

An Energy Efficient High Performance Scratch-pad Memory System

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

Abstract

A low-power high-performance scratch-pad memory system for an embedded VLIW processor is presented. It uses a simple stream address generator capable of implementing a variety of addressing modes. Array variable rotation, a technique that can replace register renaming and rotating register files is discussed. Factor of 135x energy-delay advantage is demonstrated using Spice simulations of the processor running speech, vision and signal processing algorithms.

1. Introduction

The convergence of mobile computing, ubiquitous computing and traditional embedded computing has accelerated in recent times. To realize the promise and the full potential of pervasive computing future mobile embedded environments need to support sophisticated applications such as speech recognition, visual feature recognition, secure wireless networking, and general media processing. These applications require significantly more performance than typical embedded processors can deliver. Even modern high performance processors can barely keep up with the real time requirements of sophisticated perception applications [2]. Because of performance scaling predicted by Moore's law, the performance issue is not by itself a critical problem. The two crucial problems of perception applications are: a) Energy consumption – The high end processors that can deliver the performance required to meet real time operation consume too much power to be usable in the embedded space. b) Processor availability – Perception and security interfaces are by nature *always on*. This limits the processor availability for other compute tasks such as acting upon what was perceived.

ASICs are the standard means of reducing power consumption while increasing performance. But their lengthy expensive design cycles and lack of flexibility make them undesirable candidates for rapidly evolving applications. FPGA devices provide a level of specialization while retaining significant generality. But they suffer both in perfor-

mance and power when compared to either ASIC or CPU logic functions. Processors with application domain specific acceleration mechanisms represent the middle ground.

Our work focuses on the rapid automated generation of high-performance energy efficient VLIW processors for perception applications. A number of function units are organized as a cluster and embedded in a reasonably rich interconnection network which provides communication between function units and several scratch-pad memories. We have demonstrated in previous work that perception applications are stream oriented [2]. A host processor or a DMA engine streams data into or out of the VLIW processor via double buffered I/O SRAMs. Local storage is provided by the scratch SRAMs and the cluster is controlled by a microcode program held in an instruction SRAM. Though the processor was initially designed with perception applications in mind, it is equally applicable to other streaming problems. Several of the benchmarks used in this paper are from the perception domain, but we have added encryption and signal processing algorithms to demonstrate the generality of the approach.

The mix of function units and the organization of the interconnect, the scratch-pad memory and associated address generators are specified using a configuration file. A cluster compiler automatically generates the Verilog HDL description of the domain specific processor and customizes a software simulator for the processor. Details of the overall operation of the processor are available in [2]. This paper will focus exclusively on the scratch pad memory system which is largely responsible for the excellent performance. Even though later sections will use specific numbers for the type and number of scratch-pad memories and bit-widths of datapaths, the memory system is quite generic and different configurations and bit-widths can be automatically generated. The specified widths only refer to the actual configuration used to evaluate this work. Using a process normalized energy delay product as the metric [5], the effectiveness of the cluster is compared against the obvious competition: general purpose processors and ASICs.

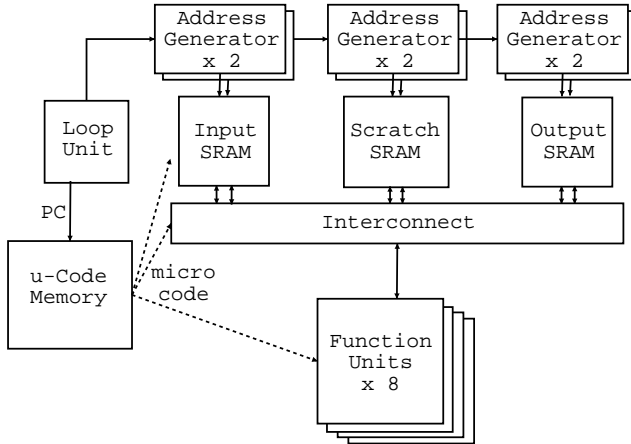


Figure 1. Cluster Architecture

2. Processor Architecture

Figure 1 shows the internal organization of a cluster. 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. A register file is conspicuously absent because the use of compiler controlled data-flow in conjunction with array variable rotation which will be discussed shortly makes architected registers unnecessary [2]. The lack of a large multi-ported register file leads to significant energy savings.

In general, cluster configurations are intended to have up to 8 function units (FUs). This limit is imposed by a target operational frequency of 300 MHz in a 2.5V, 0.25 μ CMOS process our processor was originally designed for. The design has been subsequently scaled to work at 1GHz in a 0.13 μ process. In this paper, two hardware configurations whose netlists were generated by the cluster compiler will be considered: one with 4 ALUs and 4 FPU and another with 4 ALUs, 2 integer multipliers and 2 empty slots. Both systems have the same memory configuration shown in Figure 1. The SRAMs labeled *Input* and *Output* are double buffered and are 8KB and 2KB in size respectively. Input blocks are streamed into the *Input* SRAM and *Output* blocks are streamed out of the *Output* SRAM. Left over space is used as temporary storage. The 8KB SRAM named *Scratch* holds local state and is not double buffered.

3. Memory System Architecture

Perception applications are stream oriented with a large number of 2d array and vector accesses per elementary operation. Traditional processors have a limited number of load/store ports and this limits overall performance because of the high number of array accesses [2]. 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 worsens access time and power consumption. This motivates our choice of multiple small software managed scratch SRAMs. It is also possible to power down SRAMs which are not required. For low leakage processes a large fraction of the energy consumption is in the sense amplifiers of the SRAM ports. They consume approximately 50% of the processor energy in our 0.25 μ implementation. Mechanisms to efficiently use these expensive resources are important for both performance and energy conservation.

Hardware performance counter based measurements on a MIPS R14K processor showed that 32.5% (Geometric mean) of the executed instructions were loads/stores for a set of benchmarks described in Section 4. The high rate of load/store operations combined with the regular array access patterns makes it feasible to overlap computation and SRAM access possible using hardware accelerators. A large fraction of the remaining 67.5% execution component is array address calculations that support load/store operations. Our approach is to associate with each SRAM port, an address generator that deals with common access patterns of streaming applications.

Before we delve into details, please note that though the scratch-pad memory system is evaluated in the context of our processor it is applicable to any processor which has an immediate constant field in load/store instructions. To reap the benefits from our approach 4 new instructions are required: a) *write_context context_index, src_reg* – transfer a description of the access pattern from the source register to a context register within the memory system. b) *load_context dest_reg, context_index* and *store_context context_index, src_reg* – these are loads/stores that use our address generation mechanism. The *context_index* encoded into the immediate constant field of the instruction specifies the address generator to be used and the index of a context register within it. c) *push_loop context_index* – let the memory system know that a new loop is starting.

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 itself is executed in the function units. Figure 2 shows a simplified organization of the loop unit.

In our current 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

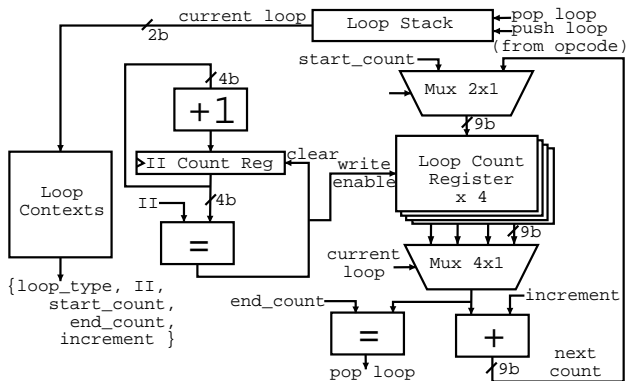


Figure 2. Loop Unit

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. Loop initiation interval represented as *II* in the figure is also stored. For normal loops the *II* is the latency of the loop body. For modulo scheduled loops this is the modulo initiation interval – the period between starting successive instances of the loop body. Writing a context into the loop context register file is a single cycle operation. Loop contexts represent the static parameters of the loop. The corresponding dynamic values of the loop variables are held in the loop counter register file. The only other piece of information required is the current loop nest, i.e., which loop body the program counter is currently within. A four entry loop stack captures this information.

Just before 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 loop body within which the program counter is at the moment. An *II counter* repeatedly counts up to the initiation interval and then resets itself. Every *II* cycles, 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 done in software, so the reduced bitwidths save energy in the common case.

3.2. Stream Address Generators

Address computation for array and vector references issued to an SRAM port are handled by its attached stream address generator. The operation of the address generator

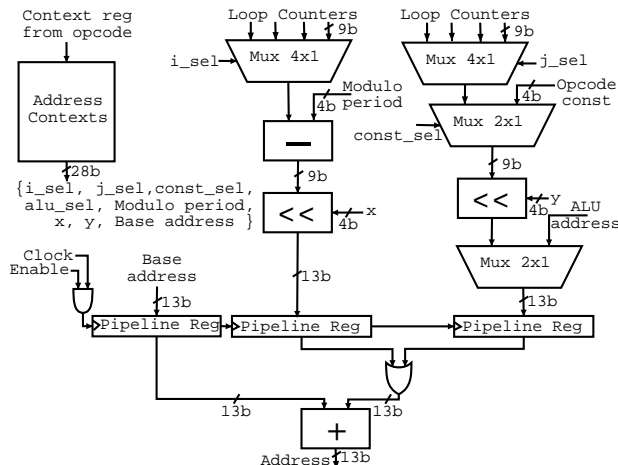


Figure 3. Stream Address Generator

depends on loop counters from the loop unit 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.

```

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

3.2.1. Array Address Expression 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. The size of a *Complex*, denoted as *elem_size* is 2 words. Thus, the size of one row of *A* is *row_size* = *elem_size* * *N*. The offset of *imag* within the struct is 1. If 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 constrain-

ing the row size to be a power of two, the address expression reduces to $address = Base + ((i \ll x)|(j \ll y))$.

The function of the address generator is to compute the address expression. 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 left to the hardware to deal with. 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 knows how to pick the correct loop variables and plug them into the address expression.

3.2.2. High Level Operation 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. Each address context includes the row and element sizes, the base address as well as the loop counter indices that correspond to the arrays loop variables. In our current implementation there are four context entries in each address generator corresponding to a total of 24 access patterns simultaneously. 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. So arithmetic can be done on the packed word to update the base address dynamically.

When the compiler generates code for an array access, the index of 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 the processors 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 com-

plex 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.

3.2.3. Array Variable Rotation Setting the *modulo period* field in the address context to a non-zero value unlocks a performance enhancing feature called *Array Variable Rotation*. Modulo scheduling makes it 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 is encoded into the immediate field of 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 rotation*, permits using the entire scratch pad memory as a rotating register file with separate virtual rotating registers for each array in the program. Though array variable rotation can be used in conjunction with a rotating register file, it is much more powerful. Our processor does not have an architected register file at all – it merely uses array variable rotation in the place of register-renaming to achieves very high throughput at low power.

3.2.4. 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 $A[B[i]]$. This happens in the bit-reversal phase of an FFT and 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 $A[]$. 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 in conjunction, complicated access patterns may be realized with high throughput. If the cost in terms of SRAM and function unit usage becomes too high, the address generator may be extended for other application specific access patterns. 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. Thus, it unifies the vector and VLIW architecture styles.

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 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 cluster architecture. 4) Half 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 [7, 2]. Fleshtone, Erode and Dilate, are used for image segmentation in a visual feature recognition system [2]. Rowley is a neural network based face detector [10]. Viola is a wavelet based face detector [12]. 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 network 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

Gonzalez and Horowitz show that a good metric of architectural merit should be based on the rate of work per energy or an energy delay product [5]. 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 ideal scaling conditions the energy delay product scales as λ^4 [6]. Since the threshold voltage rarely scales ideally, a more reasonable energy delay scaling model lies between λ^2 and λ^3 [5]. Hence, we have chosen a λ^3 basis, where delay and energy are each scaled by λ^2 and λ respectively.

4.3. Experimental Method

This evaluation is based on a 0.13μ hardware design of our cluster operating at 1GHz, 1.6v. The design is simulated at the transistor level using Spice while running the micro-code for the benchmarks. 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.

For the XScale and Pentium experiments, the PCBs have been modified to permit a current probe and a digital oscilloscope to measure average processor current. The XScale does not have floating point instructions required for some of the benchmarks. We therefore compare against an *ideal* XScale 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 mean-

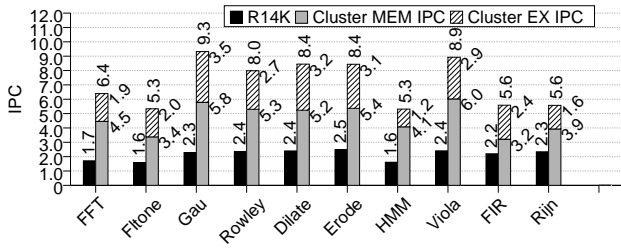


Figure 5. IPC

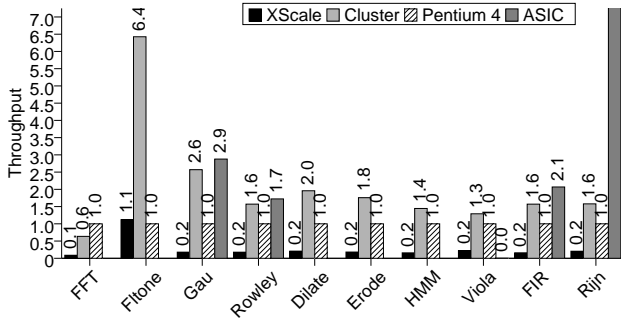


Figure 6. Throughput normalized to Pentium 4 throughput

ingless, but the performance and energy consumption represent a lower bound for any real implementation with FPU.

5. Results

The basic design of the memory system focused on the need to provide high data rate to the function units. This leads to high function unit utilization and high IPC. Figure 5 shows the IPC of the cluster compared against the IPC measured using h/w performance counters on a MIPS R14K CPU. The cluster vastly outperforms the out of order processor and the memory system component of the IPC alone is 2 to 3 times that of the R14K.

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. Figure 6 shows that the cluster architecture outperforms the Pentium 4 by a factor of 1.75 (Geometric Mean). 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 16x better than the XScale.

Figure 8 shows that the cluster architecture improves the energy delay product over the Xscale by 135 times (Geometric Mean). Note that the last two graphs use a log scale. This radical improvement suggests that when high performance, low design time and low energy levels are crucial, the cluster is an attractive option.

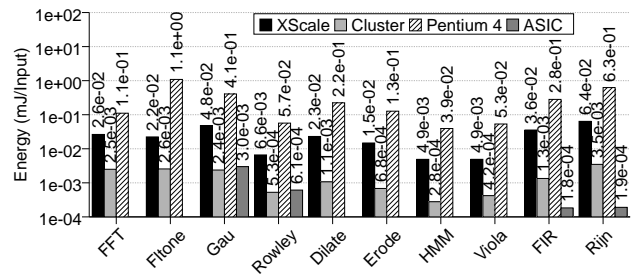


Figure 7. Process Normalized Energy Consumption

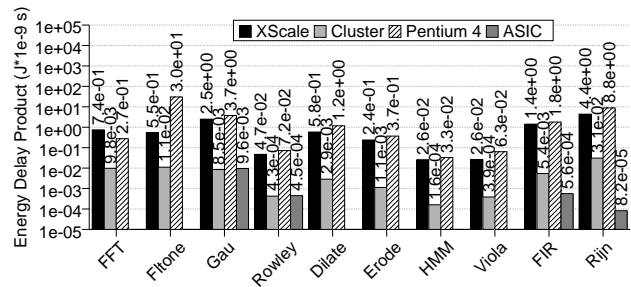


Figure 8. Process Normalized Energy Delay Product

6. Related Work

The use of memory hierarchies to save power has been explored by Panda [9] and the use of scratch-pad memories has been shown to be effective [3]. Increasing performance via VLIW techniques is a common theme in modern embedded systems including mapping and instruction scheduling techniques [8, 4, 1]. Efforts have demonstrated the benefit of VLIW architectures for either customization or power management [11]. Processors like the IBM Elite DSP include vector extensions that can operate on disjoint data similar to our cluster.

7. Conclusions

Combining domain specific execution clusters with a high performance scratch-pad memory system can produce high performance low-power accelerators for embedded streaming applications. This approach has a number of advantages: the design cycle is extremely short when compared to an ASIC; it achieves an energy-delay product that is 135 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.

References

- [1] C. Akturan and M. F. Jacome. Caliber: A software pipelining algorithm for clustered embedded VLIW processors. In *ICCAD*, pages 112–118, 2001.
- [2] A. Author. Elided for blind review. 2003.
- [3] 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.
- [4] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon. Energy estimation and optimization of embedded vliw processors based on instruction clustering.
- [5] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.
- [6] B. M. Gordon and T. H.-Y. Meng. A low power subband video decoder architecture. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 409–412, 1994.
- [7] 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.
- [8] R. Leupers. Instruction scheduling for clustered VLIW DSPs. In *IEEE PACT*, pages 291–300, 2000.
- [9] P. R. Panda, N. D. Dutt, and A. Nicolau. 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):682–704, 2000.
- [10] 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.
- [11] M. D. Smith, M. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 344–354, 1990.
- [12] 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.