Perception Coprocessors for Embedded Systems

Binu Mathew, Al Davis, Ali Ibrahim
School of Computing, University of Utah
{mbinu | ald | ibrahim}@cs.utah.edu

Abstract

Recognizing speech, gestures, and visual features are important interface capabilities for embedded mobile systems. Perception algorithms have many traits in common with more conventional media processing applications. The primary motivation for this work is that applications such as real-time, speaker-independent, large-vocabulary, domain-independent continuous speech recognition systems require more performance than is currently available on embedded processors. Even on modern high-performance processors the performance is just barely able to keep up with real-time demands while consuming power at a rate that is well beyond what can be sustained on mobile systems. The solution to this dilemma has traditionally been to design a special ASIC. ASIC design however is both expensive and lacks the generality needed to support different phases of a complex algorithm or even evolutionary improvements to base method. This paper introduces an execution cluster based coprocessor architecture and its CMOS implementation. This is compared against software implementations of algorithms running on a general purpose processor and also against custom ASICs. The cluster achieves an order of magnitude improvement in energy consumption over a conventional processor while retaining a reasonable level of generality. The architecture is evaluated on several important perception applications where energy consumption is shown to improve by a factor of 12-55 times and energy-delay product improves by a factor of 3.8 - 40 times over conventional processor approaches.

1 Introduction

The focus on embedded computing has both diversified and intensified as mobile computing, ubiquitous computing, and traditional embedded applications continue to converge. There is a need to 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, and running them on high-performance processors would consume an intractable level of power. The usual solution to this dilemma is to design a custom ASIC. ASIC design cycles are both long and costly and the level of specialization often limits their utility to a single phase and instance of the application. Reconfigurable hardware approaches are flexible but they sacrifice both performance and power to a degree that is intolerable for sophisticated applications such as speech recognition.

This paper evaluates a methodology targeted at the rapid creation of powerful energy-efficient coprocessors for perception applications. The approach includes a specialized compiler to map applications onto a domain specific function unit cluster which is general enough to support multiple algorithms or multiple phases of a complex algorithm. The compiler can further optimize the cluster design to generate a custom ASIC design in much less time than required for hand-built ASIC designs. The cluster architecture concept is open ended. In this paper we investigate the utility of two specific cluster architectures: one for integer applications and another for floating point applications. The compiler generates code in a power-efficient manner that is essentially horizontal micro-code. This provides fine-grained control over data steering, clock gating, and function unit utilization. Efficiency is primarily the result of minimized communication. Data values are transported only once and only when required; the inherent registers in bypass paths are utilized to avoid unnecessary accesses to a register file while also providing a form of register renaming. The resulting active data-path is very close to a custom ASIC for the application. However the active data-path in the cluster does utilize multiplexer circuits that provide generality but would be missing in a custom ASIC design. Hence increased customization removes these multiplexers which further reduces power consumption at the cost of reduced generality.

The result is an architecture which 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 processor architectures and ASICs. The cluster approach

achieves a design efficiency that cannot be achieved by a highly specialized ASIC, while delivering a performance and energy efficiency that cannot be matched by general purpose processor architectures.

The benefit of this approach is tested on five benchmarks that were chosen both for their importance in future embedded systems as well as for their algorithmic variety. Three represent key components of perception systems and the other two were chosen from encryption and DSP domains to test generality of the approach. The first two benchmarks, GAU and HMM, represent the Gaussian and Hidden Markov Model evaluation algorithms which account for over 99% of the execution time for the CMU Sphinx3 speech recognition system [9]. We have applied cache and memory system optimizations to the original algorithms [15]. The third benchmark, ANN (artificial neural network evaluation), is a key component of visual feature recognition [17] The fourth benchmark is the AES encryption standard, Rijndael. The final benchmark is a 16 tap FIR filter, a common DSP component used in many embedded applications.

In order to compare this approach to the the competition, four different implementation of each benchmark is considered:

1. C source code compiled and executed 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. Additional issues are discussed in section 3.

2. A clustered implementation generated by our microcode compiler from static single assignment code.

3. A custom hardware implementation which is generated from the clustered implementation by removing any data paths and multiplexer ports that are not needed by a particular application.

4. A custom designed ASIC which is highly optimized for the particular algorithm.

In the next section, we delve into the details of our architecture. A more detailed description of the characteristics of the perception benchmarks and arguments in favor of why the architecture is well suited for perception may be found in [15, 14].

2 Architecture

The common trait exhibited by perception applications and other continuous media is that data is periodic, stream oriented, and must meet real time deadlines. Input data blocks vary with the specific algorithm but are small enough to easily fit in an on-chip SRAM buffer. An input data block is accessed until an output block and state information is generated and then the input block is discarded and the process repeats itself. The state information storage requirement is also small and capable of being resident in on-die SRAM array. The output block can be streamed out of the coprocessor as it is created.

These intrinsic data properties lead to a high performance implementation in the form of a Decoupled Access/Execute coprocessor architecture (DAE). The high level organization of such an accelerator is depicted in Figure 1. In DAE designs, data is pushed into a coprocessor by a host processor or memory controller. Processed data is removed periodically by the host processor, but the coprocessor handles the rest of a constant rate compute intensive task on its own. For algorithms with simple periodic access patterns, address generation may also be entrusted to the coprocessor which allows the main processor to proceed in parallel with other tasks.

In this paper we compare two kinds of coprocessor implementations: 1) a domain specific execution cluster running software and 2) a custom execution unit. A 300 MHz host CPU with an instruction set similar to a MIPS R4600 has also been designed, but it is not the focus of this paper. The 300 MHz target was chosen to make the processing power of the host CPU similar to the well-known Intel StrongARM processor. Our designs were done in a 2.5 volt 0.25μ CMOS process.

2.1 Cluster Architecture

Execution clusters in this context are comprised of multiple function units that may be tailored to the specific functional needs of one or more applications. The overall data flow between the function units within a cluster is controlled by microcode generated by our compiler. Software control allows a particular instance of an execution cluster to support more than for just a single application. Figure 2 shows the internal organization of a cluster. In our current implementation, a cluster can have at most 7 function units and a register file. The limit of 7 FUs is not fundamental to the architecture, but is imposed by the maximum bypass path complexity that can be tolerated to achieve a clock rate of 300 MHz in our target process. Increasing the number of function units adds potential parallelism but increases the multiplexer delay. For generality, the multiplexers must have a port for each functional unit plus an additional port for the register file. Most function units operate on 2 input operands and generate a single result. The exception is load/store units which access SRAM arrays. All data-paths are 32 bits wide.

The architecture supports multi-cycle communication between multiple on-chip execution clusters. However,

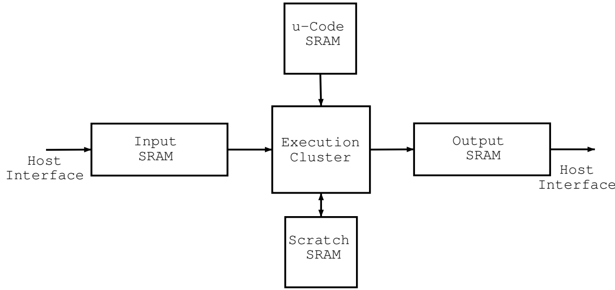

Figure 1: Coprocessor Organization

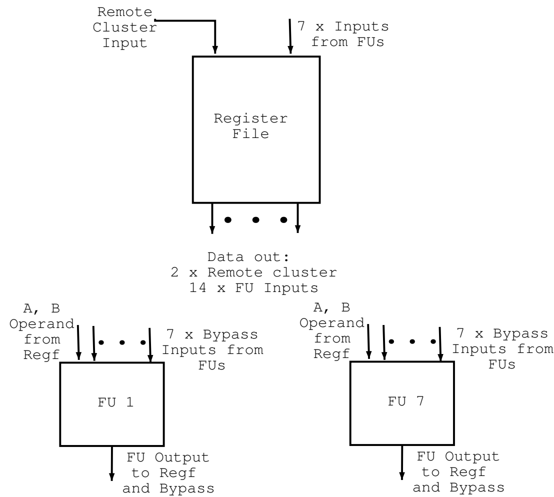

Figure 2: Cluster Organization

our micro-code compiler has not yet been extended to handle inter-cluster communication and therefore only single cluster properties are discussed in this paper.

Figure 3 shows the internal organization of the register file. To accommodate the storage requirements of 7 FUs, the register file provides 14 outputs and an additional 2 outputs for inter-cluster communication. The physical register file has only 4 read ports and 2 write ports. The 4 physical read ports are shared across 16 output connections. Only 4 individual values may be fetched from the register file in each cycle, but those values may be sent to any or all of the 16 output connections. Similarly, the 2 physical write ports are shared across 8 data inputs, 7 of which are the FU outputs and 1 input is reserved for inter-cluster communication.

Figure 4 shows the micro-architecture of a function unit. A function unit receives each of its two input operands from the output of an 8 to 1 mux. This multiplexed input provides full forwarding capability within a cluster. Each input operand can originate from the register file or the output stage of an FU within the same cluster. The forwarding mux is controlled by the microcode. Each FU has a pipeline register at its output and the load enable for this register is also software controlled. With the exception of add/sub and logic units, all FUs are internally pipelined and controlled by the compiler. The types of FUs are:

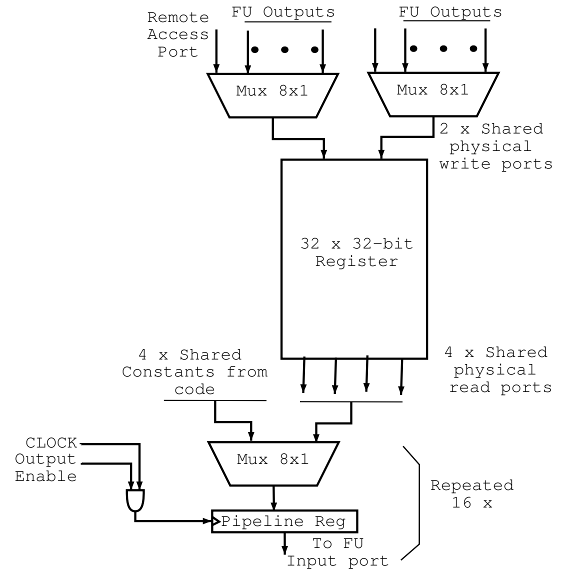

Figure 3: Register File Organization

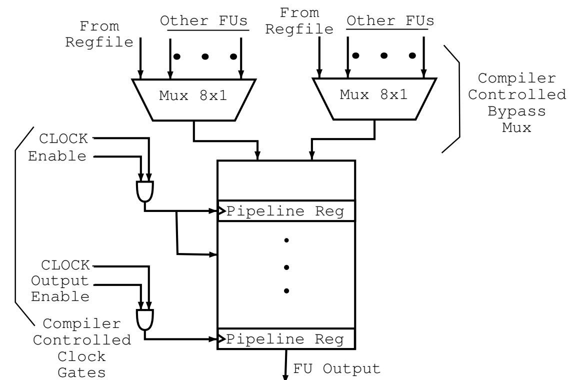

Figure 4: Function Unit Organization

Integer Units perform add, subtract, compare and conditional move with 1 cycle latency. The boolean result of a compare operation can target any of 4 single-bit predicate registers local to the FU. A conditional move operation routes one of the two FU operands to the output port depending on the value of a predicate register.

Logic Units implement and, or, xor, byte select, byte merge, sign extension of constants etc with 1 cycle latency.

Multiply Units implement integer multiplication with 3 cycle latency.
Floating Point Units implement IEEE 754 format multiply, add, subtract and fused multiply add operations with 9 cycle latency. Though inputs and outputs are IEEE 754 format, internally the mantissa precision is reduced to 13 bits. This reduces power consumption by approximately a factor of 4 compared to full precision while still making floating point available to applications that need a large range. The choice of a 13-bit mantissa is motivated by empirical studies of our two floating-point intensive benchmarks (ANN, Gaussian). In both cases, reducing the mantissa from the normal IEEE 754 by 9 bits does not noticeably impact that accuracy of either algorithm [15].

The Load/Store Unit permits the cluster to treat the input, output and scratch SRAMs as a single memory. Two read/write ports are available and these ports appear as two FUs to the other function units. Reads have a 2 cycle latency. The memories are implemented using dual port synchronous SRAMs. Input and output memories are double buffered so that the host can access data concurrently with the cluster. The scratch SRAM does not need to be double buffered. Currently the input and output SRAMs are 2KB and the scratch memory is 8KB in size. These size choices are motivated by the worst case needs of our 5 benchmarks: the largest stream input block is 1.2KB for Gaussian and the Rijndael scratch pad requirement is 5KB for the lookup table. In our process, leakage power is very small. SRAM power consumption is therefore more dependent on the number of sense amps than memory capacity. We chose the next larger power of 2 sizes for these SRAM arrays to simplify addressing. Note that load/store accesses in the cluster can target only the local SRAM. It is not possible for the cluster to access external memory without assistance from the host CPU.

While the architecture, interconnect and compiler are easily and highly customizable, this paper considers only two specific cluster configurations: a floating point cluster and an integer cluster. The function units in each cluster were selected to maximize average throughput of the benchmark suite and to maintain functional generality of the cluster. The integer cluster contains 2 load/store ports, 2 integer units, 2 logic units, and a multiply unit. The floating point cluster contains 3 floating point units, 2 load/store ports, and 2 integer units. The SRAM macro cells for the CMOS process we use has only two ports which motivates the 2 load/store ports per cluster limit. Rijndael is very logic intensive and therefore 2 logic units support this need. HMM and FIR both need a multiplier. ANN and Gaussian are floating point intensive. Two load/store ports are needed to support SRAM access, and 2 integer units are needed to support conditions and basic integer operations. Since the number of function units must be limited to 7, this leaves room for 3 floating point units. Both ANN and Gaussian could benefit from extra floating point units but the loss of performance due to additional multiplexing delays would be unacceptable.

Micro-programs are executed out of a 220-bit wide 256-word single port synchronous memory. These choices are also based on worst case needs of our benchmark suite. While a larger and broader application suite will change these choices, we do not expect the architectural differences to be large since they are more constrained by generality and frequency issues than by algorithmic differences. The exception is the floating point precision issue. The biggest algorithmic sensitivity is that both performance and energy consumption will vary with algorithmic complexity.

2.2 Clock Gating Support

The entire cluster is clock gated at a very fine granularity. This includes both the function units and the output registers of the register file. For function units, there are two separate clock gating signals: one for the final pipeline register and the other for internal pipeline registers. Both signals are controlled by the microcode. This strategy permits power optimization as well as control over the lifetime of values in flight within the cluster. The final pipeline register is clocked only when a value needs to be latched and the value is retained as long as the compiler considers it necessary. This converts the output registers and the bypass muxes into the equivalent of a small (7 entry) register file that the compiler can explicitly address. The compiler uses this fine grain control for two purposes: a) unnecessary value changes on high capacitance forwarding wires are prevented to save power, and b) to create data flows similar to a custom circuit to increase throughput. Paths are dynamically setup and torn down depending on the state of the computation and how well the currently executing section of the algorithm maps onto the cluster. Operands provided by the register file are also gated and controlled by the compiler. Values from the register file and constants from microcode may be placed into the register file output registers and held indefinitely. This facilitates creation of dynamic structures like address generators.

2.3 ASIC Architectures

The ASIC architectures evaluated here use the same 3 SRAM organization shown in Figure 1. The custom coprocessor for GAU consists of a pipeline containing the same reduced precision floating point circuits but organized in to perform $(a-b)^2 * c, \Sigma$ and $a * b + c, \Sigma$ calcula-

tions. Since Σ represents a multi-cycle floating point addition in the final stage of the pipeline, there is a problem with maintaining the flow. This is cured by interleaving the evaluation of 10 separate input vectors. The cost is a trivial increase in storage. The ANN architecture only uses the $a * b + c, \Sigma$ stages, bypassing the earlier stages and uses a similar interleaving strategy.

For Rijndael, we use a publicly available ASIC core from OpenCores. Each encryption S-box is represented as a 256-entry lookup table. The custom hardware uses 16 copies of this lookup table corresponding to 16 KBytes of SRAM resources. It is able to encrypt a 128 bit block in 12 cycles. The cluster version on the other hand is limited to two table lookups because of the paucity of load/store ports.

The accelerator for the FIR uses 16 multipliers that work in parallel, connected to a shift register to do one round of FIR. Since the multipliers have 3 cycle latency, and are fully pipelined, one round of FIR can be done in each cycle.

The ASIC versions provide maximum throughput, but they all use much more chip area than the cluster. They represent an *upper bound* on performance but lack generality.

3 Experimental Method

Our method of evaluation is based on designing hardware for each cluster configuration (the entire organization shown in Figure 1), simulating the hardware at the transistor level using Spice and running the micro-code for our benchmarks during the Spice simulation. Energy consumption is then calculated based on the supply current waveform generated by Spice. The SRAMs we use are macro-cells generated by our 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 10 or more times, but with the same randomly generated input data. We simulate the coprocessor only. The host processor is not simulated.

The function units in a cluster are described in Verilog and Synopsys MCL hardware description languages. The cluster organization and interconnection between clusters 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 wire layer. The Spice model is then simulated with NanoSim, a commercial VLSI tool with Spice-like accuracy. Transistor models and CMOS process parameters are those measured for a test chip built in the same technology. The micro-code corresponding to the benchmark is loaded into program memory and the circuit is simulated in NanoSim/Spice for the duration of several input packets. The RMS current reported by Spice is used to calculate energy consumption.

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. The code is identical to that used by our micro-code compiler. It consists of the algorithm represented in a static single assignment style after fully unrolling all loops.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.

We used a Pentium 4 as the comparison because embedded processors like the StrongARM do not have either the floating point instructions or the performance required for our benchmarks. We believe that software emulated floating point will greatly bloat the energy delay product of the StrongARM and make a meaningful comparison impossible. Another reason for the choice was technical feasibility of measuring processor power. For example, the Intel XScale (StrongARM) development platform we investigated had a processor module board with FPGA, Flash memory etc integrated on it and isolating the processor power was difficult. The particular Pentium 4 system we used was chosen because the layout of the PCB permitted us to de-solder certain components and make modifications to permit measuring the energy consumption of the processor core alone. Hopefully a more meaningful comparison can be made when the Intel Banias processor which is expected to be energy-efficient becomes available later this year.

4 Results

The energy to process one input block obtained from Spice or by actual measurement is used to calculate the energy advantage (ratio) for the cluster and custom circuits over the general purpose processor. The energy was normalized for the CMOS process feature-size, λ, before taking the ratio as described in [6].

Figure 5 shows that the custom approach provides 2 to 3 orders of magnitude improvement over the CPU while the cluster provides more than an order of mag-

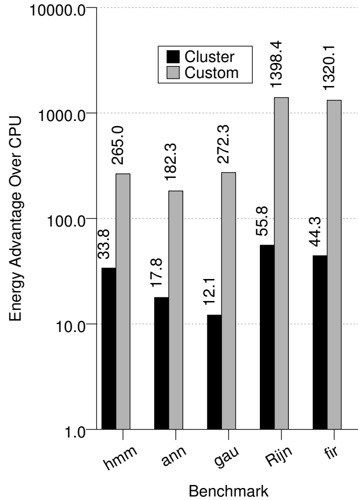

Figure 5: Energy Advantage

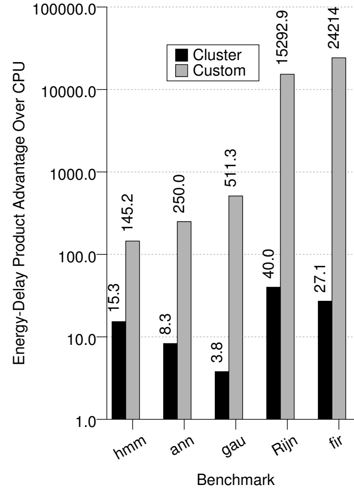

Figure 6: Energy-Delay Product Advantage

nitude advantage while retaining a large fraction of the generality of the general-purpose processor approach. In CMOS circuits, it is often possible to trade energy for increased delay. This is a bad choice when the system has stringent throughput requirements. An effective way of comparing the architectural advantage is to consider the energy-delay product which is a very difficult quantity to improve in the general case [6]. Figure 5 shows the advantage of the cluster and custom architectures over the CPU after normalizing for λ. For the two non-perception benchmarks the custom approach demonstrates a huge improvement over software since these are known to have efficient hardware implementations. The perception algorithms are much more difficult to improve without significant hardware resources. The cluster is able to achieve good improvement with modest resources, e.g. less than 5% of the area of a typical 10mm x 10mm microprocessor die. Cluster performance is limited by 3 factors: a) the scheduling policy of our compiler is several times worse than a manual schedule, b) our floating point units are automatically synthesized from HDL and are 9 times slower than Intel's custom designed FPU, and c) the forwarding paths and register file are more general purpose than they need to be which wastes some power. In later work, we have been able to improve on issues *a* and *c*. Nonetheless, the custom architectures prove that it is possible to do sophisticated perception algorithms at power budgets commensurate with embedded platforms. The cluster approach further demonstrates the possibility of achieving this goal without sacrificing generality within the domain.

5 Related Work

Scheduling tactics for power-efficient embedded processors have achieved reasonably low power operation but they have not achieved the performance/power efficiency of the work described here [8]. Reconfigurability using FPGA devices and hybrid approaches have been explored [2, 3]. 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 but our approach targets custom silicon rather than FPGA devices [16]. Customizing function units in a VLIW architecture has been studied and the Tensilica Xtensa is a commercial instance of this approach [5]. Clock power is often the largest energy culprit in a modern microprocessor [7]. Krashinsky describes the benefits of clock gating [11]. 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.

Tiwari et al have explored scheduling algorithms for less flexible architectures which split an application between a general purpose processor and an ASIC [19]. Lee et al shows instruction scheduling benefits for DSP processors [13]. Eckstein and Krall focus on minimizing the cost of local variable access to reduce power consumption in DSP processors [4]. CALiBeR reduces memory pressure in VLIW systems but cannot directly schedule activities to reduce register file communication at the cluster level [1]. Application specific clusters are investigated in [12]. This complementary scheduler approach minimizes inter-

rather than intra-cluster communication.

Efforts have demonstrated the benefit of VLIW architectures for either customization or power management [18]. Optimization techniques for VLIW architectures using clusters can also be found in [10]. These efforts do not address low-level communication issues. 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 [20]. The RAW work is similar, but is aimed at high performance rather than power efficiency.

6 Conclusions

Combining domain specific execution clusters and compiler directed clock gating can produce high performance low-power accelerators for perception applications in the embedded space. Our approach has demonstrated improvements in the energy delay product from 3.8 to 40 for a range of important benchmarks. Additional work, not presented in this paper, has demonstrated that performance can be quite close to that of an optimized custom circuit if the cluster is scheduled manually. Efforts are underway to incorporate our manual algorithms into the compiler and to make the interconnect architecture more closely resemble the data flow patterns that commonly occur in perception codes. This paper has demonstrated that a cluster based approach makes perception processing possible on embedded architectures at performance levels matching or exceeding high performance processors and at energy levels commensurate with low-power platforms.

References

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

[2] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

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

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

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

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

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

[13] 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.

[14] Mathew, B., Davis, A., and Evans, R. A characterization of visual feature recognition. Tech. Rep. UUCS-03-014, School of Computing, University of Utah, 2003.

[15] 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).

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

[17] 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.

[18] 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.

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

[20] 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. *Computer 30*, 9 (1997), 86–93.