A Cluster 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 can not be met by current embedded processors and often even exceed the performance of high performance microprocessors with an energy budget that is infeasible in the embedded space. The normal approach is to resort to a custom ASIC in order to meet performance and energy constraints. However ASICs incur expensive and lengthy design cycles. They are so specialized that they are unable to support multiple applications or even evolutionary improvements in a single application. This paper introduces a VLIW perception processor which uses a combination of clustered function units, compiler controlled data-flow and compiler controlled clock-gating in conjunction with hardware support for modulo scheduling, address generation units and a scratch-pad memory system to achieve very 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. We use DSP and encryption algorithms to demonstrate that the perception processor is general enough to be applied to other streaming problems. Since energy and delay are common design tradeoffs, the energy-delay product of a CMOS implementation of this architecture is compared against ASICs and a general purpose processors. Using a combination of Spice simulations, real processor power measurements and architecture simulation 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 delivering this performance it simultaneously achieves 135 times better energy delay product than a low power Intel XScale embedded processor.

1. INTRODUCTION

Embedded computing requirements have both escalated and diversified as mobile computing, ubiquitous computing, and traditional embedded applications continue to converge. If the dream of ubiquitous computing is to become both useful and real, the computing embedded in all aspects of our environment must be accessible via natural human interfaces. Future embedded environments need to at least support sophisticated applications such as speech recognition, visual feature recognition, secure wireless networking, and general media processing in mobile embedded platforms. The problem is that these applications 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 floating point units 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 expended in high performance processors is intractable in the embedded space. Furthermore, the power requirements of new processors is increasing. The conclusion is that technology scaling alone can not solve this problem. Second, perception and security interfaces are by nature always operational. 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. The initial focus of this work was to investigate ASIC architectures that would meet these constraints. This path led to the usual realization that ASICs are costly and inflexible. The inherent level of specialization in an ASIC makes it extremely difficult for the implementation 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 provide significant motivation to explore a more general purpose approach. The use of FPGA devices is another common approach [12]. The inherent reconfigurability of FPGAs provides a level of specialization while retaining significant generality. However the reconfiguration time is relatively long, and FPGAs suffer both in performance and power when compared to either ASIC or CPU logic functions.

Our research addresses the rapid automated generation of low-power high performance VLIW processors for the perception domain. The domain optimized processor is intended to be used as a coprocessor for a general purpose host processor. 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

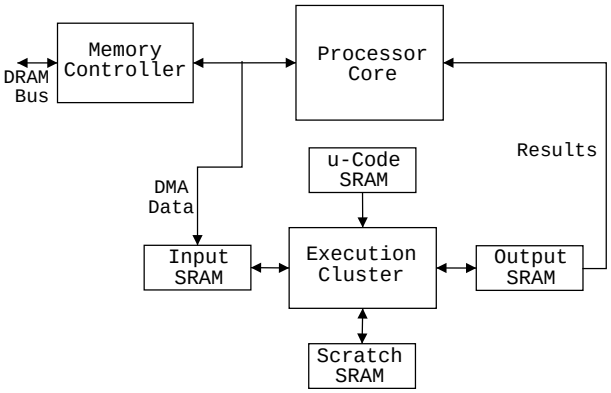

Figure 1: Coprocessor Organization

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 cluster generator. Henceforth we will use the terms perception processor and cluster interchangeably to refer to the generic architecture behind any domain specific processor created using the cluster generator tool.

The approach includes a specialized compiler which maps applications onto the perception processor. Any algorithm can be compiled into a micro-code representation, albeit with varying levels of efficiency. The compiler uses hardware support for modulo-scheduled loops [38] in conjunction with array address generators to deliver high throughput for flow dependent loops. The microcode provides fine-grained control over data steering, clock gating, function unit utilization and it permits single cycle reconfiguration of address generators. 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 data-flow 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 data-flows or *active datapaths* resemble custom computational pipelines found in ASICS, but have the advantage of flexibility offered by software control. One could think of this as a means of exploiting the natural register renaming that occurs when a multi-stage pipeline shifts and each individual pipeline register gets a new value. However the active data-path 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, such as speech recognition, 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 encryption and DSP domains to test generality of the approach outside of the perception domain. All of the benchmarks operate in what can be called *stream mode*. Stream mode applications take in a frame of data and operate on it to produce local state information and an output frame before moving on to the next input frame. The size of the local state information is usually small as is the output frame. This characteristic motivates the use of input, output, and scratch-pad SRAMs. The stream rate of perception applications has an intrinsic real time requirement since processing needs to keep up with continuously arriving auditory or visual data. This stream based computation model also applies to a wide range of important applications such as media encode/decode, security encryption/decryption, compression and decompression schemes, etc.

The perception processor is evaluated using Spice simulations of a 0.13μ CMOS implementation operating at 1 GHz. Its effectiveness is compared against the obvious competition: general purpose processors and ASICs. Comparison against conventional processors is problematic because energy efficient embedded 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.

Since energy consumption and performance are fundamental design trade-offs the main comparison in this paper is based on the metric known as energy delay product advocated by Gonzalez and Horowitz [16].

2. CLUSTER 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 for reasons discussed shortly.

In general, cluster configurations are intended to have up to 8 function units. This limit is imposed by a target operational frequency of 300 MHz in a 0.25μ CMOS process. This frequency matches that of our host processor which has an instruction set similar to a MIPS R4600. At 300 MHz this host processor performs similarly to the well-known Intel StrongArm. Increasing the number of functional units will reduce the operation frequency primarily due to delays associated with wider multiplexers. Though all of our de-

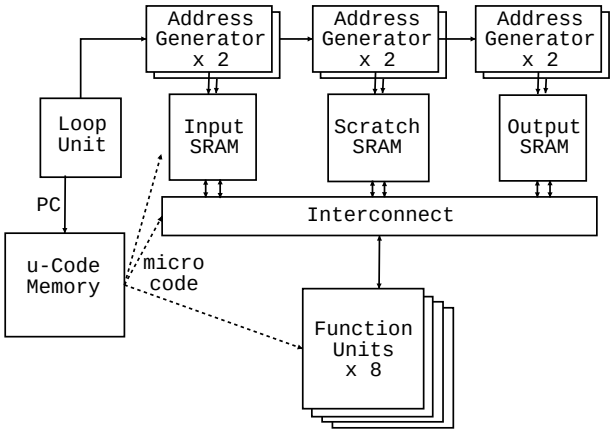

Figure 2: Cluster Architecture

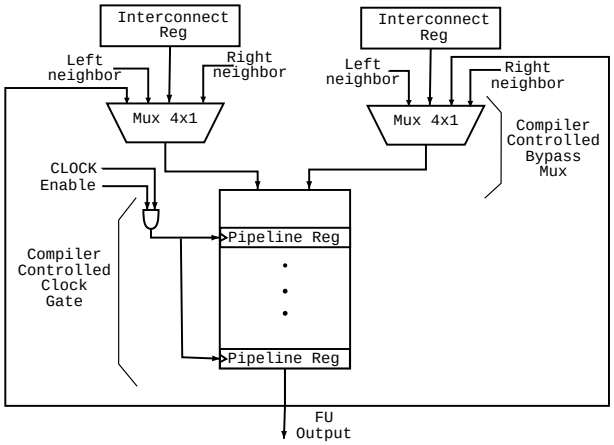

Figure 3: Function Unit Architecture

signs were originally done in a 2.5 volt 0.25μ CMOS process they were subsequently shrunk to work at 1 GHz in a 0.13μ process [9, 10]. This makes it possible to directly compare the perception processor against the Pentium IV which is fabricated using 0.13μ technology.

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.

2.1 Function Units

Function units follow the generic organization shown in Figure 3. 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.

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 [31]. 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.

2.2 Interconnect

As CMOS technology scales, wire delays get 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, inter-cluster 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 data-flow 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. We use the term *resource borrowing* to describe this approach.

2.3 The Loop Unit

While the perception processor supports regular branch instructions, it provides special acceleration for counted loops. In fact, loops are a resource that can be allocated and managed just like one would allocate memory on a traditional architecture. A loop context is a data structure that abstracts the properties of a loop. Important parameters include the loop count, the increment amount, whether the loop is a regular loop or a modulo loop, and the loop initiation interval for modulo loops. The loop unit holds 4 loop contexts at a time in addition to the micro-program counter.

When a loop context is allocated, its parameters are configured into the loop unit in a single cycle. The real purpose of loop allocation is two fold: a) to periodically admit new loop bodies into the datapath b) to enable *array variable renaming*, a powerful replacement for register renaming/rotation. The loop unit maintains a counter for each loop context and updates it periodically. It also modifies the program counter and admits new loop bodies into the pipeline in the case of modulo loops. In the latter case it also does additional manipulation of the loop counter to drain the pipeline correctly on loop termination. The application may have any number of loops, but hardware managed loops cannot be nested more than 4 deep. On entering a new loop any previous loop is pushed on a stack, though its counter value is still available for use by address generators. Loop contexts may be loaded from memory. This permits modulo scheduling a loop whose loop count is not known at compile time and loading appropriate loop parameters from SRAM depending on the size of input data.

2.4 Address Generators

Most perception algorithms have a high ratio of array variable accesses to operators. Multiple SRAM ports are essential for high throughput. Since each additional SRAM port adds significant area and energy overhead, utilizing them effectively is essential for performance. The 3 dual ported SRAMs together have a read/write power consumption approximately equal to the total function unit power consumption. A previous version of the architecture which used generic integer ALUs for address generation was found to be unable to maximize SRAM port utilization [32]. To improve the situation, dedicated address generators are attached to each SRAM port. They handle commonly occurring address sequences like vector and strided access as well as 2D array accesses including row and column walks. They can also handle address generation under regular, modulo and unrolled loops and can handle special situations that occur when multiple loop bodies are in flight simultaneously. Each address generator has a designated partner ALU in the cluster with several address generators possibly sharing the same partner. In cases where the address generator does not know how to compute the array index function, it is possible to directly issue an address computed by its partner ALU. The partner ALU can also compute address contexts on the fly and reconfigure an address generator. The combination of an address generator and its partner ALU can also effectively deal with indirect access streams of the type $A[B[i]]$. Address generation adds 1 cycle latency to load/store operations.

An array access pattern is abstracted by a data structure known as an address context. Currently, each address generator can hold 4 contexts. Since the current cluster has 6 address generators a total of 24 array variables may be accelerated at one time and any number of variables may be accessed in non-accelerated mode. Each address generator deals with a variety of access patterns merely by computing the quantity:
$B = Mux(B\_loop, constant, alu\_output)$
$address = (((A\_loop - mp) << x)|(B << y)) + base.$
Shift constants, base address, mux select, etc. are encoded in the address context. Mp is an instruction tag inserted by the compiler into the opcode. A_loop and B_loop can select any of the loop counters maintained by the loop unit. The loop unit has a counter which always contains zero, to be used as a default in cases where only one dimensional access is required.

Every load/store operation specifies a source/destination pipeline register and an address context. The targeted address generator will examine the address context, retrieve any loop counters the array reference depends on, compute the address and transfer the value to the correct pipeline register over the interconnect. When multiple loop bodies are in flight, the Mp tag inserted in the load/store opcode may be cross checked with internal counters and initiation interval information to automatically correct the address. This last step performs the equivalent of register renaming on array variables.

In a conventional VLIW processor, data is loaded from memory to the register file and register rotation is used to perform renaming. In a previous version of the architecture we found that using a 32 entry register file limited the length of vectors and thereby the ILP we could obtain. Also, register rotation as in the case if the IA-64 provides only a single rotating window into which all variables must be loaded. Ideally multiple rotating windows are required. The problem is solved by rotating the array variable itself. For example, the IA-64 may load two array references $A[i][j]$ and $B[i][k]$ into a block of consecutive registers say $r96 - r112$ and rotate the register set so that when the next loop body is admitted, $r96-r112$ points to new physical registers while the previous loop body can continue execution with the old binding. In contrast, with **array variable renaming** our hardware changes the address corresponding to $A[i][j]$ and $B[i][k]$. In effect all of the local SRAM may be used as rotating registers with an independent rotating window for each variable. Array variable rotation not only overcomes ILP limits caused by the limited size of the register file, it obviates the need for a register file. Register files with a large number of ports have been shown to be a major consumer of energy [21]. We believe that array variable rotation is a novel feature of this architecture.

2.5 Compiler Controlled Clock Gating

A distinguishing feature of the architecture is that a compiler can manage pipeline activity on a cycle by cycle basis. Micro-instructions 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-

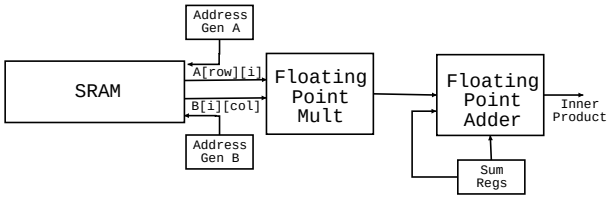

Figure 4: Inner Product Accelerator

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 [50].

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

3. 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 5. 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 pattern.

Figure 4 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 pipeline problem. 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

```
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 5: Matrix Multiply Algorithm

scratch registers are required to hold the intermediate sums.

Compilers for high performance architectures attempt to approximate the dataflow in the custom accelerator. In vector processors, vector chaining creates a similar data flow and reduction operators help alleviate some of the performance penalty caused by the floating point accumulate operation. By selecting independent adds and multiplies which are ready for issue from its instruction window an out of order processor will work somewhat like a vector processor that can be time sliced across several interleaved vectors. In addition, a combination of software pipelining and branch prediction ensures that the pipeline has as few wasted cycles as possible. Address generation will be handled by generic ALUs which send computed addresses to available load/store ports. Some form of register renaming will be required to enable software pipelining to work well in non-trivial kernels.

```
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 6: Assembly code for interleaved inner product

Figure 6 shows cleaned up assembly code for the interleaved inner product for the cluster architecture. For brevity the outer loops which invoke the interleaved inner product are 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 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 inter-function 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 [32].

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 6 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 seven 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.

4. EVALUATION

The benefit of this approach is tested on ten benchmarks that were chosen both for their importance in future embedded systems as well as for their algorithmic variety. In order to compare our approach to the the competition, four different implementations of each benchmark are considered:

1. Software running on a 400 MHz Intel XScale (StrongARM) processor. The XScale represents a highly energy efficient embedded processor.

2. Software running on a 2.4 GHz Intel Pentium 4 processor. We note that the Pentium 4 is not optimized for energy efficiency but more efficient processors can not currently support real-time perception tasks such as speech recognition.

3. A micro-code implementation running on the perception processor.

4. Half the benchmarks are compared a custom ASIC implementation.

4.1 Benchmarks

The first two algorithms called GAU and HMM are dominant components of several speech recognizers. The next five algorithms named Rowley, Fleshtone, Erode, Dilate and Viola are components of visual feature recognition systems. The last three algorithms are FFT, FIR and Rijndael and these represent the DSP and encryption domains. The DSP algorithms were added to test the generality of our approach. Rowley, GAU, FFT and Fleshtone are floating point intensive. The remaining benchmarks are integer only computations. Some components of GAU, Rowley and Fleshtone may be vectorized while the rest of the algorithms cannot. HMM is intensive in data dependent branches which may be if-converted. The others with the exception of Fleshtone are loop oriented.

Several source level optimizations have been made to the software versions that run on the Pentium and XScale to boost their performance as much as possible [31]. The only exception is that no SIMD optimizations were made in order to keep the comparison fair. The perception processor could use SIMD floating point units, just like SSE on the Pentium. But widening datapaths contributes to a disproportionate increase in energy consumption and also makes isolating the impact of architectural options like compiler controlled dataflow impossible.

GAU and **HMM** represent Gaussian probability density evaluation and hidden Markov model evaluation respectively. GAU occupies 57.5% and HMM consumes 41.5% of the execution time of Sphinx 3.2, a leading research speech recognition system from CMU. Both Gaussian distributions

and Hidden Markov models are components of most mature speech recognizers [26, 49, 43]. GAU computes how closely a 10ms frame of speech matches a known Gaussian probability distribution. One input packet corresponds to evaluating a single acoustic model state over 10 frames of a speech signal. A real time recognizer needs to process 600,000 invocations of the GAU algorithm every second. The HMM algorithm performs a Viterbi search over a hidden Markov model corresponding to one model state. One input packet to the HMM implementation consists of 32 five-state Hidden Markov Models. While the GAU algorithm is entirely floating point, the HMM algorithm is dominated by integer compare and select operations. Its average rate of invocation varies significantly with context.

Rowley represents a neural network based visual feature detector [40]. In the face recognizer a multi-layer neural network is swept over 30x30 rectangular regions of an image. Each individual neuron is evaluated by the function $tanh(\Sigma_{i=1}^{n} Weight_i \times Image[Connection_i])$. Neurons have multiple sizes for their fan-in (n) and each layer depends on the preceding layer's output. The software implementations of the neuron evaluations have been hand unrolled and special versions were created for each input size. Also $tanh()$ has been implemented via table lookup which boosted the Pentium's performance almost by a factor of 2.5. A 30x30 image as well as the outputs of all the neurons are maintained within the cluster. Depending on the sizes of the neurons an input packet consisting of the weights and connections of 7 to 64 neurons is streamed through the cluster. All computations involve single precision floating point numbers.

Fleshtone represents a skin toning algorithm typically used as a preprocessing step to find skin colored regions of an image so that a more sophisticated object detector like the Rowley detector may be applied to it. The algorithm we have implemented converts RGB pixels to another color space and checks if the projected pixel falls in between two parabolic curves [42]. This algorithm represents a case that is difficult to vectorize since there are far more floating point operators per pixel than the number of FPUs present in the cluster. This necessitates multiple passes and saving of intermediate results. It also contains multiple if statements in the body. Each input packet consists of a single raster line of a 320x200 24-bit color image. The output is a 320 entry bitmap with bits set where flesh color is found.

Erode and **Dilate** represent two operators from mathematical morphology that help in image segmentation. Erode sweeps a 3x3 pixel filter over the bitmap produced by Fleshtone and cuts away weakly connected regions, i.e. it blacks out pixels if all pixels within the filter are not set. Dilate does the opposite, it sweeps a 5x5 pixel filter over a bitmap and fills in pixels if any of the pixels are set. Fleshtone, Erode and Dilate are used for image segmentation in a visual feature recognition system [30]. Erode works on 3 raster lines and dilate on 5 raster lines of a 320x200 image.

Viola is a reimplementation of the Viola and Jones' method of object detection based on a well known machine learning algorithm known as AdaBoost [46]. The algorithm relies on computing features or wavelets which are the weighted sum or difference of rectangular regions within a 30x30 window into an image. We maintain the coordinate and weight information for 100 features within the cluster and each input packet contains a 30x30 pixel image. The output contains the evaluation of all 100 features over the 30x30 image.

FFT implements a 128 point complex to complex Fourier transform on floating point data. The Fourier coefficients are maintained within the cluster. Input and output packets are 128 complex numbers where each complex number consists of 2 single precision floating point numbers. FFT represents a common algorithm for which many DSP processors implement ISA extensions. FFT also represents a case that causes bad interconnect conflicts on our architecture. Good performance depends on the resource borrowing technique described in Section 2.2. The software version on the Pentium is based on FFTW, a highly tuned FFT implementation which used dynamic programming techniques to adapt itself to the processor architecture [15]. Our cluster implementation on the other hand uses a simple radix 2 algorithm and no ISA extensions. Since FFTW cannot be used on the XScale, we use the simple radix 2 algorithm instead.

FIR is a 32 tap finite impulse response filter, a common primitive in DSP applications. Impulse response coefficients are maintained inside the cluster. Input packets of various sizes may be applied to the filter which successively evaluates each input and outputs one integer corresponding to every input word.

Rijndael is the AES encryption standard. Our implementation uses 128 bit keys and works on 16 byte blocks [11]. To simulate network level encryption of IP packets, the input packet is 576 bytes, the rfc894 recommended MTU for Ethernet. The key as well as the encryption S-boxes are maintained within the cluster.

4.2 Metrics

The tradeoff between energy consumption and performance is a common modern design choice. Increasing performance almost always involves increasing the energy requirements. As a result, it is misleading to compare solely on the basis of either energy or performance. This dilemma is even more meaningful for the real-time embedded perception applications that are the driving force for this work. The ability to process faster than real-time simply means that power is being wasted. Therefore a common tactic in such cases is to either reduce clock frequency, supply voltage, or both. The fine grain scheduling capability of the perception processor also allows scheduling of work rate which is a more intuitive mechanism and achieves results similar to clock frequency scaling.

An attractive and intuitive metric is to compare designs based on the energy expended to perform work at some rate [8]. Gonzalez and Horowitz showed that a more relevant comparison of architectural merit should be based on the rate of work per energy or an energy delay product [16]. Both architecture and semiconductor process influence the energy delay product. Since the feature size of the process, λ, has such a large impact it is necessary to normalize any design comparison to the same process. Under ideal scaling conditions, Meng [18] argues that the energy delay product will scale as λ^4. Since the threshold voltage rarely scales ideally, a more reasonable energy delay scaling model lies between λ^2 and λ^3 [16].

For the comparisons reported here, we have chosen a λ^3 basis, where performance and energy are each scaled by λ^2 and λ respectively. The perception processor and the Pentium 4 are both implemented in 0.13μ CMOS technology

and their results need not be normalized. The XScale and the custom ASICs are implemented using 0.18μ and 0.25μ technologies respectively, and their results are normalized using this method to a 0.13μ technology. It could be argued that as λ drops below 100 nanometers the rapid rise in leakage power is not adequately addressed. These leaky new processes are not well suited to the stringent energy constraints of embedded devices and the choice of a λ^3 is appropriate given the embedded perception focus of this work.

4.3 Experimental Method

This evaluation is based on generating hardware for two different perception processor configurations which we will henceforth refer to as the integer and floating point clusters. The integer cluster consists of 4 ALUs, 2 multiply units, and the remaining two slots are unused. The floating point cluster contains 4 ALUs and 4 FPUs. All of the integer benchmarks except FIR and VIOLA would run equally well on the floating point cluster. FIR and VIOLA both require integer multiply operations. The hardware for each cluster configuration (the entire organization shown in Figure 2) is generated. The input and scratch SRAMs are sized at 8KB and the output SRAM is 2KB. The design is simulated at the transistor level using Spice while running the microcode for the benchmarks. The Spice simulation provides a supply current waveform with one sample per 100 pico seconds. We use this information along with the supply voltage to compute instantaneous power consumption and then do numerical integration of power over time to compute energy consumption.

The dual-ported SRAMs are macro-cells generated by the CAD suite and simulating the entire SRAM array using Spice is not feasible. For the SRAMs we therefore log each read, write and idle cycle and compute the energy consumption based on the read, write and idle current reported by the SRAM generator. Each benchmark is run for several thousand cycles until the energy estimate converges. The host processor is not simulated.

The function units are described in Verilog and Synopsys MCL hardware description languages. The overall cluster organization and interconnection between function units is automatically generated by the compiler. The whole design is then synthesized to the gate level and a clock tree is generated. The net list is then annotated with worst case RC wire loads assuming all routing happened on the lowest metal layer. The energy measurements are therefore pessimistic and represent a worst case bound for each design. Exact measurements are extremely sensitive to wire routing decisions and as a result we calculate our wire capacitance based on the worst-case wiring layer. The micro-code corresponding to the benchmark is loaded into program memory and the Spice model is simulated in NanoSim, a commercial VLSI tool with Spice-like accuracy. The circuits were originally designed for a 0.25μ CMOS process and simulated using transistor models and CMOS process parameters measured for a test chip built in the same technology. Subsequently, we shrunk the same circuit to a 0.13μ technology [9, 10] and found that the λ^3 scaling model for energy-delay product was an excellent match to map simulation results from the 0.25μ to the 0.13μ technology. As a result, we provide only the 0.13μ results here.

The software version of each benchmark is compiled with the GNU GCC compiler and run on a 2.4 GHz Intel Pentium 4 processor. This system has been modified at the board level to permit measuring average current consumed by the processor module using a digital oscilloscope and non-intrusive current probe. We ensure that the input data always hits in the L1 Cache so that the L2 Cache and memory system effects are isolated as much as possible and the measurement thus represents core power. For the XScale system a similar approach is used except that we use software control to turn off unnecessary activity and measure the difference between the quiescent state and the computation. This method could slightly inflate the processor power, but measuring the core power alone is not technically feasible on this system due to packaging constraints. The choice of both systems were based on the technical feasibility of PCB modifications to permit measuring energy consumption.

Embedded processors like the StrongARM do not have floating point instructions that are required for some of the benchmarks. Software emulated floating point will bloat the energy delay product of the StrongARM and make a meaningful comparison impossible. We therefore compare against an *ideal* StrongARM which has FPUs which have the same latency and energy consumption as an integer ALU. This is done by replacing each floating point operator in the code with a corresponding integer operator. The computed results are meaningless, but the performance and energy consumption represent a lower bound for any real implementation with FPUs. This puts the cluster results in a more pessimistic position than in reality.

5. 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 the 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 a paying a high price in terms of hardware complexity. This in turn permits adjusting the frequency and operating voltage to be just enough to 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 7 shows the IPC of the perception processor compared against the IPC measured using native performance counters on an SGI R14K processor. The benchmarks were 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 (geometric mean) over the sophisticated super-scalar 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. The results clearly demonstrate that the design goal of high throughput through ILP has been achieved.

This claim is further bolstered by Figure 8 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 8 are normalized to the throughput

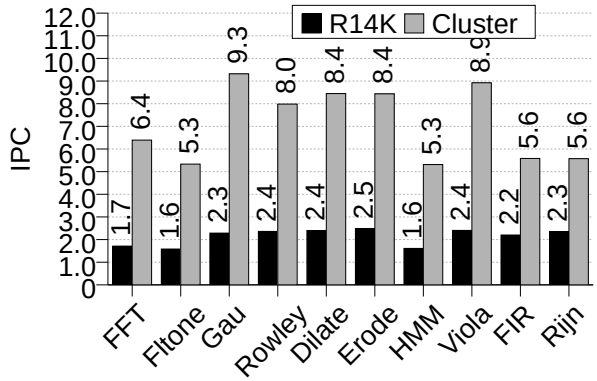

Figure 7: IPC

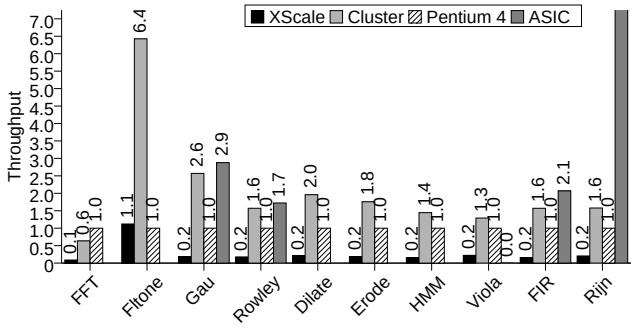

Figure 8: Throughput normalized to Pentium 4 throughput

of the Pentium 4. The perception processor operating at 1GHz outperforms the 2.4 GHz Pentium 4 by a factor of 1.75 (Geometric Mean). The perception processor's mean throughput is 41.6% 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. For the set Gau, Rowley and FIR, the perception processor in fact achieves on average 85.5% of the throughput of the ASIC implementation. These results clearly demonstrate the benefit of our architectural solution to the problems posed by perceptual algorithms.

Improving both energy and performance simultaneously is often quite difficult. Figure 9 shows that while delivering high throughput, the perception processor consumed 15.9 times (Geometric Mean) less energy than the XScale embedded processor. In terms of energy delay product, Figure 10 shows that the perception processor outperforms the XScale processor by a factor of 135 (Geometric Mean) 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 by a factor of 7.4. Its energy delay product is just a factor of two larger than the ASIC considering the Gau, Rowley and FIR implementations alone. 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

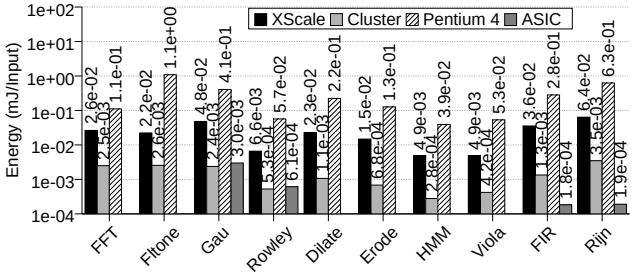

Figure 9: Process Normalized Energy Consumption

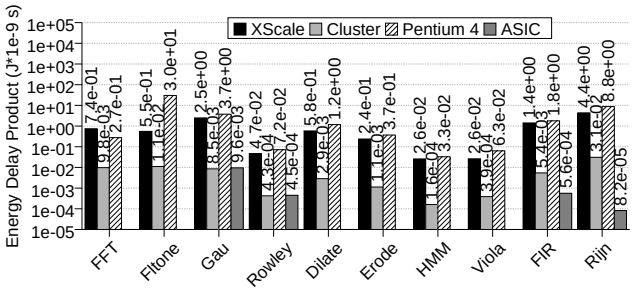

Figure 10: Process Normalized Energy Delay Product

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 11 shows the synergistic effect of applying clock gating to a cluster that supports compiler controlled datapaths. Compiler controlled datapaths provide energy reduction by providing 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 to clock gate pipeline reg-

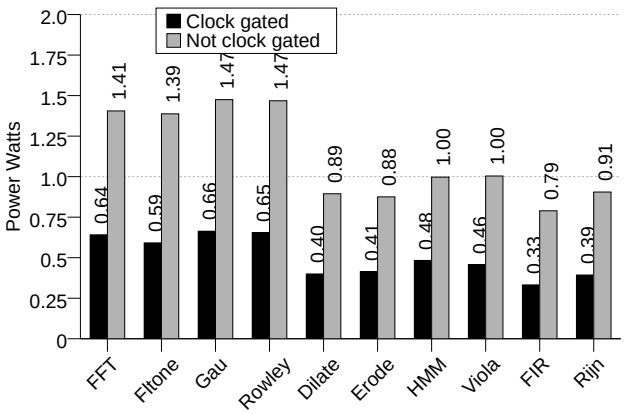

Figure 11: Impact of Clock Gating

isters using the same signal. It is seen in the graph that on average this saves 55.1% 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 multi-cycle datapaths like the FPUs. Further reduction is possible by clock gating multi-cycle datapaths.

6. 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 [20]. Reconfigurability using FPGA devices and hybrid approaches have been explored [7, 12]. 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 [33]. Customizing function units in a VLIW architecture has been studied and the Tensilica Xtensa is a commercial instance of this approach [17].

Memory power is a concern in embedded systems [35]. Given the high level of utilization, register file power is a particular concern [25] and numerous methods to improve on baseline register file have been investigated [21, 3, 45]. The method described here saves register file power quite simply by not needing one. The use of memory hierarchies to save power has been explored by Panda [36] and the use of scratch-pad memories has been shown to be effective [4]. This work employs similar strategies.

Clock power is often the largest energy culprit in a modern microprocessor [19]. Krashinsky describes the benefits of clock gating [24]. There are two disadvantages of clock gating: the enable signal must arrive sufficiently ahead of the clock signal, and the use of additional gates in the signal path will increase clock skew. Both effects reduce the maximum achievable clock frequency. For low-power designs, this is seldom a serious issue.

Increasing performance via VLIW techniques is a common theme in modern embedded systems [5, 2] including mapping and instruction scheduling techniques [29, 48]. Efforts have demonstrated the benefit of VLIW architectures for either customization or power management [41]. Optimization techniques for VLIW architectures using clusters can also be found in [23]. However, these efforts do not address low-level communication issues. CALiBeR reduces memory pressure in VLIW systems but cannot directly schedule activities to reduce register file communication at the cluster level [1]. Tiwari et al have explored scheduling algorithms for less flexible architectures which split an application between a general purpose processor and an ASIC [44]. Lee shows instruction scheduling benefits for DSP processors [28]. Eckstein and Krall focus on minimizing the cost of local variable access to reduce power consumption in DSP processors [13]. Application specific clusters are investigated in [27, 14]. These complementary scheduler approaches minimize inter- rather than intra-cluster communication and therefore are not able to optimize register utilization as described in this work. In some sense, the fine grain horizontal microcode approach taken here can be viewed as a fine-grained extension of the VLIW concept. However the addition of more sophisticated address generators, multiple address contexts per address generator, the removal of the register file, and the fine-grained steering of data are aspects of this work that are not evident in these other efforts.

The other parallelism approach that is becoming increasingly popular is short vector or SIMD data parallelism [34, 6]. These techniques have been shown to improve performance by up to an order of magnitude on DSP style algorithms and even on some small speech processing codes [22]. The cluster approach is capable of capitalizing on this form of data parallelism as well. From an energy delay perspective however, SIMD operation does not have an advantage and we have therefore not pursued this option.

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 [47]. Imagine [39] has demonstrated the significant performance gain that can be attained when appropriate storage resources surround execution units. Given the poor wire scaling properties, it is somewhat inevitable that function unit clusters will need to be considered [37] in order to manage communication delays in high performance wide-issue super-scalar processors. These approaches however 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.

7. CONCLUSIONS

The perception processor uses a combination of VLIW execution clusters, compiler directed data-flow 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 IV by 1.75 times with an energy delay product that is 135 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.

8. REFERENCES

[1] Akturan, C., and Jacome, M. F. FDRA: A software-pipelining algorithm for embedded VLIW processors. In *ISSS* (2000), pp. 34–40.

[2] Akturan, C., and Jacome, M. F. Caliber: A software pipelining algorithm for clustered embedded VLIW processors. In *ICCAD* (2001), pp. 112–118.

[3] Alvandpour, A., Krishnamurthy, R., Soumyanath, K., and Borkar, S. A low-leakage dynamic multi-ported register file in 0.13mm cmos. In *Proceedings of the 2001 international symposium on*

Low power electronics and design (2001), ACM Press, pp. 68–71.

[4] Banakar, R., Steinke, S., Lee, B., Balakrishnan, M., and Marwedel, P. Scratchpad memory : A design alternative for cache on-chip memory in embedded systems, 2002.

[5] Bona, A., Sami, M., Sciuto, D., Silvano, C., Zaccaria, V., and Zafalon, R. Energy estimation and optimization of embedded vliw processors based onq instruction clustering.

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

[7] Callahan, T., and Wawrzynek, J. 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.

[8] Campbell, M. Evaluating asic, dsp, and risc architectures for embedded applications. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (1998), Springer-Verlag, pp. 261–265.

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

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

[11] Daemen, J., and Rijmen, V. The block cipher rijndael. *Smart Card Research and Applications, LNCS 1820* (2000), 288–296.

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

[13] Eckstein, E., and Krall, A. Minimizing cost of local variables access for DSP-processors. In *LCTES'99 Workshop on Languages, Compilers and Tools for Embedded Systems* (Atlanta, 1999), Y. A. Liu and R. Wilhelm, Eds., vol. 34(7), pp. 20–27.

[14] Faraboschi, P., Brown, G., Fisher, J. A., Desoli, G., and Homewood, F. Lx: a technology platform for customizable VLIW embedded processing. In *The 27th Annual International Symposium on Computer architecture 2000* (New York, NY, USA, 2000), ACM Press, pp. 203–213.

[15] Frigo, M., and Johnson, S. G. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing* (Seattle, WA, May 1998), vol. 3, pp. 1381–1384.

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

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

[18] Gordon, B. M., and Meng, T. H.-Y. A low power subband video decoder architecture. In *International Conference on Acoustics, Speech, and Signal Processing* (1994), pp. 409–412.

[19] Gowan, M. K., Biro, L. L., and Jackson, D. B. Power considerations in the design of the alpha 21264 microprocessor. In *Design Automation Conference* (1998), pp. 726–731.

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

[21] Jain, M. K., Wehmeyer, L., Steinke, S., Marwedel, P., and Balakrishnan, M. Evaluating register file size in asip design. In *Proceedings of the ninth international symposium on Hardware/software codesign* (2001), ACM Press, pp. 109–114.

[22] Joshi, S. M. Some fast speech processing algorithms using altivec technology.

[23] Karl, W. Some design aspects for VLIW architectures exploiting fine - grained parallelism. In *Parallel Architectures and Languages Europe* (1993), pp. 582–599.

[24] Krashinsky, R. Microprocessor energy characterization and optimization through fast, accurate, and flexible simulation. Master's thesis, Massachusetts Institute of Technology, May 2001.

[25] Kwon, J.-H., Lim, J., and Chae, S.-I. A three-port nrerl register file for ultra-low-energy applications. In *Proceedings of the 2000 international symposium on Low power electronics and design* (2000), ACM Press, pp. 161–166.

[26] Lai, C., Lu, S.-L., and Zhao, Q. Performance analysis of speech recognition software. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads* (Feb. 2002).

[27] Lapinskii, V., Jacome, M., and de Veciana, G. Application-specific clustered vliw datapaths: Early exploration 32 on a parameterized design space, 2002.

[28] Lee, C., Lee, J. K., Hwang, T., and Tsai, S.-C. Compiler optimization on instruction scheduling for low power. In *ISSS* (2000), pp. 55–61.

[29] Leupers, R. Instruction scheduling for clustered VLIW DSPs. In *IEEE PACT* (2000), pp. 291–300.

[30] Mathew, B., Davis, A., and Evans, R. A Characterization of Visual Feature Recognition. In *Proceedings of the IEEE 6th Annual Workshop on Workload Characterization (WWC-6)* (October 2003).

[31] Mathew, B., Davis, A., and Fang, Z. 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)* (October 2003).

[32] Mathew, B., Davis, A., and Ibrahim, A. Perception Coprocessors for Embedded Systems. In *Proceedings of the Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)* (October 2003).

[33] Memik, S. O., Bozorgzadeh, E., Kastner, R., and Sarrafzade, M. A super-scheduler for embedded reconfigurable systems. In *ICCAD* (2001), pp. 391–.

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

[35] Palem, K., Rabbah, R., Pinar, V., and Kiran, K. Design space optimization of embedded memory systems via data remapping, 2002.

[36] Panda, P. R., Dutt, N. D., and Nicolau, A. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 5*, 3 (2000), 682–704.

[37] Postiff, M. Function unit clustering in wide-issue superscalar processors.

[38] Rau, B. R. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture* (1994), ACM Press, pp. 63–74.

[39] Rixner, S., Dally, W. J., Kapasi, U. J., Khailany, B., Lopez-Lagunas, A., Mattson, P. R., and Owens, J. D. A bandwidth-efficient architecture for media processing. In *International Symposium on Microarchitecture* (1998), pp. 3–13.

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

[41] Smith, M. D., Lam, M., and Horowitz, M. A. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual Symposium on Computer Architecture* (1990), pp. 344–354.

[42] Soriano, M., Martinkauppi, B., Huovinen, S., and Laaksonen, M. Using the skin locus to cope with changing illumination conditions in color-based face tracking. In *Proceedings of the IEEE Nordic Signal Processing Symposium* (2000), pp. 383–386.

[43] Srivastava, S. Fast gaussian evaluations in large vocabulary continuous speech recognition. M.S. Thesis, Department of Electrical and Computer Engineering, Mississippi State University, Oct. 2002.

[44] Tiwari, V., Malik, S., Wolfe, A., and Lee, M. Instruction level power analysis and optimization of software, 1996.

[45] Tseng, J. Energy-efficient register file design, 1999.

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

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

[48] Weiss, M., and Fettweis, G. Dynamic codewidth reduction for vliw instruction set architectures in digital signal processors, 1996.

[49] Young, S. Large vocabulary continuous speech recognition: A review. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding* (Dec. 1995), pp. 3–28.

[50] Yun, H.-S., and Kim, J. Power-aware modulo scheduling for high-performance vliw. In *ACM SIGPLAN 2001, Workshop on languages, compilers and tools for embedded systems* (2001).