A Low Power Architecture for Embedded Perception

Binu Mathew, Al Davis, Mike Parker
School of Computing, University of Utah
Salt Late City, UT 84112
{mbinu | ald | map}@cs.utah.edu

ABSTRACT

Recognizing speech, gestures, and visual features are important interface capabilities for future embedded mobile systems. Unfortunately, the real-time performance requirements of complex perception applications cannot be met by current embedded processors and often even exceed the performance of high performance microprocessors whose energy consumption far exceeds embedded energy budgets. Though custom ASICs provide a solution to this problem, they incur expensive and lengthy design cycles and are inflexible. This paper introduces a VLIW perception processor which uses a combination of clustered function units, compiler controlled dataflow and compiler controlled clock-gating in conjunction with a scratch-pad memory system to achieve high performance for perceptual algorithms at low energy consumption. The architecture is evaluated using ten benchmark applications taken from complex speech and visual feature recognition, security, and signal processing domains. The energy-delay product of a 0.13μ implementation of this architecture is compared against ASICs and general purpose processors. Using a combination of Spice simulations and real processor power measurements, we show that the cluster running at 1 GHz clock frequency outperforms a 2.4 GHz Pentium 4 by a factor of 1.75 while simultaneously achieving 159 times better energy delay product than a low power Intel XScale embedded processor.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; C.1.1 [**Computer Systems Organization**]: Processor Architectures—*RISC/CISC, VLIW architectures*; C.-1.3 [**Computer Systems Organization**]: Processor Architectures—*Data-flow architectures*; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Mobile processors*

CASES'04, September 22–25, 2004, Washington, DC, USA.

General Terms

Performance, Design, Algorithms

Keywords

Embedded systems, Low power design, Speech recognition, Computer vision, Perception, Stream processor, VLIW

1. INTRODUCTION

The term *Perception Processing* encompasses processor support for technologies that can enable computers to perceive the world the way we humans do with our sensory faculties. It targets areas like object detection, recognition and tracking, speech and gesture recognition and multimodal abilities like lip reading to support speech recognition. The applications for perception processing are both immense and diverse. More and more computing devices are being invisibly embedded into our living environment and we notice their existence only when they cease to serve us. For this fledgling computing fabric to develop into tomorrows ubiquitous computing environment, the primary means of interacting with it should be human friendly ones like speech and gesture. Future mobile embedded environments need to at least support sophisticated applications such as speech recognition, visual feature recognition, secure wireless networking, and general media processing.

By their very nature perception applications are likely to be most useful in mobile embedded systems like intelligent PDAs, unmanned robots and prosthetic devices for vision and hearing impaired people. A fundamental problem that plagues these applications is that they require significantly more performance than current embedded processors can deliver. Most embedded and low-power processors, such as the Intel XScale, do not have the hardware resources and performance that would be necessary to support a full featured speech recognizer. Even modern high performance microprocessors are barely able to keep up with the real time requirements of sophisticated perception applications. Given Moore's law performance scaling, the performance issue is not by itself a critical problem. However two significant problems remain. First, the energy consumption that accompanies the required performance level is often orders of magnitude beyond typical embedded power budgets. Furthermore, the power requirements of new high performance processors is increasing. The conclusion is that technology scaling alone cannot solve this problem. Second, perception and security interfaces are by nature always operational.

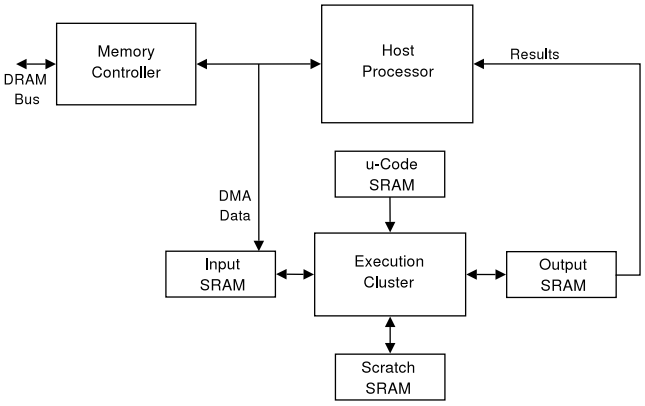

Figure 1: Coprocessor Organization

This limits the processor's availability for other compute tasks such as understanding what was perceived.

The usual solution to reducing power consumption while increasing performance is to use an ASIC. Given the complexity and the *always on* nature of perception tasks, a more relevant approach would be to use the ASIC as a coprocessor in conjunction with a low power host processor. In the initial phase of this research, an ASIC coprocessor for one of the dominant phases of the CMU Sphinx speech recognition system was investigated [17]. This effort reinforced the view that ASICs are costly and inflexible. Their high fabrication cost coupled with the costs associated with a lengthy design cycle are difficult to amortize except in high volume and margin situations. The inherent ASIC specialization makes it extremely difficult to support multiple applications, new methods, or even evolutionary algorithmic improvements. Given that embedded applications evolve rapidly and that embedded systems are extremely cost sensitive, these problems motivate the search for a more general purpose approach. The use of reconfigurable logic and FPGA devices is another common approach [5]. The inherent reconfigurability of FPGAs provides a level of specialization while retaining significant generality. However, FPGAs have a significant disadvantage both in performance and power when compared to either ASIC or CPU logic functions.

This paper describes a specialized processor architecture that can provide high performance for perception applications in an energy efficient manner. Our research addresses the rapid automated generation of low-power high performance VLIW processors for the perception domain. Such a domain optimized processor is intended to be used as a coprocessor for a general purpose host processor whose duty is to act on what the coprocessor perceives. A high level view of the architecture is shown in Figure 1. The host processor moves data into or out of the coprocessor via double buffered input and output SRAMs. Local storage for the cluster is provided by the scratch SRAM and the microcode program that controls the operation of the cluster is held in the u-Code SRAM. The execution cluster can be customized for a particular application by the selection of function units. In fact the type and number of function units, SRAMs, address generators, bit widths and interconnect topology are specified using a configuration file. The hardware design (Verilog HDL netlist) and a customized simulator are automatically generated by a processor generator tool. Henceforth the term perception processor refers to the generic architecture behind any domain specific processor created using the processor generator tool.

Perception algorithms tend to be stream oriented, i.e., they process a sequence of similar data records where the data records may be packets or blocks of speech signals, video frames or the output of other stream processing routines. Each input packet is processed by a relatively simple and regular algorithm that often refers to some limited local state tables or history to generate an output packet. The packets have fixed or variable but bounded sizes. The algorithms are typically loop oriented with dominant components being nested *for* loops with flow-dependent bodies. Processors which are optimized for this style of computation are called stream processors [21]. The perception processor is a low-power stream processor optimized for speech recognition and vision. However, general applicability to other stream oriented algorithms will be shown in Section 6.

Energy efficiency is primarily the result of minimized communication and activity. The compiler uses fine-grain clock gating to ensure that each function unit is active only when required. Compiler controlled dataflow permits software to explicitly address output and input stage pipeline registers of function units and orchestrate data transfer between them over software controlled bypass paths. Data values are transported only if necessary and the compiler takes care to ensure that value changes are visible on heavily loaded wires and forwarding paths only if a unit connected to that path needs the data value. By explicitly enabling pipeline registers the compiler is able to control the lifetime of function unit outputs and directly route data to other function units avoiding unnecessary access to a register file. The resulting dataflows or *active datapaths* resemble custom computational pipelines found in ASICs, but have the advantage of flexibility offered by software control. This may be thought of as a means of exploiting the natural register renaming that occurs when a multistage pipeline shifts and each individual pipeline register gets a new value. However the active datapath in the cluster will utilize multiplexer circuits that provide generality at the cost of power, area and performance. These muxes and the associated penalties will not be present in a custom ASIC design.

The resultant architecture is powerful enough to support complex perception algorithms, at energy consumption levels commensurate with mobile device requirements. The approach represents a middle ground between general purpose embedded processors and ASICs. It possesses a level of generality that cannot be achieved by a highly specialized ASIC, while delivering performance and energy efficiency that cannot be matched by general purpose processor architectures. In support of this claim, the approach is tested on ten benchmarks that were chosen both for their importance in future embedded systems as well as for their algorithmic variety. Seven represent key components of perception systems and the other three were chosen from the encryption and DSP domains to test generality of the approach outside of the perception domain. The perception processor is evaluated using Spice simulations of a 0.13μ CMOS implementation operating at 1 GHz [3, 4]. Its effectiveness is compared against the obvious competition: general purpose processors and ASICs. Comparison against conventional processors is problematic because energy efficient embedded

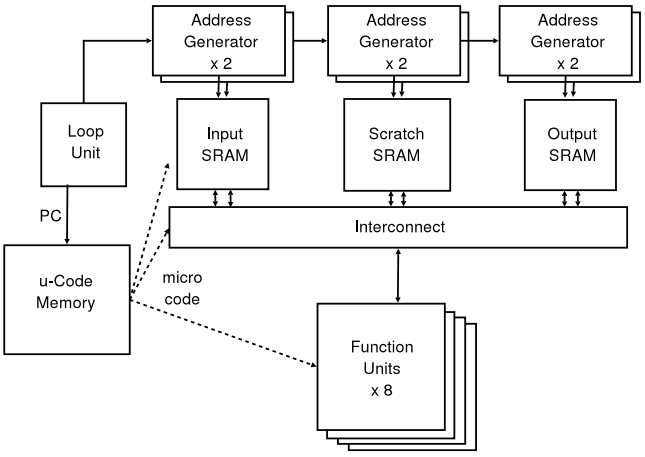

Figure 2: Perception Processor Organization

processors often do not have the performance and the floating point support required for perception applications while mainstream processors are optimized for performance rather than energy consumption. This paper therefore compares the perception processor against both a Pentium IV and an Intel XScale processor. For a subset of the benchmarks we also compare against custom ASIC implementations of the respective algorithms.

2. PERCEPTION PROCESSOR ARCHITECTURE

Figure 2 shows the internal organization of the perception processor. It consists of a set of clock gated function units, a loop unit, 3 dual ported SRAMs, 6 address generators (one for each SRAM port), local bypass paths between neighboring function units as well as a cluster wide interconnect. A register file is conspicuously absent because the combination of compiler controlled dataflow and a technique called array variable renaming makes a register file unnecessary [15]. Though none of the clusters described here need a register file it is possible to incorporate one into a function unit slot. Clusters can be configured to maximize the performance of any particular application or set of applications. Typically there will be a minimum number of integer ALUs as well as additional units that are more specialized. Hardware descriptions for the cluster and the interconnect are automatically generated by a cluster generator tool from a configuration description.

To understand the rationale behind this organization it is important to know that typical stream oriented loop kernels found in perception algorithms may be split into three components. They consist of control patterns, access patterns and compute patterns. The control pattern is typically a set of nested *for* loops. Access patterns seen in these algorithms are row and column walks of 2D arrays, vector accesses and more complex patterns produced when simple array accesses are interleaved or software pipelined. Compute patterns correspond to the dataflow between operators within the loop body. For example, the compute pattern of a vector dot product is a multiply-accumulate flow where a multiplier and an adder are cascaded and the adders output is fed back as one of its inputs.

Previous research showed that traditional wide issue out of order processors are unable to exploit the high levels of ILP available in perception algorithms on account of load store ports saturating before all the function units can be put to use [17, 16]. Improving IPC without significantly increasing hardware complexity is an important technique for reducing dynamic power consumption [14]. Hence, architectural solutions that improve data delivery to function units are beneficial in this situation. We are able to satisfy the twin goals of high performance and low power consumption by creating programmable primitives that accelerate each of the three patterns found in loops. The details of how the loop units and address generators implement control and access patterns in a generic VLIW processor are the subject of another publication [15]. Only a high level description of these units is provided here. This paper concentrates on energy efficient acceleration of compute patterns and the overall operation of the perception processor. The creative contribution of both papers are distinct though benchmarks and experiments are shared.

The loop unit accelerates the control patterns of algorithms. Encoded descriptions of nested *for* loops may be stored in context registers contained in this unit. A compiler can transfer a loop description into a context register using a single cycle instruction. Run-time loop variable values corresponding to various loops are maintained in loop count registers. Once a loop description has been entrusted to the loop unit, the unit works semi-autonomously. It periodically updates the loop variables and makes the loop counts available to other units. It is aware of nested loops, modulo scheduling, loop unrolling and software pipelining and can handle complicated update patterns to loop variables arising when such scheduling optimizations are done. The loop unit is similar to the control state machine of a custom circuit except that it is programmable. When a loop pattern is too complex to be handled in the loop unit, it may be implemented in software on the function units or alternately the loop unit may be extended to handle the new pattern.

A large number of SRAM ports are required to ensure data delivery to the execution units. Increasing the number of ports on a large SRAM increases power consumption and worsens access time. The traditional solution is to bank large SRAM structures like caches. The same reasoning motivates our choice of multiple software managed scratch SRAMs. To use these SRAM resources efficiently, it is necessary to generate addresses for each SRAM port with high throughput. Since there are 6 SRAM ports in the configuration shown in Figure 2, a large number of function units may be occupied for address calculation leaving inadequate resources to do the actual computation. The situation may be improved by attaching dedicated address generators to each SRAM port. These address generators use loop variables maintained by the loop unit to autonomously compute address expressions for 2D array accesses, vectors and indirect vectors of the form $A[B[i]]$. As in the case of the loop unit, the compiler stores descriptions of access patterns into context registers within the address generators. Thereafter, with the help of the loop unit, the address generators pump data at a high rate into the execution cluster to ensure high function unit utilization. They handle common access patterns found in perception algorithms. In uncommon cases the generic $A[B[i]]$ pattern may be used to implement the

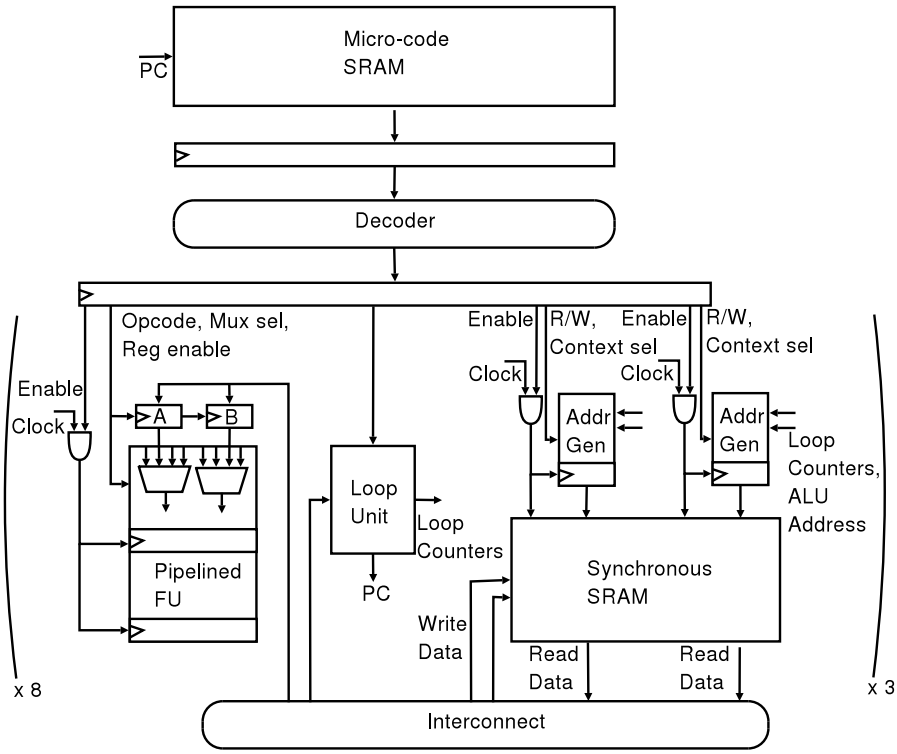

Figure 3: Pipeline Structure

access pattern. For example, by storing bit-reversed indices in $B[\,]$, bit-reversed access of $A[\,]$ is possible for FFT computations. It is also possible to use function units to compute and issue addresses like in a traditional architecture.

3. PIPELINE STRUCTURE

The perception processor architecture was designed to be able to emulate dataflows that typically occur within custom ASIC accelerators. To this end, it has a simple and rather different pipeline structure from a traditional processor. In sharp contrast to the typical five stage Instruction Fetch/Instruction Decode/Execute/Memory/Write Back (IF/ID/EX/MEM/WB) pipeline of a MIPS like RISC processor, the perception processor pipeline consists of just 3 stages: Fetch/Decode/Execute [9]. The number of actual stages in the final execute phase depends on the function unit. The pipeline structure is shown in Figure 3. Conspicuous departures from the RISC model include the absence of register lookups in the decode stage and the lack of memory and write back stages.

In the perception processor, the microinstructions are fetched from a very wide instruction memory which is more than 200 bits wide. The decode stage is minimal and is limited to performing sign or zero extensions to constants, generating NOPs for function units while the memory system is being reconfigured and generating clock enable signals for active function units. The wide instruction is then dispatched to a set of function units, a loop unit and a set of address generators. All resources including actual function units and SRAM ports appear as peers in the EX stage. The final output of all these peer units can be transferred back to the input of the units by an interconnect network. Nearest neighbors can be reached in the same cycle while reaching a non-neighboring unit incurs an additional cycle of latency.

In the MIPS RISC execution model, every single instruction implicitly encodes a path through the pipeline. An integer instruction takes the IF/ID/EX/MEM/WB path, while a floating point instruction takes a detour through the FPU in the EX stage. There is also an implicit hardware controlled timing regime that dictates the relative cycle time at which an instruction reaches each stage subject to dependences checked by interlocks.

In the perception processor, instructions do not encode any such implicit paths. The instructions are called microcode because they serve the traditional horizontal microcode function where individual bits directly control hardware functions like mux selects and register write enables. To get the functionality implied by a MIPS instruction, the stage by stage functionality of the MIPS instruction must be identified and the equivalent microinstruction bits set in several successive microinstruction words. The advantage of this lower level approach is that the hardware can be controlled in a fine grained fashion which is impossible in the MIPS case. For example, interconnect muxes may be set to route data between selected function units and memory in a manner which directly represents the dataflow graph of an algorithm and data may be streamed through the dynamically configured structure. The ability to reconfigure the structure through microcode on a cycle by cycle basis means that the function units may be virtualized to map flow-graphs which are too large to fit the processor. This manifests itself as higher loop initiation intervals and larger number of temporary results that need to be saved

or rerouted when compared to a processor that has enough physical resources to allocate to the entire flow-graph. Performance degrades gracefully under virtualization. The perception processor supplants the instruction centric RISC execution model with a data centric execution model which lends it the flexibility to efficiently mimic the styles of computation found in VLIW and vector processors as well as custom ASIC datapaths.

3.1 Function Units

Function units follow the generic organization shown in Figure 4. Their operands may be the output of their own final stage or the output of their left or right neighbor. In addition an operand may also arrive over the interconnect in which case the transferred value is first latched in a register. Several types of function units are used in this study.

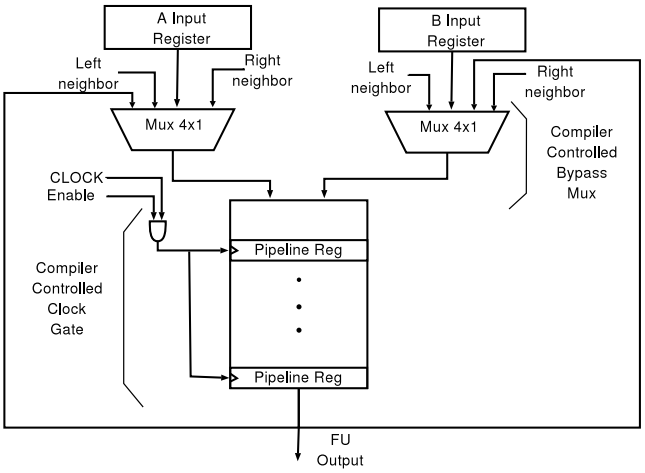

Figure 4: Function Unit Architecture

Integer ALUs perform common operations like add, subtract, xor etc. ALUs also have compare instructions which not only return a value, but also set condition codes local to the particular ALU. Conditional move operations may be predicated on the condition codes set by previous compare instructions to route one of the two ALU inputs to the output. This makes if-conversion and conditional data flows possible. All ALU operations have single cycle latency.

FPUs support floating point add, subtract, multiply, compare and integer to floating point convert operations. While the FPU is IEEE 754 compatible at its interfaces, for multiply operations it internally uses a reduced precision of 13 bits of mantissa since it has been demonstrated that our target applications work well with this precision [17]. Reduced precision in the multiplier contributes significant area and energy savings. All FPU operations have 7 cycle latency.

Multiply units support 32-bit integer multiply operations with 3 cycle latency.

In order to illustrate the advantages of fine grain pipeline control and modulo support and to demonstrate our generality claims, no application specific instructions have been added to the function units with two exceptions: the reduced precision of floating point multiplies and byte *select/merge* instructions which select an individual byte from a word. The latter is similar to the pack/unpack instruction in Intel's IA-64 architecture or the *AL/AH* register fields in the IA-32 architecture. These instructions significantly ease dealing with RGB images.

3.2 Interconnect

As CMOS technology scales, wire delays get relatively worse. The cluster interconnect reflects our belief that future architectures will need to explicitly address communication at the ISA level. The local bypass muxes in each function unit are intended for fast, frequent communication with the immediate function unit neighbors. The interconnect supports communication with non-neighbor function units and SRAMs. Such communications have a latency of one cycle. In a multicluster configuration, intercluster communication will incur even larger delays. Values transferred via the interconnect to the input registers of a function unit may be held indefinitely which is useful for caching common constants.

In modulo scheduled loops, each resource may be used only during one modulo period. Reusing a resource later will render the loop body unschedulable. It is common to find a lot of data reads early in the loop body and a few stores toward the end that correspond to computed values graduating. Conflicts in the interconnect often make modulo scheduling difficult. We found it useful to partition interconnect muxes by direction so as to reduce scheduling conflicts. Incoming muxes transfer data between function units and from SRAM ports to function units while outgoing muxes are dedicated to transferring function unit outputs to SRAM write ports. Another common occurrence is that two operands need to be made available at a function unit as part of a dataflow but interconnect conflicts make such a transfer impossible. In such cases it might be possible to transfer one operand in an earlier cycle and freeze its destination pipeline register using clock gate control till both operands arrive and can be consumed. The conflict can thus be resolved and a feasible schedule attained, but latency and loop initiation interval increase somewhat as congestion increases.

3.3 Compiler Controlled Clock Gating

A distinguishing feature of the architecture is that a compiler can manage pipeline activity on a cycle by cycle basis. Microinstructions contain an opcode field for each function unit in the cluster. The fetch logic enables the pipeline shift and clock signals of a function unit only if the corresponding field is not a NOP. It can also generate a NOP when the opcode field is used for another purpose. The net result is that a function unit pipeline makes progress only during cycles when operations are issued to it and stalls by default. The scheme provides fine grain software control over clock gating while not requiring additional bits in the instruction to enable or disable a function unit. When the result of an N-cycle operation is required, but the function unit is not used after that operation, dummy instructions are inserted by the compiler into following instruction slots to flush out the required value. To avoid excessive power-line noise a compiler may keep a function unit active even when it has nothing to compute. The regular nature of modulo scheduled loops make them good candidates for analytical modeling and reduction of power-line noise [26].

Fine grain compiler directed pipeline control has two main purposes. Firstly, the compiler has explicit control over the

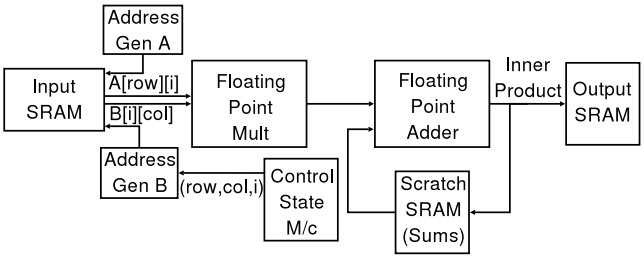

Figure 5: Inner Product Accelerator

life times of values held in a pipeline unlike a traditional architecture where values enter and exit the pipeline under hardware control and only quantities held in architected registers may be explicitly managed. Pipeline registers and the associated bypass paths may be managed as if they were a small register file and dataflows found in custom hardware can be easily mimicked. Secondly, it lets the compiler control the amount of activity within a cluster. Software control of dynamic energy consumption makes energy vs ILP tradeoffs possible. The resulting activity pattern is similar to the ideal condition where each function unit has its own clock domain and runs with just the right frequency.

4. PROGRAMMING EXAMPLE

This section illustrates the operation of the perception processor using a simple kernel which is mapped into microcode. The algorithm to multiply two 16×16 floating point matrices is shown in Figure 6. The control pattern consists of 3 level nested *for* loops. Assuming that the matrices are stored in row major order, the inner product computation will access array A along the row while B will be accessed along the column causing a base stride access pattern. The compute pattern consists of multiply accumulate operations which form the core of the inner product function.

Figure 5 outlines a simple custom hardware accelerator for this algorithm. Address generator A fetches the rows of matrix A. Address generator B generates the base stride pattern for the columns of matrix B. Corresponding rows and columns are fetched and applied to the floating point multiplier. The output of the multiplier is accumulated in a scratch register by the floating point adder. When an inner product sum is ready it is written to a result SRAM which is not shown in the figure.

In theory, this simple pipeline could compute one inner product every 16 cycles. However, the final accumulation of the inner product value creates a performance bottleneck. The floating point add takes 7 cycles and since the output is accumulated, a new multiply value can only be handled every 7 cycles. Hence inner products take 16×7 cycles. Interleaving the computation of 7 or more inner products relieves this bottleneck. The cost is: a) address generator B needs to be able to generate multiple interleaved base-stride patterns b) address generator A needs to hold each row element long enough for all the interleaved inner products and, c) Several scratch registers are required to hold the intermediate sums.

Efficient compilation for any architecture attempts to maximally utilize execution units while minimizing storage pressure. The same is true for our cluster architecture but the fine grain control provides more options. Figure 7 shows cleaned up assembly code for the interleaved inner product for the cluster architecture. For brevity the outer loop which invokes the interleaved inner product is not shown. This code is capable of sustaining the same throughput (7 inner products every 16×7 cycles) as the refined custom hardware accelerator. Performance and energy efficiency are achieved by a combination of techniques.

The inner product loop *i_loop* is marked for hardware modulo loop acceleration and its parameters are configured into a free context in the loop unit. Two address contexts *A_ri* and *B_ci* are allocated and the address generators attached to the input SRAM ports are reconfigured. Both contexts are tied to the loop *i_loop*. *B_ci* is set to generate a column walk indexed by *i_loop*, with the starting offset specified in a constant field in the load opcode. *A_ri* is set to access the matrix row by row in conjunction with an

```
def inner_product(A, B, row, col):
    sum = 0.0
    for i in range(0,16):
        sum = sum + A[row][i] * B[i][col]
    return sum

def matrix_multiply(A, B, C):
    # C is the result matrix
    for i in range(0, 16):
        for j in range(0, 16):
            C[i][j] = inner_product(A, B, i, j)
```

Figure 6: Matrix Multiply Algorithm

```
i_loop = LoopContext(start_count=0,
                     end_count=15,
                     increment=1, II=7 )
A_ri  = AddressContext(port=inq.a_port,
                       loop0=row_loop,
                       rowsize=16,
                       loop1=i_loop, base=0)
B_ic  = AddressContext(port=inq.b_port,
                       loop0=i_loop,
                       rowsize=16,
                       loop1=Constant,
                       base=256)

for i in LOOP(i_loop):
    t0 = LOAD( fpu0.a_reg, A_ri )
    for k in range(0,7): # Will be unrolled 7x
        AT(t0 + k)
        t1 = LOAD(fpu0.b_reg, B_ic,
                  loop1_constant=k)
        AT(t1)
        t2 = fpu0.mult( fpu0.a_reg,
                        fpu0.b_reg )
        AT(t2)
        t3 = TRANSFER( fpu1.b_reg, fpu0 )
        AT(t3)
        fpu1.add( fpu1, fpu1.b_reg )
```

Figure 7: Microcode for Interleaved Inner Product

outer loop. The address contexts effectively implement array variable renaming functions, a fact which is not evident in the code.

On entering *i_loop* the previous loop is pushed on a stack, though its counter value is still available for use by the address contexts, particularly *A_ri*. The new loop updates its counter every 7 cycles and admits new loop bodies into the pipeline. This is not a branch in a traditional sense and there is no branch penalty.

Communication is explicit and happens via load/store instructions or via interfunction unit data transfers both of which explicitly address pipeline registers. In the example $A[r][i]$ and $B[i][c]$ are allocated to pipeline registers $fpu0.a\_reg$ and $fpu0.b\_reg$ respectively. In fact, it is more appropriate to say that $B[i][c+k]$ where k refers to the k^{th} interleaved inner product resides in $fpu0.b\_reg$ at time $t0+k$. No scratch registers are required for the sum. The intermediate sums are merely circulated through the long latency fpu adder. This notion of allocating variables both in time and space is central to programming the perception processor.

The return value of each opcode mnemonic is the relative time at which its result is available. The *AT* pseudo op is a compile time directive that controls the relative time step in which following instructions are executed. Dataflow is arranged by referring to the producer of a value and the time step it is produced in. Such a reference will be translated by the compiler into commands for the forwarding logic. More complex programs are written as several independent execution streams. The streams are then made to rendezvous at a particular time by adjusting the starting time of each stream. The example shows that compile time pseudo ops can perform arithmetic on relative times to ensure correct data flow without the programmer needing to be aware of the latencies of the actual hardware implementation.

The loop body for *i_loop* will consist of 7 inner loop bodies created by loop unrolling. Each inner loop body before unrolling takes 18 cycles to execute. Since *i_loop* has been specified to have an initiation interval of 7 cycles, a total of 3 *i_loop* bodies corresponding to 21 of the original loop bodies will be in flight within the cluster at a time. It is the modulo aware nature of the address generators that permits each of these loop bodies to refer to array variables in a generic manner like $A[r][i]$ and get the reference that is appropriate for the value of r and i which were current at the time that loop body was started. Without special purpose address generation such high levels of ILP will not be possible. A previous version of the architecture without modulo address generators had limited ILP because generic function units and registers were used for address generation [18].

For this example, interleaving 7 inner products at a time results in 2 left over columns. They are handled by a similar loop to the one shown in Figure 7 except that it will have more idle slots. The adder needs to be active all the time, but the multiplier needs to work only 2 out of every 7 cycles. Since the multiplier pipeline will not shift 5 out of 7 cycles, the dynamic energy consumption resembles an ideal circuit where the adder runs at full frequency and the multiplier runs at 2/7 of the frequency thereby consuming less energy.

The overall effect is that the dataflow and throughput of the perception processor matches the custom hardware but in a more programmable manner.

5. EVALUATION

The perception processor will be compared against its competition using a set of ten benchmarks. Four different implementations are considered: a) Software running on a 400 MHz Intel XScale processor. The XScale represents a popular energy efficient embedded processor. b) Software running on a 2.4 GHz Intel Pentium 4 processor. While the Pentium is not as energy efficient as typical embedded processors, most energy efficient processors currently are unable to support real-time perception tasks. c) Micro-code implementations of benchmarks running on the perception processor. d) Custom ASICs were implemented for four benchmarks.

5.1 Benchmarks

A set of ten benchmarks are used in this study. Seven represent components of perception applications while three were added from the DSP and encryption domains to test the general applicability of the perception processor to other streaming problems. Two algorithms named GAU and HMM occupy about 99% of the execution of the CMU Sphinx 3.2 speech recognizer [12, 17]. Fleshtone is used for skin color detection and Erode and Dilate are used for image segmentation in a visual feature recognizer [16]. Rowley and Viola are face detectors based on neural network and wavelet based methods respectively [22, 23]. In this study these algorithms operate on 30×30 pixel grayscale image regions.

The FFT benchmark performs a 128 point complex to complex Fourier transform on floating point data. On the Pentium, we use the FFTW package which is believed to be the worlds fastest software FFT implementation. The microcode implementation running on the cluster is based on a simple radix 2 algorithm. The FIR benchmark implements a 32 tap finite impulse response filter. The encryption algorithm Rijndael is the AES standard. Its usage will be in the context of encrypting 576 byte Ethernet packets using a 128 bit key.

Parts of the GAU, Rowley and Fleshtone algorithms are amenable to vectorization while the rest are dominated by 2D array row and column accesses. HMM and Fleshtone both contain data dependent branches which have been if-converted while the remaining algorithms do not contain branches in the loop body. Graphs in this section will show floating point benchmarks (Rowley, GAU, FFT and Fleshtone) first. The remaining benchmarks use only integer arithmetic.

5.2 Metrics

Energy consumption per input packet and throughput are important metrics used in this study to compare energy efficient acceleration of algorithms. However they describe only half of the story because the energy vs delay tradeoff inherent in CMOS circuits makes it possible for low performance circuits to drastically reduce energy consumption. The energy delay product is a very useful metric in this context since it tracks both energy and performance improvements [6]. Since this metric depends on the feature size of the CMOS process, λ, designs need to be normalized to the same process for comparison. The perception processor and the Pentium 4 both use 0.13μ CMOS processes. Hence, no scaling is necessary to compare them. Since the XScale is implemented in a 0.18μ process and the ASICs are implemented in a 0.25μ process, to give the competition

the advantage of a better process, we assume constant field scaling and use λ^3, λ and λ^4 to normalize energy, delay and energy-delay product to a 0.13μ process [25]. It is important to note that below 0.9μ Vth and Vdd are unlikely to scale in order to limit leakage currents. This will increase the advantage of our fine grained cluster approach.

5.3 Experimental Method

Two configurations of the perception processor were designed for 1 GHz operation at 1.6 volts in a 0.13μ CMOS process. The floating point configuration has 4 FPUs and 4 ALUs while the integer configuration contains 4 ALUs and 2 integer multipliers. The configurations are otherwise identical. The Verilog netlists for each processor is synthesized, and a clock tree and wire loads are added. The circuit is then simulated at the transistor level using a commercial version of Spice (Synopsys Nanosim). Micro-code for the benchmarks are run on the processor during Spice level simulation. Numerical integration of the supply current waveform provided by Spice is used to compute energy consumption. The SRAMs used in the processor are macro-cells. Simulating them at the transistor level is not feasible. Reads, writes and idle cycles on each SRAM are logged during simulation and used to compute energy based on current consumption reported by the SRAM macro-cell generator tool. Benchmarks are run for several thousand cycles until the energy estimate converges.

Circuit boards of Pentium and XScale based systems were modified at the board level to measure processor power consumption. The benchmarks were then run on these systems and average power consumption was measured using a current probe and an oscilloscope. Comparison against a DSP would be nice, but no system suitable for such PCB modification is currently available to us.

Since the XScale does not have an FPU, the floating point benchmarks are compared against an *ideal* XScale whose FPU has the same latency and energy consumption as an integer ALU. For this purpose, floating point operators in the code are substituted with their integer counterparts. This change renders the results computed by the algorithm on the XScale meaningless. However, the performance and energy consumption represent a lower bound for any real XScale implementation with an FPU.

6. RESULTS

The design goal of the perception processor was to achieve high performance for perceptual algorithms at low power. For stream computations, a very important consideration is if a system has sufficient throughput to be able to process a known data rate in real-time. Since dynamic energy consumption is directly proportional to operating frequency, one method for achieving this goal is to exploit high levels of instruction level parallelism for stylized applications without paying a high price in terms of hardware complexity. This in turn permits adjusting the frequency and operating voltage to adequately meet real time requirements. More specifically, since dynamic power consumption is proportional to CV^2f, if high throughput can be achieved without a corresponding increase in C, then V and f may be scaled down to achieve significant energy savings.

Figure 8 shows the IPC of the perception processor compared against the IPC measured using hardware performance counters on an SGI R14K processor. The benchmarks were

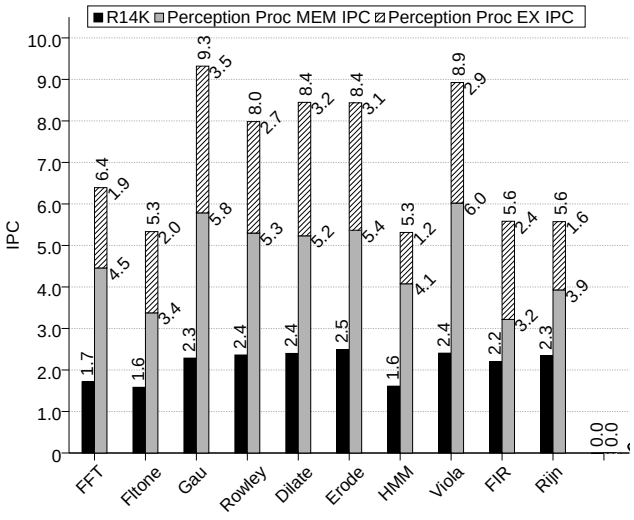

Figure 8: IPC

compiled for the R14K using the highly optimizing SGI MIPSPro compiler suite. The perception processor achieved a mean improvement in IPC of 3.3 times [1] over the sophisticated out of order processor. A large fraction of this improvement may be directly attributed to the memory system which can transfer data at a high rate into and out of the function units. This leads to high function unit utilization and high IPC.

This claim is further bolstered by Figure 9 which shows the throughput of the perception processor, the Pentium 4 and the XScale processors. Throughput is defined as the number of input packets processed per second and the results shown in Figure 9 are normalized to the throughput of the Pentium 4. The perception processor operating at 1 GHz outperforms the 2.4 GHz Pentium 4 by a factor of 1.75. The perception processor's mean throughput is 41.4% of that of the ASIC implementations (Gau, Rowley, FIR, Rijn). Bear in mind that this is severely skewed by the fact that the ASIC implementations, particularly Rijn expends vastly more hardware resources than the perception processor. The goal for the ASIC implementations has been to set an upper bound on performance without radical change to the base algorithm. For example in the case of the encryption benchmark Rijn, a single copy of the s-box lookup table is stored in the scratch memory of the perception processor. In contrast, the ASIC version uses 12 copies of the same lookup table in parallel to enhance performance. For the set Gau, Rowley and FIR, the perception processor in fact achieves on average 84.6% of the throughput of the ASIC implementation. This demonstrates the benefit of our architectural solution to the problems posed by perceptual algorithms.

Improving both energy and performance simultaneously is often quite difficult. Figure 10 shows that while delivering high throughput, the perception processor consumed on average 13.5 times less energy than the XScale embed-

[1] In this paper the terms mean and average refer to the geometric mean. All graphs except Figure 8 use log scale for the Y axis.

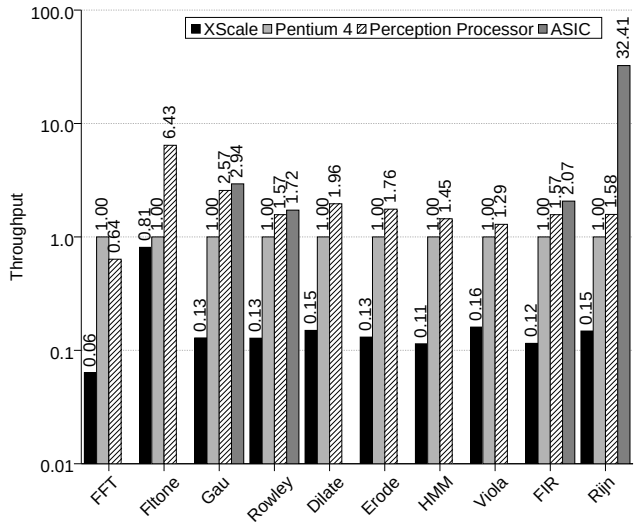

Figure 9: Throughput Normalized to Pentium 4 Throughput

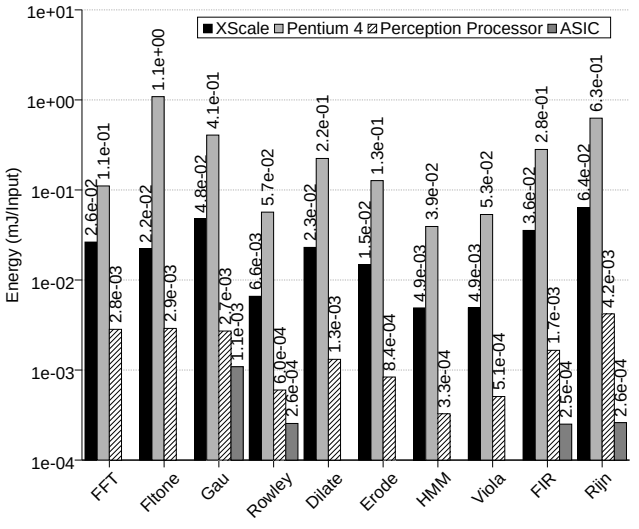

Figure 10: Process Normalized Energy Consumption

ded processor. In terms of energy delay product, Figure 11 shows that the perception processor outperforms the XScale processor by a factor of 159 and the Pentium 4 by more than 3 orders of magnitude. Note that these two graphs use a log scale. When compared to the ASIC implementations the perception processor is worse merely by a factor of 12. This again shows that the perception processor is able to retain a large amount of generality while paying a relatively small penalty in energy delay product.

All together these radical improvements suggest that in cases where high performance, low design time and low energy consumption need to be addressed simultaneously, the perception processor could be an attractive alternative.

Figure 12 shows the synergistic effect of applying clock gating to a cluster that supports compiler controlled datapaths. Compiler controlled datapaths provide energy reduction by decreasing datapath activity, and avoiding register files accesses. To implement it, the load enable signal of each pipeline register should be controlled by software. Once the design is adapted for explicit pipeline register enabling, it is a trivial extension to clock gate pipeline registers using the same signal. This graph shows that on average this saves 39.5% power when compared to the implementation without clock gating. These results are aliased by two factors a) SRAM power adds a large constant factor to both the cases and, b) our CAD tools are unable to clock gate multicycle datapaths like the FPUs. Further reduction is possible by clock gating multicycle datapaths.

7. RELATED WORK

Scheduling techniques for power-efficient embedded processors have achieved reasonably low power operation but they have not achieved the energy delay efficiency of the work described here [10]. Reconfigurability using FPGA devices and hybrid approaches have been explored [2, 5]. These approaches offer generality but not at a performance level that can support perception applications. Of particular relevance are compiler directed approaches which are similar to that described here. The primary difference is that this approach targets custom silicon rather than FPGA devices [19]. Customizing function units in a VLIW architecture has been studied and the Tensilica Xtensa is a commercial instance of this approach [7].

The RAW machine has demonstrated the advantages of low level scheduling of data movement and processing in function units spread over a 2 dimensional space [24]. Imagine has demonstrated that significant performance gains can be attained when appropriate storage resources surround execution units [21]. Given the poor wire scaling properties of deep sub-micron CMOS processes, it is somewhat inevitable that function unit clusters will need to be considered in order to manage communication delays in high performance wide-issue super-scalar processors. However, these approaches are all focused on providing increased performance. The approach here is somewhat similar but is tuned to optimize energy while providing just enough performance to meet the real time guarantees of sophisticated perception applications.

The MOVE family of architectures explored the concept of transport triggering where computation is done by transferring values to the operand registers of a function unit and starting an operation implicitly via a move targeting a trigger register associated with the function unit [11]. Like in the MOVE architecture, the concept of compiler directed data transfer between function units is used in this paper too, but the resultant architecture is a traditional operation triggered one and transport triggering is not used.

Clock power is often the largest energy culprit in a modern microprocessor [8]. While there has been significant research into exposing clock gating to the compiler, this paper is perhaps unique in using compiler controlled clock gating not only as a power reduction mechanism, but also as a means to control life time of variables and ensure improved data flow.

Harnessing data parallelism using short vector or SIMD

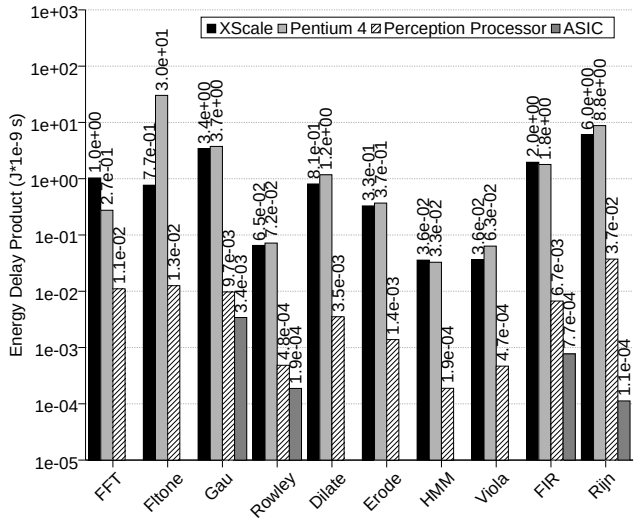

Figure 11: Process Normalized Energy Delay Product

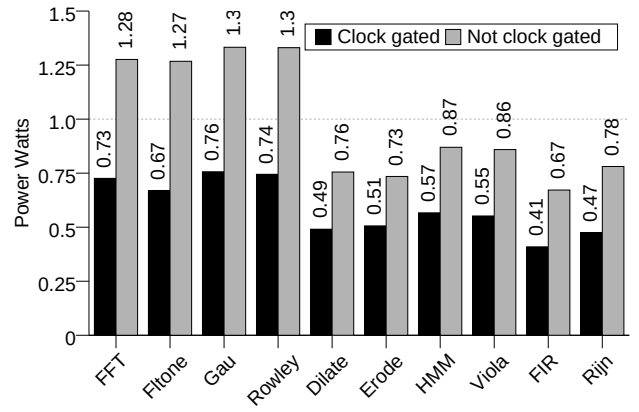

Figure 12: Impact of Clock Gating

architectures is a popular approach [20, 1]. This technique has been shown to improve performance by up to an order of magnitude on DSP style algorithms and even on some small speech processing codes [13]. The perception processor is capable of capitalizing on this form of data parallelism as well. From an energy delay perspective however, SIMD operation is orthogonal to the compiler directed data flow and clock gating approaches described in this paper. Therefore, we have not pursued this option.

8. CONCLUSIONS

The perception processor uses a combination of VLIW execution clusters, compiler directed dataflow and clock gating, hardware support for modulo scheduling and special purpose address generators to achieve high performance at low power for perception algorithms. It outperforms the throughput of a Pentium 4 by 1.75 times with an energy delay product that is 159 times better than an XScale embedded processor. This approach has a number of advantages: a) its energy-delay efficiency is close to what can be achieved by a custom ASIC; b) the design cycle is extremely short when compared to an ASIC; c) it retains a large amount of generality compared to an ASIC; d) it is well suited for rapid automated generation of domain specific processors. We have shown that fine-grained management of communication and storage resources can improve performance and reduce energy consumption whereas simultaneously improving on both these axes using a traditional microprocessor approach has been problematic. Of similar importance is that sophisticated real-time perception applications can be adequately supported on this architecture within an energy budget that is commensurate with the embedded space.

9. REFERENCES

[1] D. Brash. The ARM Archtecture Version 6 (ARMv6). ARM Holdings plc Whitepaper, January 2002.

[2] T. Callahan and J. Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, CA, 2000. ACM.

[3] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu. New paradigm of predictive MOSFET and interconnect modeling for early circuit design. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, pages 201–204, June 2000.

[4] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu. Predictive technology model. http://www-device.eecs.berkeley.edu/˜ptm, 2002.

[5] A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In D. A. Buell and K. L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[6] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[7] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, March 2000.

[8] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Design Automation Conference*, pages 726–731, 1998.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.

[10] J. Hoogerbrugge and L. Augusteijn. Instruction scheduling for TriMedia. *Journal of Instruction-Level Parallelism, 1(1)*, Feb. 1999.

[11] J. Hoogerbrugge, H. Corporaal, and H. Mulder. MOVE: a framework for high-performance processor design. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 692–701. ACM Press, 1991.

[12] X. Huang, F. Alleva, H.-W. Hon, M.-Y. Hwang, K.-F. Lee, and R. Rosenfeld. The SPHINX-II speech recognition system: an overview. *Computer Speech and Language*, 7(2):137–148, 1993.

[13] S. M. Joshi. Some fast speech processing algorithms using Altivec technology. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 2135 – 2138, Mar. 1999.

[14] B. Mathew. *The Perception Processor*. PhD thesis, School of Computing, University of Utah, Aug. 2004.

[15] B. Mathew and A. Davis. A Loop Accelerator for Low Power Embedded VLIW Processors. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004.

[16] B. Mathew, A. Davis, and R. Evans. A characterization of visual feature recognition. In *Proceedings of the IEEE 6th Annual Workshop on Workload Characterization (WWC-6)*, pages 3–11, October 2003.

[17] B. Mathew, A. Davis, and Z. Fang. A low-power accelerator for the Sphinx 3 speech recognition system. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03)*, pages 210–219, October 2003.

[18] B. Mathew, A. Davis, and A. Ibrahim. Perception coprocessors for embedded systems. In *Proceedings of the Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 109–116, October 2003.

[19] S. O. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzade. A super-scheduler for embedded reconfigurable systems. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, page 391, Nov. 2001.

[20] H. Nguyen and L. K. John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In *International Conference on Supercomputing*, pages 11–20, 1999.

[21] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pages 3–13, Nov. 1998.

[22] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):23–38, 1998.

[23] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Dec. 2001.

[24] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

[25] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, A Systems Perspective*. Addison Wesley, second edition, 1993.

[26] H.-S. Yun and J. Kim. Power-aware modulo scheduling for high-performance vliw processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 40–45. ACM Press, 2001.