to compare some datapath tradeoffs, the programming models did take into account the reality of a finite instruction cache. The 8-cluster models (I4C8S4, I4C8S4C, I4C8S5) assume a 1K instruction on-chip cache while the 16-cluster models assume only 512 instructions in the cache due to the increased clock speed. Since traditional demand-driven instruction cache refills are likely to be in excess of 100 cycles for this type of processor, this places overwhelming restrictions on practical scheduling approaches. Essentially, all critical loops must fit into the cache.

3.3. Hand-scheduled Algorithms

These architectural models are compared via simulation of hand-scheduled application kernels. Although hand scheduled code is not the best quantitative mechanism for evaluating architectural tradeoffs, it provides a number of advantages in the early stages of system development. The most compelling justification for simulating hand-scheduled code is that it is practical. It takes far longer to develop or retarget a compiler for each of these datapath models than to hand-code a reasonable number of illustrative examples. Furthermore, a first-generation compiler is likely to have numerous inefficiencies in optimization and be biased towards particular architectural features thus providing no more accurate information than hand scheduling. Perhaps more importantly at this stage in architecture development, the process of having skilled system designers perform the detailed scheduling allows them to identify specific features of the architecture that can impede the code scheduling process or act as bottlenecks to good performance. Opportunities for these specific observations are often concealed when using a compiler. Finally, the hand-scheduling process is an opportunity to try out specific compilation strategies in such a way that minor obstacles are avoided and the compilation techniques are fully exploited. This provides a strong indication of which compilation mechanisms should be included into a research compiler.

It is only practical to schedule small algorithmic kernels rather than real complete applications, however iterative kernels tend to dominate signal processing code even more so than general-purpose or scientific code so they are likely to indicate final performance. The kernels we selected are either extracted from real video applications or constructed from algorithms in textbooks. They range from about 25-200 lines of procedural `C` code. In scheduling and optimizing each of the kernels, we tried to use techniques that could practically be used by a compiler. Most importantly, we performed aggressive optimization based on information that can be derived from the code specification but did not depend on knowledge of the algorithm other than what could be extracted from the `C` source. Whenever possible we based our scheduling strategy on well known algorithms such as loop unrolling, list scheduling and software pipelining. We also aggressively applied scalar optimizations such as common subexpression elimination and strength reduction. The goal was to generate object code comparable to what could be expected from the best production compilers.

Six different specific kernels from MPEG encoder applications were evaluated. The first two algorithms, Full Motion Search and Three-step Search, are two different algorithms for comparing 16x16 pixel macroblocks from consecutive frames in order to estimate the most likely direction and distance of motion for objects represented by those samples. This is generally believed to be the most time-consuming step in video compression. The inner-loops of both algorithms are identical but some outer loops that determine the search strategy differ substantially. As such, the same sequence of optimization techniques was applied to both algorithms but the effectiveness varies. The baseline implementation provides a point of reference for interpreting the scheduled VLIW code. This is sequential code using the full capabilities of the machine including predicated execution but limited to one operation per instruction. Scalar optimizations are aggressively applied including pipeline scheduling to fill load-delay and branch-delay slots, but code is not moved between basic blocks other than loop invariant code. A second baseline sequential version of each search algorithm is then implemented where the innermost loop is fully unrolled, eliminating many branch operations and some loop-index and address arithmetic. This represents a fairer starting point for comparing sequential and parallel code since this type of unrolling is implicit in the parallel scheduling algorithms we have used.

Once the sequential code was measured, software pipelining was used to schedule parallel code on one cluster. Since there is abundant parallelism in both of these algorithms due to the outer loop levels being **do-all** type loops (representing repeating the search for each macroblock in a frame), it is possible to perform several searches in a SIMD style rather than scheduling a single search across several clusters. This is effective for these particular algorithms when the inner loops are unrolled. Although we used software pipelining to schedule each cluster, in this type of code where the inner loops have fixed iteration counts and can be fully unrolled, branch and bound scheduling or even list scheduling of inner loops is equally effective. A second version of the software-pipelined code was then implemented in which the second-level inner loop was unrolled as well. The shorter program length due to multiple operations per instruction make this feasible. Finally, in addition to the VLIW-style software pipelining we applied some data blocking techniques as are typical in vectorizing compilers. This involves exchanging levels of loops and sometimes splitting loops into several separate loops in order to increase the register lifetimes of particular values and thus eliminate load and store operations. Essentially one attempts to perform all required operations on a small block of data before using other data in the arrays. Due to the perceived importance of motion search in video, we also considered the effect of an architectural modification on each of our models at this point. A special operator, absolute difference, can be included in the cluster datapath to assist in motion estimation. This replaces two operations with one at the expense of doubling the area cost of one of the ALUs and adding about 2 gate delays to that ALU's critical path. The last two versions of each search program were rescheduled to take advantage of such an operator.

The third and fourth algorithms to be evaluated are different implementations of the two-dimensional *discrete cosine transform* or *DCT*, a space to frequency transform used as a critical stage in many compression methods. The traditional implementation computes each element of the transform on an 8x8 block of pixels directly. The row/column algorithm first performs a one-dimensional DCT on each row of the block then performs a one-dimensional DCT on each column.

A similar sequence of code implementations was constructed for the DCT algorithms. The initial sequential implementation employs numerous scalar optimizations but in this case does not use predication. As before, unrolling the inner loop for a sequential implementation greatly reduced the total operation count by removing branches and some computations and increasing the opportunities to fill delay slots. As a first attempt to generate parallel code, list scheduling was used on the unrolled inner loops to generate parallel code for one cluster and once again SIMD-style replication was used to generate code for the remaining clusters. In addition to scheduling, some additional CSE was performed to reduce the total number of arithmetic operations. A second parallel implementation uses software pipelining instead of list scheduling and incorporated predication as well. A further optimization uses numerical analysis to eliminate arithmetic operations that do not contribute significantly to the retained precision of the final result. The final version of the code for each algorithm unrolled the second-level loop in addition to the first before scheduling. This requires more registers than are available in one cluster and would exceed the size of the instruction cache; therefore, the code is scheduled across four clusters in order to gain extra resources. This is then replicated SIMD style.

The fifth algorithm is a color space (RGB to YC_rC_b) conversion and color subsampling (4:4:4 to 4:2:0) routine that is typical of the first stage in compression. The algorithm contains numerous arithmetic operations as well as several paths through the inner loop in order to handle block boundaries. Sequential versions of the code were generated both with and without loop unrolling. Many of the branches depend only on loop index values and thus can be eliminated with unrolling but fully unrolling the loop would exceed cache and register availability. Parallel versions of the code were generated using list scheduling of the unrolled loop and using both software pipelining and predication.

The final algorithm differs substantially from the earlier codes. The Variable-Bit-Rate Coder (VBR) is the final lossless encoding stage in MPEG-style compression. This particular code combines run-length and Huffman coding into a single algorithm. Typically it is considered a minor stage in the compression procedure, but it contains numerous long dependency chains and has very limited parallelism. Sequential versions of this algorithm were implemented both with and without predication and then a parallel version of each was list scheduled. Replication is not possible since dependencies exist between blocks of pixels; therefore, the entire 33-issue machine was available to the list scheduler. Additional attempts were made to use software pipelining on this algorithm and to split the computation into multiple phases and separately apply software pipelining to each.

3.4. Results

The performance simulations of the hand-coded examples include a substantial quantity of raw data. Six different algorithms were coded onto 5 different datapath models using 4-7 different schedules for each for a total of 180 simulations. These data provide substantial information about the effectiveness of the various scheduling strategies and the impact of specific datapath tradeoffs, however they must be interpreted with great care. The data represent a number of different attempts to exploit this architecture rather than a single controlled experiment and thus only similar trials should be compared for specific quantitative differences. The more valuable results of the experiments are the explanations of the performance variations since these represent specific bottlenecks in the datapath that limit performance or restrict scheduling or additional resources that may or may not benefit actual code. Table 1 shows the simulated cycle count for each experiment for one 720x480 pixel video frame as well as our best estimates of the datapath area and the relative clock speed for each datapath model.

3.4.1. Full Motion Search

The innermost loop of the motion search algorithm contains two loads, two address calculations, and several arithmetic operations on the pixel data and the loop-bounds test and branch. There are enough operations in this inner loop to fill the few delay slots present in any of the five datapath models so the sequential code shows no variation in performance. Once the inner loop is unrolled, most of the compare and branch operations are eliminated, improving performance, and in the three models that allow complex addressing modes, the address calculations can be incorporated into the load operations further improving performance. There are several dependencies between successive iterations of the inner loop, but none of the dependency chains have a length greater than one. This allows the code to be software pipelined with an initiation interval of one cycle without any complex transformations if adequate resources are available. In the initial software pipelining implementation, there are not enough resources in a single cluster to support this issue rate. The limiting resource in the data shown for the first 3 models is the load/store unit which is limited to one load per cluster per cycle requiring an initiation interval of 2 cycles. Since the load units are the critical bottleneck, the variations in the addressing models only have a tiny impact on the performance. The last two datapath models with only two issue slots per cluster are limited by the availability of issue slots. These clusters take longer to compute a single iteration of a loop, using iteration intervals of 2.5 and 3.5 cycles, but there are twice as many clusters available so overall throughput increases. Most importantly, since the total number of load/store units is doubled in the I2C16S5 model and quadrupled in the I2C16S4 model as compared to the others, this is no longer a bottleneck and the overall performance increases substantially. The iteration interval could have been reduced to 1 cycle or even below 1 cycle on any of these machines by scheduling each loop body across multiple clusters to provide more resources; however, for this type of algorithm this never provides any better overall performance than replicating the computation across clusters in a data parallel style. The overall improvement in cycle count over a sequential implementation of essentially the same code varies from 19.1x to 30.3x on these 33-issue machines.

Decreasing the total number of operations required by unrolling an additional loop level improves performance an additional 6%-15%, primarily due to reductions in the amount of prologue and epilogue code. As further expected, adding a specialized absolute difference operation improved performance on the I2C16* models but not on the load-limited I4C8* models. Since a great deal of the issue bandwidth is consumed by load instructions, the next step is to apply aggressive blocking transformations to the code in order to eliminate loads. This involves exchanging inner and outer loops in order to increase the lifetime of some data values, replicating temporaries to account for the changes in control flow, and taking advantage of the unrolled loop structure to implement aggressive register renaming. Using these transformations, more than 90% of the load operations can be eliminated as well as corresponding address computations. In addition to greatly improving overall performance, this eliminates the differences among datapath models. Every model has adequate load bandwidth and uses only simple address calculations. Overall issue rate is now the only limiting factor. Since issue rate limits performance, an absolute difference operation substantially improves performance.

In addition to the data in Table 1, we evaluated the benefits of including two load/store units in the I4C8* models using dual-ported memories. As expected, they reduced cycle counts to approximately match the I2C16* models in the situations where they had previously been limited by load bandwidth. However, since this is expensive and the benefit disappears when the most aggressive scheduling mechanisms are used, this did not seem appropriate. None of the other algorithms tested are load/store limited.

3.4.2. *Three-step Search*

The inner loops of the three-step search are identical to the full-motion search but the outer loops are data dependent. This greatly reduces the total number of operations required, but it limits the opportunities for aggressive optimizations based on blocking. The sequential and software pipelined versions of the algorithms use similar scheduling techniques as in the full motion search and show similar results. Speedups due to software pipelining range from 19.0x-30.3x. Once again, the I4C8* models are load limited while the others are issue-slot limited. The biggest difference in this algorithm is the fact that its overall improvement in performance as compared to full motion search is primarily based on eliminating similar comparisons and thus it does not have the opportunities for data reuse via blocking. The number of load operations and address computations eliminated via blocking is far less than in full motion search thus the improvement over simple software pipelining is not as large. The I4C8S4C and I4C8S5 models are both load limited and issue-slot limited. The remaining models are issue-slot limited and thus the I4C8S4 and I2C16S4 models require extra cycles for address calculations. These issue-slot limited modes benefit somewhat from an absolute difference operation but not as much as the full motion search algorithms did. The initial improvement of approximately 10x over full motion search is reduced to as little as 5x due to the restricted opportunities for optimization.

3.4.3. *Discrete Cosine Transform*

The initial sequential implementations of both DCT algorithms devote a majority of their cycles to loop-closing branches and unfilled branch-delay slots. This is due to the presence of a tiny dominant inner loop. This is easily solved in the second sequential implementation by unrolling the innermost loop. The sequential code shows a noticeable penalty for address computation in the I4C8S4 and I2C16S4 models but no penalty for load delays in the 5-stage pipeline models. The list-scheduled versions of the DCT code show high degrees of parallelism ranging from 18.0x to 36.9x reflecting the additional general-purpose optimizations that were exposed during the scheduling process. Large segments of the code are limited by multiply resources, therefore the I2C16S4 and I2C16S5 models that contain 16 multipliers instead of 8 perform better overall. Software pipelining with predication performed somewhat better in all cases primarily due to the elimination of branches rather than a fundamental difference in the scheduling approaches. The use of a five-stage pipeline improved the performance of the I2C16S5 model but not the multiply-bound I4C8S5 model.

The DCT requires multiplying numbers greater than 8 bits in length. Since these models only include 8x8 multipliers, this can require as many as 21 issue slots and at least 8 cycles as well as several extra registers for each 16x16 multiply. This has a major effect on performance for this algorithm and probably for many other traditional signal processing algorithms like filters. Aggressive numerical analysis can reduce the multiply penalty substantially by using less than complete 16x16 multiplies, but these operations still are the dominant performance bottleneck. Table 2 introduces two new architectural configurations, I4C8S5M16 and I2C16S5M16. These include 16-bit 2-stage multipliers rather than 8-bit single-stage multipliers. Since these require 2 execute stages in the pipeline, they are only simulated in the 5-stage pipeline models and require a multiply-use delay of 1 cycle. They only produce 16-bits of result each cycle due to writeback port limitations and thus require 2 cycles if all 32-bits are to be retained. The 16-bit multipliers improve DCT performance by 3x-5x. Performance of the other tested algorithms is not significantly affected.

Datapath Model	I4C8S4	I4C8S4C	I4C8S5	I2C16S4	I2C16S5
Estimated Relative Clock Speed	1.0	0.6	0.95	1.3	1.3
Estimated Area	181.4 mm^2	181.4 mm^2	183.5 mm^2	180 mm^2	217 mm^2
Full Motion Search					
Sequential–predicated	815.7M	815.7M	815.7M	815.7M	815.7M
Unrolled Inner Loop	633.2M	467.3M	467.3M	633.2M	467.3M
SW pipelined & unrolled	25.70M	24.41M	24.41M	20.91M	16.42M
SW pipelined & unrolled 2 lev.	22.33M	22.25M	22.25M	19.55M	13.99M
Add spec. op (> cycle & area)	22.29M	22.20M	22.20M	16.78M	11.21M
Blocking/Loop Exchange	9.44M	9.44M	9.44M	9.44M	9.44M
Add spec. op (> cycle & area)	6.85M	6.85M	6.85M	6.85M	6.85M
Three-step Search					
Sequential–predicated	86.12M	86.12M	86.12M	86.12M	86.12M
Unrolled Inner Loop	66.88M	49.20M	49.20M	66.88M	49.20M
SW pipelined & unrolled	2.72M	2.59M	2.59M	2.21M	1.74M
SW pipelined & unrolled 2 lev.	2.37M	2.36M	2.36M	2.07M	1.48M
Add spec. op (> cycle & area)	2.36M	2.35M	2.35M	1.78M	1.19M
Blocking/Loop Exchange	1.62M	1.33M	1.33M	1.60M	1.32M
Add spec. op (> cycle & area)	1.33M	1.33M	1.33M	1.32M	1.02M
DCT - traditional					
Sequential-unoptimized	703.1M	692.2M	692.2M	702.1M	692.2M
Unrolled inner loop	305.5M	303.1M	303.1M	305.5M	303.1M
List Scheduled	18.55M	18.14M	18.55M	11.03M	10.33M
SW pipelined & predicated	14.79M	14.75M	14.79M	10.70M	10.01M
+arithmetic optimization	13.71M	13.03M	13.71M	8.46M	7.77M
+unroll 2 levels & widen	13.92M	13.90M	13.92M	10.17M	9.48M
DCT -row/column					
Sequential-unoptimized	135.0M	129.5M	129.5M	135.0M	129.5M
Unrolled inner loop	97.98M	92.45M	92.45M	97.98M	92.45M
List Scheduled	4.92M	4.84M	4.92M	3.33M	3.15M
SW pipelined & predicated	4.58M	4.43M	4.58M	3.25M	3.07M
+arithmetic optimization	2.85M	2.84M	2.85M	2.30M	2.13M
+unroll 2 levels & widen	2.70M	2.70M	2.70M	2.38M	2.20M
RGB:YC_rC_b					
Sequential	15.15M	13.24M	13.24M	15.15M	13.24M
Sequential–unrolled	12.15M	10.42M	10.42M	12.15M	10.42M
List-scheduled	0.59M	0.59M	0.64M	0.40M	0.39M
SW Pipelined & predicated	0.46M	0.41M	0.42M	0.40M	0.38M
Variable-Bit-Rate Coder					
Sequential	4.44M	4.21M	4.44M	4.44M	4.44M
Sequential–predicated	4.37M	4.02M	4.37M	4.37M	4.37M
List-scheduled	2.62M	2.62M	2.96M	2.74M	2.74M
List-scheduled–predicated	1.78M	1.76M	1.78M	1.99M	1.99M
SW pipelined + comp. pred.	1.81M	1.79M	1.81M	2.01M	2.01M
+phase pipelining	1.76M	1.75M	1.76M	1.95M	1.93M

Table 1: Performance Simulations for Hand-scheduled Code (Cycles per 720x480 frame).

Datapath Model	I4C8S4	I4C8S5	I4C8S5M16	I2C16S5	I2C16S5M16
Estimated Relative Clock Speed	1.0	0.95	0.95	1.3	1.3
Estimated Area	181.4 mm^2	183.5 mm^2	199.5 mm^2	217 mm^2	249 mm^2
DCT - traditional					
Sequential-unoptimized	703.1M	692.2M	271.9M	692.2M	271.9M
Unrolled inner loop	305.5M	303.1M	117.5M	303.1M	117.5M
List Scheduled	18.55M	18.55M	5.98M	20.67M	3.90M
SW pipelined & predicated	14.79M	14.79M	4.68M	20.03M	3.38M
+unroll 2 levels & widen	13.92M	13.92M	3.95M	18.96M	1.91M
DCT -row/column					
Sequential-unoptimized	135.0M	129.5M	63.16M	129.5M	63.16M
Unrolled inner loop	97.98M	92.45M	25.23M	92.45M	25.23M
List Scheduled	4.92M	4.92M	1.29M	6.31M	0.80M
SW pipelined & predicated	4.58M	4.58M	1.03M	6.15M	0.77M
+unroll 2 levels & widen	2.70M	2.70M	0.86M	4.41M	0.61M

Table 2: Impact of 16-bit Multipliers.

The performance of these implementations is still somewhat limited by the length of the unrolled inner loop. Unrolling by 8 (the inner loop count) and then scheduling on 4-wide clusters still results in short code sequences and in the case of software pipelining allows the prologue and epilogue code to have a substantial impact. The obvious approach to investigate is the unrolling of the second-level loop but this requires more register and icache resources than one cluster can provide. In many of the DCT schedules, there is a clear benefit in scheduling across multiple clusters to increase resource availability. A four-cluster implementations with 2-level loop unrolling provides the best performance on several models, particularly those that include 16-bit multipliers. The additional multipliers in the 16-cluster datapaths still provide a significant performance boost.

3.4.4. Color-space conversion

The results of the color-space conversion are quite similar to the DCT. The sequential code is somewhat burdened by branching. Much of this is eliminated through loop unrolling. The parallel versions of the code are primarily limited by the availability of the multiplier and shifter. The additional resources available in the I2C16S4 and I2C16S5 somewhat improve performance.

3.4.5. VBR encoder

The VBR encoder provides completely different characteristics than the previous examples. Unlike those algorithms which provide abundant parallelism, the VBR encoder has very little. The code is dominated by many tight branches and long dependency chains. Also, the cycle count is data dependent so it was simulated with typical data extracted from video. In the case of the sequential code, complex addressing capabilities only are of benefit when they do not increase the pipeline length. Predication provides only a minimal improvement despite the large number of branches because the conditions cannot be computed early. Simple list scheduling only improves performance by at best a factor of 1.7x. Predication has more impact on the list-scheduled code by increasing basic block size and exposing more opportunities for scheduling, but the best improvement is still only 2.5x over predicated sequential code. Complex attempts at software pipelining and aggressive speculative parallelism provided only a minimal increase in performance since the long dependency chains could not be broken for the most common paths. Since there is no potential for replicating the computation, the additional resources in the I2C16S4 and I4C16S5 models were not of any benefit. The increased communication latency to use these resources increased the cycle count in the high-performance implementations, but not enough so to overcome the cycle time advantage.

4. Conclusions

In many ways the results validate both our design methodology and our architectural approach; however, there are some interesting surprises as well. The data show that it is possible to design a VLIW VSP in a .25μ process that will provide adequate and in some cases exceptional performance in current real-time video tasks. These processors will be capable of performing a real-time full-motion search on CCIR-601 video using only 33%-46% of compute time. This is comparable to or better than today's best dedicated chips. The VLSI simulations show that it is possible to have a richly connected, very-wide word machine with an extremely fast (650MHz-850MHz) clock rate. The code simulations show that for several current key VSP algorithm, it is possible to effectively utilize the resources of this machine, exceeding 15GOPS sustained performance for large periods of time. By quantifying the area and cycle-time tradeoffs of the key VLSI modules in advance of selecting architectural parameters, we are able to explore the highest performance alternatives within the design space, those that allow very high clock rates and maximally exploit available area.

Several specific conclusions about these experiments can be drawn from the data:

- Most of the examples demonstrate a great deal of possible parallelism. The VBR coder is the exception, however the 2.5x parallelism available via the VLIW ILP mechanisms still makes a significant difference in overall performance.
- Resource limitations are the primary bottleneck preventing higher performance including load bandwidth, multiply bandwidth, and issue rate. No single resource limited the performance of a majority of the examples indicating a relatively balanced design; however, small multipliers significantly slow down the DCT computations. Including 16x16 bit multipliers is probably indicated.
- The global interconnect was severely underutilized in these examples although the connectivity provided by the local register files was heavily exploited. It is likely that this is partially due to the use of homogeneous clusters such that there is always a local version of every resource available. There are many instances where it is advantageous or necessary to move data between clusters but they seldom arose in these examples. In any case, even total elimination of the global interconnect network would only reduce chip area by about 3%.
- Load-use delays present in the models with 5-stage pipelines rarely increased execution time.
- Complex addressing modes improved performance on several examples but only minimally on the most highly optimized code.
- The working set for these typical VSP algorithms never exceeded 4K bytes/cluster thus an 8K byte memory would suffice as opposed to the 16K-32K byte memories we simulated. This could save up to 40% in datapath area and open opportunities to alleviate some bottlenecks though additional issue slots and functional units or larger multipliers. This may, however, be overly aggressive since the performance can drop rapidly when the working set exceeds the available storage and we still have little knowledge about the memory requirements of compiled code or next-generation video applications.
- Small clusters with limited global interconnect provided better performance than the initial design point. In most cases the additional resources provided to compensate for the limited communications reduced the overall cycle count. In addition to this architectural improvement, the small-cluster machines are estimated to provide a 30% faster clock rate. The combined performance improvement ranges from 17% to 129% faster than the initial I4C8S4 model.

This final observation is the most surprising and in many ways the most disturbing conclusion from these experiments. The less flexible programming model of the small-cluster datapaths did not have any appreciable impact on the performance of the tested algorithms. On the contrary, these algorithms benefited from the largest number of resources and the fastest clock speed possible with little regard for connectivity. This contradicts the conventional wisdom about VLIW architectures. This may be a characteristic of all VSP algorithms or it may be unique to the set selected here.

Despite the measured advantages of the small-cluster datapaths, we are still quite hesitant to commit to the conclusion that they are better for a real VLIW VSP system. The clock rates in question are already at the state of the art for .25µ processors; choosing to increase it from 650MHz to 850MHz will further complicate signal distribution and power issues. Also, the additional complexity of adapting a compiler to handle the reduced connectivity of these models is yet another obstacle to generating high-performance code for real applications. The most rational solution is to continue to maintain both models as reasonable candidate architectures through compiler development.

We are continuing to work on several open problems. Based on these results, we will examine some additional datapath models to see if we can improve on the performance of the ones proposed here while maintaining a relatively simple compilation model. Concurrently, we are beginning to target a version of the IMPACT compiler from Univ. of Illinois to generate code for the I4C8S4 datapath model. We would have liked to explore a wider range of tradeoffs, but the time involved in hand scheduling code is substantial. We are trying to develop an analytical model that matches these results for a limited class of applications. This would allow exploration of a wider range of alternatives at the expense of accuracy. This could suggest additional candidates for the type of evaluation we have performed here. We are also continuing to improve our VLSI designs and to generate designs for arithmetic units and for the control-path elements. Eventually we intend to develop enough design infrastructure and the compilation and simulation tools to be able to quantitatively measure the performance of these architectures on compiled code from complete production applications.

5. References

[1] S. Dutta, *VLSI issues for Video Signal Processors*, Ph.D. Thesis, Princeton Univ. 1996.

[2] S. Dutta, W. Wolf, and A. Wolfe, "VLSI Issues in Memory-System Design for Video Signal Processors," in proc. of ICCD '95, pp. 498-505, Oct. 1995.

[3] K. Ebcioglu, Some Design Ideas for a VLIW Architecture for Sequential-Natured Software, IBM Research Report, April 1988.

[4] J. A. Fisher and B. R. Rau, "Instruction-Level Parallel Processing," Science, vol. 253, no. .5025, pp. 1233-1241, Sept. 13, 1991.

[5] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh, "VIPER: A 25-MHz 100MIPS Peak VLIW Microprocessor", in proc. of 1993 IEEE Custom Integrated Circuits Conf., pp. 4.4.1-4.4.5, 1993.

[6] J. Labrousse and G. Slavenburg, "A 50MHz microprocessor with a VLIW architecture," in proc. International Solid State Circuits Conf., San Francisco, 1990.

[7] W. Maly, M. Patyra, A. Primatic, V. Raghavan, T. Storey, and A. Wolfe, "Memory Chip for 24-Port Global Register File," in proc. IEEE Custom Integrated Circuits Conference, San Diego, May 1991.

[8] N. Ohkubo, et al., "A 4.4ns CMOS 54x54-b Multiplier Using Pass-transistor Multiplexer," in proc. of 1994 IEEE Custom Integrated Circuits Conf., pp. 26.4.1-26.4.4, 1994.

[9] M. Suzuki, et al., "A 1.5ns, 32b CMOS ALU in Double Pass-Transistor Logic," in International Solid State Circuits Conf.. pp. 90-91, 1993.

[10] S. Dutta, K. O'Connor, and A. Wolfe, "High-Performance Crossbar Interconnect for a VLIW Video Signal Processor", in 1996 IEEE ASIC Conference, pp. 45-50, Sept. 1996.

[11] T. Komarek and P. Pirsch, "Array architectures for block-matching algorithms," IEEE Trans. on Circ. and Sys., 36(10), Oct. 1989, pp. 1301-1308.

[12] Santanu Dutta and Wayne Wolf, "Processing element architectures for programmable video signal processors," in VLSI Signal Processing VIII, IEEE, 1995, pp. 401-410.

[13] M. Yamashina, *et. al*, "A microprogrammable real-time video signal processor (VSP) for motion compensation," IEEE J. Solid-State Circuits, 23(4), Aug., 1988, pp. 907-914.

[14] H. Fujiwara, M. Loiu, M. Sun, K. Yang, M. Maruyama, K. Shomura, and K. Ohyama, "An all-ASIC implementation of a low bit-rate video codec," IEEE Trans. Circ. Sys for Video Tech., 2(2), June, 1992, pp. 123-133.

[15] S. Dutta, A. Wolfe, W. Wolf, and K. O'Conner, "Design Issues for a Very-Long-Instruction-Word VLSI Video Signal Processor," in VLSI Signal Processing IX, pp. 95-104, Oct. 1996.