A Loop Accelerator for Low Power Embedded VLIW Processors

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

ABSTRACT

The high transistor density afforded by modern VLSI processes have enabled the design of embedded processors that use clustered execution units to deliver high levels of performance. However, delivering data to the execution resources in a timely manner remains a major problem that limits ILP. It is particularly significant for embedded systems where memory and power budgets are limited. A distributed address generation and loop acceleration architecture for VLIW processors is presented. This decentralized on-chip memory architecture uses multiple SRAMs to provide high intra-processor bandwidth. Each SRAM has an associated stream address generator capable of implementing a variety of addressing modes in conjunction with a shared loop accelerator.

The architecture is extremely useful for generating application specific embedded processors, particularly for processing input data which is organized as a stream. The idea is evaluated in the context of a fine grain VLIW architecture executing complex perception algorithms such as speech and visual feature recognition. Transistor level Spice simulations are used to demonstrate a 159x improvement in the energy delay product when compared to conventional architectures executing the same applications.

Categories and Subject Descriptors:C.3[Special-Purpose and Application-Based Systems]:Real-time and embedded systems

General Terms: Performance, Design

Keywords: Embedded systems, Low power design, VLIW

1. INTRODUCTION

Traditionally, embedded computing was synonymous with the use of highly energy efficient, but low performance processors and micro-controllers for control and monitoring applications. The emergence of super DSPs with ever increasing BDTI scores introduces a new category of compute intensive embedded workloads. Sophisticated applications such as speech recognition, visual feature recognition, secure wireless networking, and general media processing will define the architecture and performance requirements of future mobile embedded environments. The need to deliver high performance at low energy levels has resulted in increased specialization in DSPs. Common tactics include static extraction of ILP using VLIW techniques, specialization of the on-chip memory system, software managed scratch pad memory systems, cache banks which can be locked down under software control, bit-reversed and auto increment addressing modes, etc. Two important themes have emerged: a) improving the throughput of execution resources and b) improving data delivery to those execution resources. This paper will address the architecture of an on-chip memory system which significantly improves the capacity to deliver data to execution resources in a timely manner. It provides addressing mechanisms which generalize the auto increment or vector oriented addressing modes commonly used in processors. The strategy results in radical performance improvements in loop intensive algorithms.

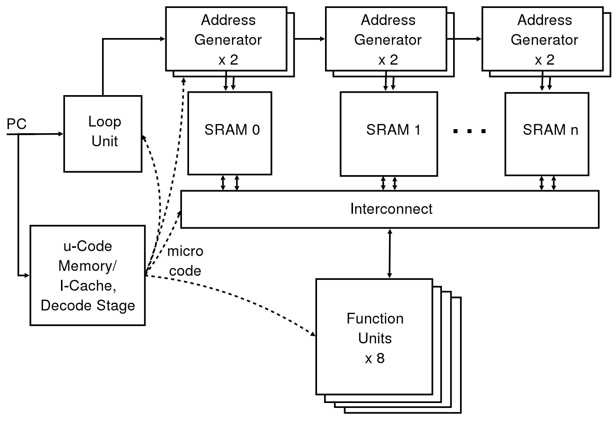

Figure 1: Processor Architecture

A set of ten algorithms from the speech recognition, computer vision, encryption and signal processing domains are used to evaluate this work. These algorithms tend to be stream oriented with a large number of 2D array and vector accesses per elementary operation. Hardware performance counter based measurements on a MIPS R14K processor showed that 32.5% (Geometric mean) of the executed instructions were loads and stores. Since this RISC processor does not have auto increment addressing mode, assuming that there is at least one address calculation instruction for each load and store, it may be seen that 65% of the instructions are related to array accesses in one form or another. Amdhal's law therefore argues in favor of accelerating these patterns.

2. HIGH LEVEL ORGANIZATION

Figure 1 shows the internal organization of a VLIW processor used to evaluate this research. It consists of a set of clock gated function units, a loop unit, multiple software managed dual ported SRAMs and associated address generators (one for each SRAM port), local bypass paths between neighboring function units as well as a cluster wide interconnect.

Traditional processors have a limited number of load/store ports and this limits overall performance and data availability. To efficiently feed data to function units a large number of SRAM ports are required. Increasing the number of ports on a single SRAM or cache degrades access time and power consumption. The traditional solution is banking which motivates our choice of multiple small software managed scratch SRAMs. It is also possible to power down SRAMs which are not required.

To improve operand delivery, an address generator is attached to each SRAM port. Load/store instructions decoded from a VLIW instruction bundle are directly issued to the address generators by the decode stage of the processor. The address generators work in tandem with a VLIW execution unit called the *loop unit*. All branch and loop related instructions are dispatched to it. Several loop count registers and a semi-autonomous state machine maintained within this unit mirror the loop counts of multi-level nested loops running on the processor. Loop parameters including the program counter value at which the loop body starts, start, increment and termination counts etc. are configured into the unit by the compiler before entering a loop intensive section of the application. Thereafter the loop unit works autonomously. Every time the program counter passes the first instruction of a loop body, the corresponding loop count is automatically incremented. The loop count values are used by the address generators. Prior to entering a loop intensive section of code, the start address, lay out and access pattern of arrays used within that section of code is configured into each address generator by the compiler. Once the loop body is started, the address generators use the loop count values along with access pattern information to autonomously generate addresses for load/store instructions. This distributed address calculation leads to high data availability and efficient SRAM port utilization. While the scheme itself is relatively simple, complications arise from additional functionality required in the loop unit and address generators to deal with software pipelining and modulo scheduling as well as offering a level of generality in address generation.

While this loop acceleration method is evaluated in the context of an energy efficient scratch-pad memory system, the method is equally applicable to a multi-bank cache. The only difference is that in the case of a scratch pad memory, physical indices into the memory are used as array variable addresses while in the case of a cache, the address generators work on logical addresses which should then pass through tag lookup. In either case it is possible to localize data structures to just one SRAM bank or to stripe data across multiple banks. In the case of striped data, the compiler configures each bank separately. The loop unit uses the program counter value to increment loop count registers rather than incrementing them with a periodic timer. Hence the loop unit is immune to stalls caused by cache misses.

Any VLIW processor which uses the proposed loop acceleration mechanism needs four new instructions. A single cycle *write_context* instruction transfers loop parameters or data access patterns encoded as a single 32-bit word into context registers within the loop unit or the address generators. The instructions *load.context* and *store.context* are enhanced versions of load/store instructions that use our address generation mechanism. Two fields named *context_index* and *modulo_period* are packed into the immediate constant field of these instructions. *Context_index* controls address calculation by selecting an address generator and a context register within that generator that describes an access pattern. Finally a *push_loop* instruction is used by the compiler to inform the hardware that a nested inner loop is being entered. As explained in Section 2, the PC value may be used to find when a particular loop is being entered. But for an n level nested loop this requires n comparators. With this instruction we are able to take advantage of the strict nesting of loops and compare the PC only against the instruction range of the innermost loop.

3. ARCHITECTURE

The loop acceleration technique will be evaluated on two processor configurations, a VLIW integer core and a VLIW floating point core both of which follow the generic organization shown in Figure 1. Both processors are designed for a 0.13μ CMOS process and operate at 1 GHz, 1.6 volts. The Verilog and Synopsys MCL HDL netlist for these domain specific CPUs optimized for speech recognition and vision were automatically generated from a configuration description using a netlist compiler tool we have developed. Since the loop acceleration technique is applicable to any VLIW processor, the details of the processor implementations will be omitted in this paper. Details may be found in [5].

For this study, both processors issue 8 instructions per cycle and use identical scratch-pad memory systems. The scratch-pad memory consists of 3 dual ported SRAMs of sizes 8KB, 8KB and 2KB. This choice is motivated by the nature of our applications which depend on processing blocks of input, refer to some local state tables and produce blocks of output. There are a total of 6 address generators and a loop unit in each processor.

3.1 Loop Unit

The index expressions of array accesses in a multi-level nested loop will depend on some subset of the loop variables. The purpose of the loop unit is to compute and maintain the loop variables required for address generation in the memory system while the loop body is executed in the function units. Figure 2 shows a simplified organization of the loop unit.

In this implementation, the loop unit can keep track of four levels of loop nest at a time which is sufficient for our applications. For larger loop nests the address expressions that depend on additional outer loops may be done in software as in a traditional processor. A four entry loop context register file holds the encoded start and end counts and the increment of up to four inner most *for* loops.

The loop unit offers hardware support for modulo scheduling, a software pipelining technique which offers high levels of loop performance in VLIW architectures [6]. A brief introduction to some modulo scheduling terminology is necessary to understand the functioning of the loop unit. Assume a loop body which takes *N* cycles to execute. Modulo scheduling allows starting the execution of a new instance of this loop body every *II* (Initiation Interval) cycles where *II* is less than *N*. A normal loop which is not modulo scheduled may be considered a modulo scheduled loop with *II*=*N*. How *II* is determined and the conditions that must be satisfied by the loop body are described in [6]. The original loop body may be converted to a modulo scheduled loop body by replicating instructions such that every instruction that was originally scheduled in cycle *n* is replicated so that it also appears in all possible cycles $(n+i*II)\%N$ where i is an integer. This has the effect of pasting a new copy of the loop body at intervals of *II* cycles over the original loop body and wrapping around all instructions that appear after cycle *N*. If a particular instruction is scheduled for cycle *n*, then n/II is called its *modulo period*. The compiler configures static parameters including *II* and loop count limits into loop context registers. The corresponding dynamic values of the loop variables are held in the loop counter register file. The only other piece of infor-

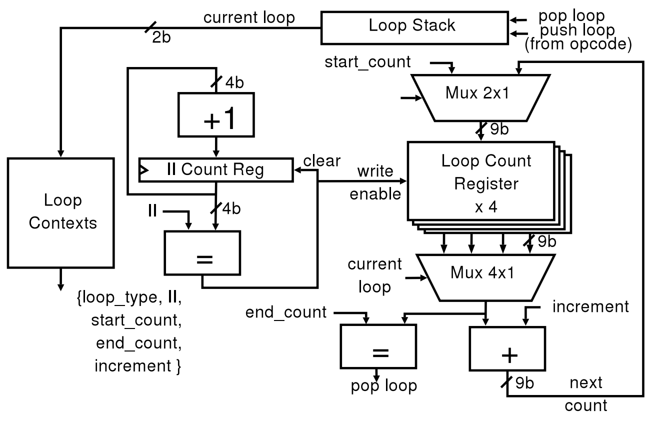

Figure 2: Loop Unit

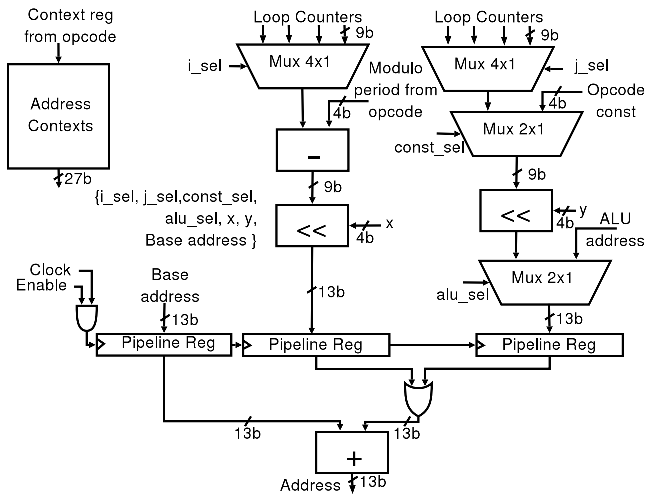

Figure 3: Stream Address Generator

mation required is which loop body is currently pointed to by the program counter. A four entry loop stack captures this information.

Prior to starting a loop intensive section of code, loop parameters (perhaps dynamically computed) are written into the context registers using *write_context* instructions. On entry into each loop body, a *push_loop* instruction pushes the index of the context register for that loop on to the stack. The top of the stack thus represents the current loop body. Whenever the program counter is incremented an *II counter* is also incremented. This register counts up to the initiation interval and then resets itself. Every *II* instructions, the loop increment is added to the loop variable that is held in the loop counter register file. When the end count of the loop is reached, the inner most loop will have completed. The top entry is automatically popped off the stack and the process is repeated for the enclosing loop. Note from Figure 2 that the registers and datapaths have small widths of 4 and 9 bits that cover most common loops. Loops which don't fit this size can always be handled by additional software. The reduced bit widths reduce energy for the common case.

3.2 Stream Address Generators

Address computations for array and vector references issued to an SRAM port are handled by its attached stream address generator. The operation of the address generator depends on the loop unit counters and array parameters like base address and row size that are stored in its address context register file. Figure 3 shows the internal structure of an address generator.

To understand how this simple structure can accomplish a variety of address calculations, it is essential to understand how a compiler generates addresses for array references. Consider the 2D arrays declared and used in *C* as shown in Figure 4. To simplify the discussion, we will assume word oriented addressing. Let the size of the $Complex$ struct be denoted as $elem\_size$. Then, the size of one row of A is $row\_size = elem\_size * N$. If the offset of $imag$ within the struct is 1 and the base address of A is $Base_A$, then the base addresses of the $imag$ field will be $Base_{imag} = Base_A + 1$. So the address expressions corresponding to the load into $t1$ is $Base_{imag} + i * row\_size + j * elem\_size$ since C stores arrays in row major order. A vector is a single dimensional array, so its address expression is just a special case where $row\_size = 0$. For more complex index expressions of the form $P * i + Q$, the factors P, Q may be absorbed into the row size and base address respectively. A column-walk of the form $A[j][i]$ can be evaluated similarly. By constraining the row and element sizes to be a powers of two, the address expression reduces to the form $address = Base + ((i << x)|(j << y))$.

For cases where row size cannot be a power of two, to help pack more data into the scratch memory, row size may be picked as the sum of two powers of two and separate expressions may be used to access the bulk of the array and the residue. For arrays with $n > 2$ dimensions, the base address is repeatedly recalculated to account for $n-2$ dimensions and the last two levels of loop nest are supported by the hardware. Not all array accesses need to use the same loop variables. In the example, the access of B depends on i, k unlike A which depends on i, j. The address generator selects the correct loop variables to be used for address expression.

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 can not 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 effectively deal with indirect access streams of the type $A[B[i]]$. Address generation adds 1 cycle latency to load/store operations.

Before entering into a loop intensive section of code, the compiler uses *write_context* instructions to write descriptions of array access patterns into the address context register files of address generators. For increased throughput the same access pattern may be written into multiple address generators attached to the same SRAM. Each address context includes the row and element sizes,

```
struct Complex A[N][M];
struct Complex B[N][K];
...
for(i=0; i<N; i++) { ...
  for(j=0; j<M; j++) { ...
    t1 = A[i][j].imag; ...
    for(k=0; k<K; k++) { ...
      t2 = B[i][k].real;
```

Figure 4: Array Access Example

the base address as well as the loop counter indices that correspond to the arrays loop variables. In this implementation there are four context entries in each address generator which support 24 simultaneous access patterns. Since *write_context* is a single cycle operation, dynamic reconfiguration has very low overhead. The parameters for an array access pattern are packed into a single 32-bit word with the base address at the lsb. Arithmetic can then be done on the packed word to update the base address dynamically.

When the compiler generates code for an array access, the index of the address generator and the index of the address context register within that generator are encoded into the immediate field of the load/store instruction. The selected address generator then uses the context index field to retrieve the array parameters from the context register file as shown in Figure 3. The retrieved context entry specifies the loop variables to be used for calculating the address. The muxes at the top right of the figure use this information to select the appropriate loop variables. The shifters then shift the selected loop variables and the result is $ORed$ and added to the base address to generate an address. To improve cycle time, pipeline registers have been inserted just before the final add operation.

Several special cases are handled in the address generator. It is common to unroll loops by a small factor and software pipeline them for performance. In that case, instead of using 2 loop variables, it is possible to use one loop variable and one unroll factor to compute the address. The unroll factor is packed into the immediate field of the instruction and selected in lieu of the loop variable using the upper 2x1 mux in the figure. When the access pattern is too complex to be handled by the address generator, the lower 2x1 mux selects an address that is computed by an ALU. To handle vectors and ALU generated addresses with one or zero loop variables respectively, the loop unit has a special loop counter which is always zero.

3.3 Array Variable Renaming

Setting the *modulo period* field in *load.context/store.context* instructions to a non-zero value unlocks a performance enhancing feature we call *Array Variable Renaming*. Modulo scheduling makes it is possible to overlap the execution of multiple instances of the inner loop body. Assume that *k loop* from our example has a latency of 30 cycles and that after satisfying resource conflicts and data dependences it is possible to start a new copy of the loop body every 5 cycles. Then, up to 6 copies of the loop body could be in flight through the execution pipeline. To get data dependences correct for new loop bodies, the loop variable should be incremented every 5 cycles. However, when it is incremented, old instances of the loop body which are in flight will get the wrong value and violate dependences for load/store instructions that happen close to the end of the loop body.

The traditional solution is to use multiple copies of the loop variable in conjunction with the VLIW equivalent of register-renaming: a rotating register file. Multiple address calculations are performed, the appropriate values loaded into the register file and the register file is rotated. For long latency loop bodies with short initiation intervals, this leads to increased register pressure. Our solution to this problem is to increment a single copy of the loop variable every initiation interval and compensate for the increment in older copies of the loop body which are in flight. The compensation factor, which is really the modulo period, within which a load/store instruction belongs is encoded into the immediate field of load/store instructions. It may be thought of as a compiler generated tag attached to load/store instructions. It is subtracted from the loop variable's value to cause dependences to resolve correctly. In effect, this has the effect of *rotating* the array variable and letting a generic expression like $A[i][j]$ be re-bound to separate addresses. *Array variable renaming*, permits using the entire scratch pad memory as a rotating register file with separate virtual rotating registers for each array in the program. This allows removes the need for a register file in our processor. The result is very high throughput while consuming low power.

3.4 Addressing Modes

The address generator can directly compute array references of the form $A[i * P + Q][j * R + S].field$ and vector accesses when both loop variables are nested loops, when one loop has been unrolled, and more importantly when the inner loop has been modulo-scheduled. For higher dimensional arrays, the base address is repeatedly re-computed using an ALU and the last two dimensions are handled by the address generator.

Another important access pattern is indirect access of the form $A[B[i]]$. It is a common ingredient of neural network evaluation. It is also a generic access pattern – any complex access pattern can be pre-computed and stored in $B[\ ]$ and used at run-time to access the data in $A[\ ]$. For example we are able to implement bit-reversed addressing for FFT using this mechanism. By passing an ALU generated $B[i]$ address through the adder in Figure 3 thereby offsetting it with a base address we provide indirect vector access.

The ALU address can be computed or it can be streamed into the ALU from SRAM by another address generator. Using two address generators and an ALU, complicated access patterns may be realized with high throughput. The stream address generator effectively converts the scratch-pad memory into a vector register file that can operate over complex access patterns and even interleave vectors for higher throughput. In a sense this unifies the vector and VLIW architecture styles.

4. EVALUATION

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. In order to compare our approach to the the competition, four different implementations are considered: 1) Software running on a 400 MHz Intel XScale 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 our VLIW architecture. 4) Four of the benchmarks are compared to custom ASIC implementations.

4.1 Benchmarks

The first two algorithms called GAU and HMM are dominant components of several speech recognizers. Together, they consume 99% of the execution time of the CMU Sphinx 3.2 speech recognizer [4, 5]. Fleshtone, Erode and Dilate, are used for image segmentation in a visual feature recognition system [5]. Rowley is a neural network based face detector [7]. Viola is a wavelet based face detector [8]. FFT, FIR and Rijndael represent the DSP and encryption domains. These were added to test the generality of our approach. FFT implements a 128 point complex to complex Fourier transform on floating point data. The cluster implementation uses a simple radix 2 algorithm. The software version on the Pentium uses FFTW, a highly tuned FFT implementation which is believed to be one of the fastest in the world. FIR implements a 32 tap finite impulse response filter. Rijndael, the AES standard is used here to encrypt 576 byte Ethernet packets using a 128 bit key. 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.

4.2 Metrics

Two important metrics used in this evaluation are throughput and the energy consumed to process one block of input. The trade off 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. Gonzalez and Horowitz show that a good metric of architectural merit should be based on the rate of work per energy or the energy delay product [3]. Both architecture and semiconductor process influence the energy delay product. Since the feature size of the process, λ, has a large impact it is necessary to normalize designs to the same process for comparison. Under the assumption of constant field scaling energy, delay and energy-delay product scale as λ^3, λ and λ^4 respectively [9]. Since our VLIW processors and the Pentium 4 are both implemented in a 0.13μ CMOS process, the data for these systems is not scaled. The XScale is implemented in a 0.18μ process. Hence all XScale numbers are scaled to give it the advantage of a better CMOS process.

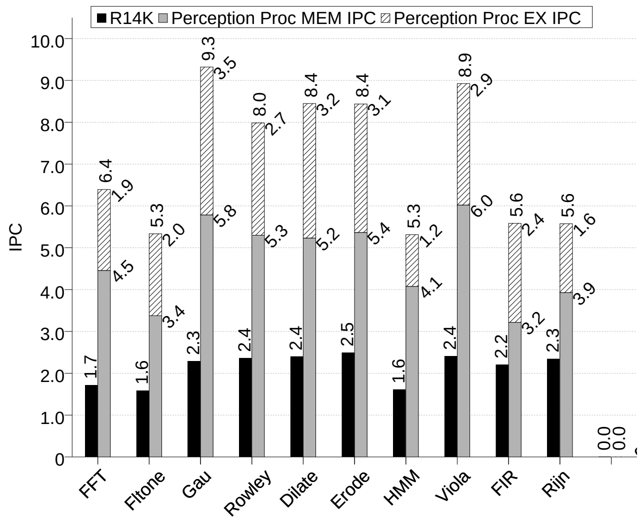

Figure 5: IPC

4.3 Experimental Method

This evaluation is based on a 0.13μ hardware design of our cluster operating at 1 GHz, 1.6 v. The design is simulated at the transistor level using Spice (Synopsys Nanosim) and the 0.13μ BPTM transistor models provided by the Device Group at UC Berkeley. Spice provides a supply current waveform which is used to compute instantaneous power consumption. Numerical integration of power over time provides the energy consumption. The SRAMs are generated as macro-cells by a CAD tool. Simulating the entire SRAM array using Spice is not feasible. For SRAMS we 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. Netlists have clock trees and pessimistic heuristic wire loads incorporated.

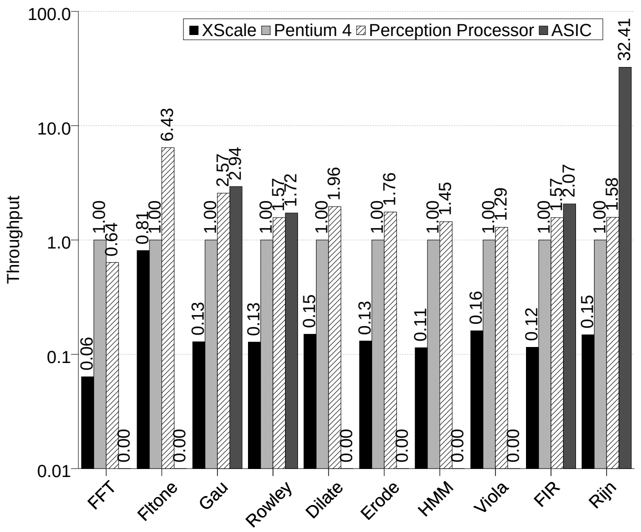

Figure 6: Throughput normalized to Pentium 4 throughput

For the XScale and Pentium experiments, circuit boards were modified to permit a current probe and a digital oscilloscope to measure average processor current. These changes isolate the processor power consumption from other components. Both the Pentium and XScale systems were chosen to permit these modifications. Additional comparison against a DSP system would be desirable, but no system suitable for such PCB modification was found to date.

The XScale does not have floating point instructions required for some of the benchmarks. The comparison is therefore made against an *ideal* XScale which has FPUs with 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 values are meaningless, but the performance and energy consumption represent a lower bound for any real implementation with FPUs.

5. RESULTS

The loop accelerator and scratch-pad memory design focused on the need to provide high operand flow to the function units. This improves function unit utilization and IPC. Figure 5 shows the IPC of the cluster compared against the IPC measured using performance counters on a MIPS R14K CPU. Our VLIW processor vastly outperforms the the out of order processor and the highly optimizing SGI MIPSPro compiler. The mean[1] IPC of the processor is 3.3 times that of the competition. A large fraction of this performance advantage is due to the on-chip memory subsystem.

Sufficient throughput to meet real time performance is important for stream computations. Figure 6 shows the throughput of the various implementations. Throughput is defined as the number of input packets processed per second. The numbers are normalized to the throughput of the Pentium 4. Our VLIW architecture outperforms the Pentium 4 by a factor of 1.75 and achieves 41.4% of the ASIC's performance. The real advantage of the architecture becomes apparent in Figure 7 where it may be seen that this throughput is achieved at energy levels that are on average 13.5 times better than the XScale.

[1]All references to mean and average imply the Geometric Mean

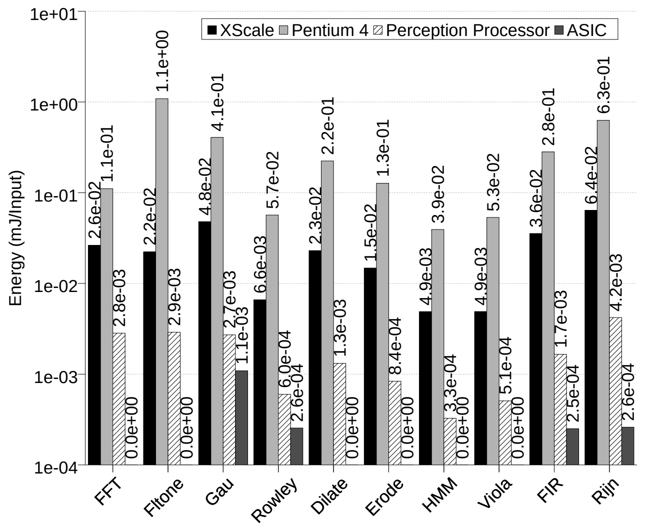

Figure 7: Process Normalized Energy Consumption

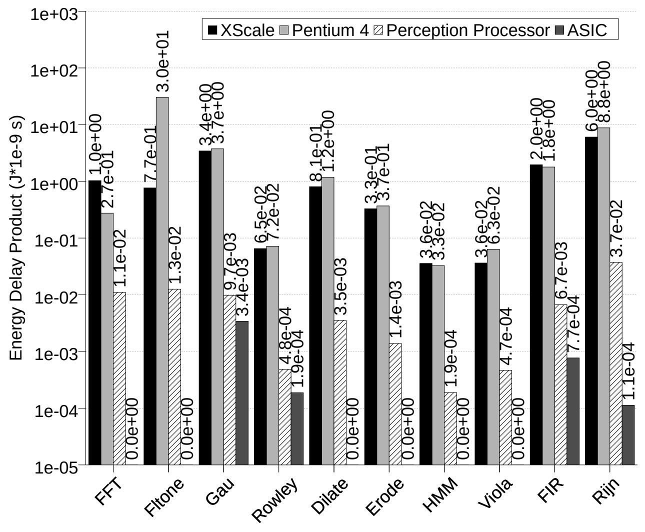

Figure 8: Process Normalized Energy Delay Product

Figure 8 shows that the cluster architecture improves the energy delay product over the Xscale on average by 159 times and is only 12 times worse than an ASIC implementation. Note that the last three graphs use a log scale for the Y axis. The radical improvements in energy and performance suggests that when high performance, programmability and low energy levels are crucial, a loop accelerated VLIW architecture is an attractive alternative to general purpose processors and ASICs.

6. RELATED WORK

Banakar et al. have demonstrated the advantages of scratchpad memories as a power saving mechanism [1]. Processors like the IBM Elite DSP include vector extensions that can operate on disjoint data, but do not possess the level of autonomy our distributed address generation mechanism provides. The Reconfigurable Streaming Vector Processor targeted at low power multimedia systems implements hardware acceleration for base-stride vectors [2]. Vector register files possess a limited amount of autonomy in address generation, but the scheduling is completely done in hardware and is therefore inflexible. Our method is not only more general than vector processing, but also amenable to compiler instruction scheduling since we use enhanced scalar loads in conjunction with a VLIW architecture. High performance DSP processors provide bit-reversed addressing modes and auto increment instructions. The methods described in this paper are significantly more general in the access patterns they can deal with.

7. CONCLUSIONS

Adding a distributed address generation and loop acceleration mechanism to a VLIW processor has been shown to have considerable benefit both in terms of power and energy consumption. The architecture has been evaluated for important classes of future embedded applications like speech recognition, vision and cryptography. This approach has a number of advantages: the approach provides performance and energy efficiency that is close to an ASIC while avoiding the costs of an ASIC and retaining most of the generality of a general purpose processor approach;, it achieves an energy-delay product that is 159 times better than an idealized Intel XScale processor while delivering 1.75 times the performance of a Pentium IV system; it makes sophisticated perception applications possible in real-time within an energy budget that is commensurate with the embedded space.

8. REFERENCES

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

[2] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (RSVPTM). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 141. IEEE Computer Society, 2003.

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

[4] 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.

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

[6] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74. ACM Press, 1994.

[7] 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.

[8] 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.

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