An Energy-Efficient Processor Architecture for Embedded Systems

James Balfour, William J. Dally, *Fellow*, David Black-Schaffer, Vishal Parikh, JongSoo Park
Computer Systems Laboratory, Stanford University, Stanford, CA, USA
{jbalfour,dally}@cva.stanford.edu

***Abstract*— We present an efficient programmable architecture for compute-intensive embedded applications. The processor architecture uses instruction registers to reduce the cost of delivering instructions, and a hierarchical and distributed data register organization to deliver data. Instruction registers capture instruction reuse and locality in inexpensive storage structures that are located near to the functional units. The data register organization captures reuse and locality in different levels of the hierarchy to reduce the cost of delivering data. Exposed communication resources eliminate pipeline registers and control logic, and allow the compiler to schedule efficient instruction and data movement. The architecture keeps a significant fraction of instruction and data bandwidth local to the functional units, which reduces the cost of supplying instructions and data to large numbers of functional units. This architecture achieves an energy efficiency that is $23\times$ greater than an embedded RISC processor.**

***Index Terms*— energy-efficient embedded processor architecture, instruction registers, hierarchical and distributed register organization**

I. Introduction

EMBEDDED applications such as baseband modem processing, digital video compression, and high-definition television display processing exhibit demanding performance and efficiency requirements that continue to increase as more sophisticated communication standards, compression methods, and algorithms are developed. For example, signal processing for a 14.4 Mbps channel in a 3G mobile phone receiver requires 35 – 40 GOPS, whereas a 100 Mbps OFDM channel is estimated to require 210 – 290 GOPS [14]. Commercial embedded processors achieve efficiencies of 4 GOPS/W (250 pJ/op) in a 90 nm CMOS technology [4]. Although more efficient than mobile general purpose processors, which achieve efficiencies of 0.04 GOPS/W (25 nJ/op) [2], the performance and efficiency of embedded processors are inadequate for computationally intensive applications. Consequently, embedded systems use significant amounts of application-specific fixed-function logic to perform computationally demanding tasks, where efficiencies of 200 GOPS/W (5 pJ/op) can be realized in a comparable 90 nm technology [6]. However, the effort required to implement and verify special-purpose logic increases with the scale and complexity of the system, and the design and verification of a complex system-on-chip can require hundreds of engineer-years [10] and incur non-recurring engineering costs in excess of \$20M – \$40M. These high implementation costs deter the development of more sophisticated systems.

Manuscript submitted: 20-Feb-2008. Manuscript accepted: 05-Mar-2008. Final manuscript received: 10-Mar-2008.

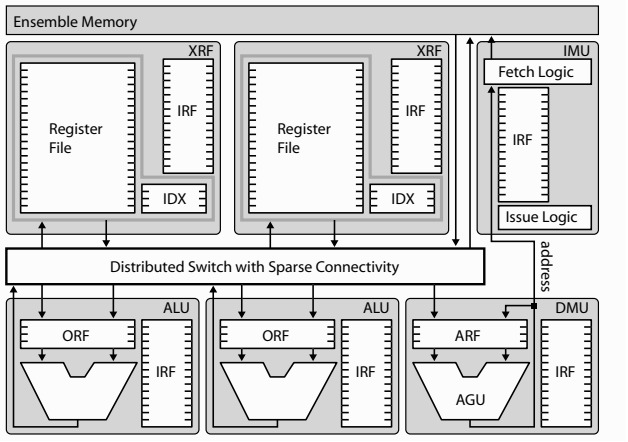

Fig. 1. The Architecture of an Ensemble Processor. Instructions are issued from software-managed instruction registers in the distributed instruction register files (IRFs). The instruction management unit (IMU) coordinates the loading and issuing of instructions. Operands are supplied from the small operand register files (ORFs) and address register files (ARFs) that are distributed among the functional units. Indexed register files (XRFs) provide an intermediate storage between the ORFs and Ensemble Memory. The data management unit (DMU) executes load and store instructions, and coordinates the movement of data between registers and the local Ensemble Memory.

Energy consumption in modern processors is dominated by the supplying of instructions and data to functional units. Because interconnect benefits less than logic from advances in semiconductor technologies, driving interconnect accounts for an increasing fraction of the energy consumed in integrated circuits. Interconnect, which dominates the memories and buses used to store and transfer instructions and data, can account for more than 70% of the energy expended in a processor. Programmable architectures for compute-intensive embedded applications require many functional units to achieve performance requirements. While advances in semiconductor technology have made it possible to integrate large numbers of functional units in a system, the lack of efficient mechanisms for delivering instructions and data to the functional units has prevented programmable systems from satisfying both performance and efficiency requirements. This paper introduces an efficient programmable architecture for compute-intensive embedded applications. The processor used in this architecture achieves energy efficiencies that are $23\times$ greater than an embedded RISC processor, and which approach within $1.5\times$ of ASIC efficiency on computationally intensive tasks. The following section describes the architecture of an efficient embedded processor, focusing on efficient data and instruction supply organizations. An evaluation of the architecture follows. Finally, we address related work and conclude.

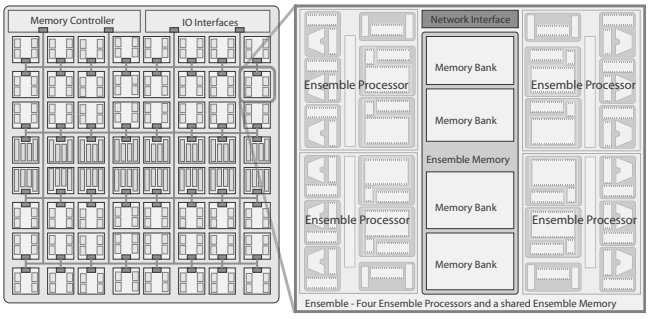

Fig. 2. Embedded System Architecture. The system is composed of a distributed collection of Ensemble Processors. Clusters of four neighboring Ensemble Processors are grouped into an Ensemble. The processors within an Ensemble share a local software-controlled memory and an interface to the global on-chip interconnection network. The Ensemble Processors keep instructions and data close to the distributed functional units. Distributed memory tiles capture large working sets. The Ensemble Memory is used to stage instruction and data transfers, and allows moderate working sets to be captured within an Ensemble. Communication resources include a global on-chip network for transferring instructions and data and low-latency point-to-point links between neighboring Ensemble Processors.

TABLE I
PROCESSOR CONFIGURATIONS AND DETAILS

Ensemble Processor			
Technology	TSMC CL013G (V_{DD}=1.2V)		
Clock Frequency	200 MHz		
Average Power	28 mW		
Multipliers	16-bit + 40-bit acc.	16.5 pJ/op	
IRFs	64 128-bit registers	16 pJ/read	18 pJ/write
XRFs	32 32-bit registers	14 pJ/read	8.7 pJ/write
ORFs	8 32-bit registers	1.3 pJ/read	1.8 pJ/write
ARF	8 16-bit registers	1.1 pJ/read	1.6 pJ/write
Ensemble Memory	8KB	33 pJ/read	29 pJ/write
RISC Processor			
Technology	TSMC CL013G (V_{DD}=1.2V)		
Clock Frequency	200 MHz		
Average Power	72 mW		
Multiplier	16-bit + 40-bit acc.	16.5 pJ/op	
Register File	40 32-bit registers	17 pJ/read	22 pJ/write
Instruction Cache	8KB (2-way)	107 pJ/read	121 pJ/write
Data Cache	8KB (2-way)	131 pJ/read	121 pJ/write

II. ARCHITECTURE

This section describes the architecture of an efficient embedded processor, which we refer to as an Ensemble Processor because of where it appears in the system organization. Fig. 1 provides a block diagram of the Ensemble Processor. Exposed instruction and data storage resources are distributed among the functional units. Communication resources are exposed through the sparse, distributed switch that transfers results from the functional units to distributed register files. The exposed architecture allows the compiler to explicitly manage the movement of instructions and data through the storage hierarchies to reduce movement. At the system level, the compiler explicitly schedules computation and instruction and data movement, which provides predictability as well as significant control over how computation proceeds. Fig. 2 illustrates how an embedded system is built from a collection of Ensemble Processors. Table I compares the access energies of the instruction and data storage hierarchies with those of an embedded RISC processor; the cache access energies include accessing the tag and memory arrays. The following sections describe in detail the data and instruction supply organizations.

A. Data Supply

The distributed and hierarchical data register organization exploits reuse and locality in computations to satisfy most references from the operand register files (ORFs) located at the inputs of the functional units. These small (4-entry) register files capture short-term data locality to keep a significant fraction of the data bandwidth local to each functional unit. Placing the ORFs within the functional units reduces the cost of transferring operands and results between registers and functional units, particularly when a result is consumed locally within the functional unit that produced it. The ORFs are preceded by a switch that forwards data produced in remote functional units to the local operand registers. Exposing the distributed ORFs to the compiler allows it to place data close to the functional units that consume them, which reduces expensive data movements. Operands are read from the ORFs in the *same* cycle in which the instruction that consumes then executes, which reduces the effective pipeline depth and eliminates pipeline registers. The pipeline register at the output of the ALUs is exposed as a register in the local ORF. This convention eliminates the need to allocate registers for storing values that are consumed immediately after being produced, which relieves register pressure in the ORFs. The value of the result register is held when NOPs execute to simplify scheduling.

The centralized collection of indexed register files (XRFs) forms the second level of the register hierarchy. Their capacity allows them to capture the next level of the working set, and to exploit locality and reuse over longer intervals than can be captured in the ORFs. Although more expensive to access than the ORFs, the XRFs satisfy fewer operand references, and their capacity improves efficiency by filtering memory references. If the ORFs were not present in the data register hierarchy, the XRFs would require additional ports to directly supply operands to the functional units. The resulting register files would be less area efficient and more expensive to access, and would require more instruction bits for control.

Registers in the XRFs can be accessed indirectly through index registers in the IDX units. The index registers can be configured to update after each reference, which allows the XRFs to be operated as software-managed vector or streaming register files. The DMU can use the index registers to autonomously transfer blocks of data between registers and the Ensemble Memory, which improves the efficiency of moving data through the register hierarchy. The index registers can be used to implement structures such as circular buffers in the XRFs without requiring loop unrolling, which increases pressure on the instruction registers.

Load and store instructions execute in the DMU, which coordinates data movement between the register hierarchy and the Ensemble Memory. The address generation unit (AGU) executes basic arithmetic operations. Operands are supplied from address registers in the local address register file (ARF). The AGU datapath and registers are designed for address computations to reduce the cost of executing memory operations. A subset of the address registers can be configured to automatically update after each use. The update actions, which include auto-increment with wrap, improve efficiency by accelerating sequences of operations that appear in address calculations, and by reducing the number

of instructions that need to be issued to calculate the address of a load or store. Load and store instructions can specify a repeat count to produce the effect of issuing the instruction to the DMU multiple times. A single instruction can produce a sequence of loads or stores at different addresses by using an address register with an update policy. This allows a single instruction to initiate the transfer of block of data between the XRFs and Ensemble Memory. The compiler converts complex memory operations, such as vector moves, scatters, and gathers, into sequences of repeated instructions that operate on address registers with automatic update actions.

B. Instruction Supply

Instructions are issued from software-controlled instruction registers (IRs). The IRs capture reuse and locality within the kernels that dominate embedded applications. The IRs are organized as shallow (64-entry) instruction register files (IRFs) to keep the cost of reading instructions low, and the register files are distributed among the functional units to reduce the cost of transferring instructions to control points. The exposed pipeline allows instruction bits to be transferred directly to control points without propagating through pipeline registers. When ILP is low, NOPs are generated by clock-gating within the functional units and control logic in the IMU inhibits the reading of inactive IRFs. Because most instruction references are satisfied by IRs, the Ensemble Memory can be used to stage instructions for all of the processors within an Ensemble without degrading performance.

Instructions in the Ensemble Memory are stored as instruction blocks, sequences of instructions that are loaded into the IRs as a group. Instruction blocks may enclose multiple basic blocks and may overlap. The IMU maintains an Instruction Presence Vector (IPV), which records which IRs contain current instructions, to allow the loading of an instruction block to proceed while instructions are issued from the IRs. The location of the next instruction to be issued from the IRs is designated by the Instruction Counter (IC), which is maintained in the IMU. The IC functions as an abridged PC, which is implicitly defined by the IC and the instruction block from which instructions are being issued. Code may branch arbitrarily within the IRs. Programmable loop counters are available to eliminate loop overhead code. Because an instruction must reside in a register to be issued, branch targets must be explicitly loaded before the branch instruction transfers control.

Certain instructions are modified as they are loaded into the IRs. Because the IRs define a unique address space within each processor, instruction addresses must be translated from the Ensemble Memory address space to the IR address space. To allow position-independent code to execute, branch instructions may name IRs using relative, position-dependent addresses. Accordingly, branch instructions in the Ensemble memory may specify either absolute or relative IR positions. Relative positions are resolved as instructions are loaded and their load positions bound. Consequently, the address generator that calculates the targets of relative branches, which would conventionally be located in the instruction fetch stage, appears between the Ensemble Memory and IRF. Because an instruction is usually executed multiple times per load, this organization reduces activity in the logic that generates branch targets.

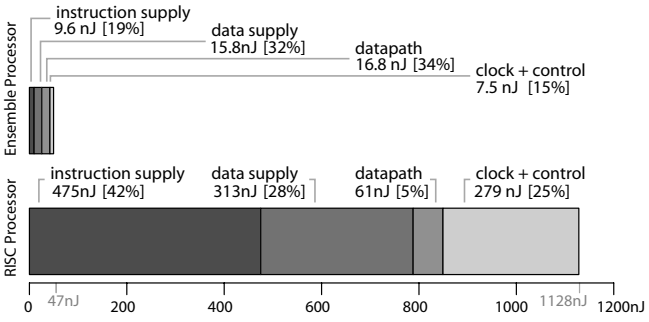

Fig. 3. Processor Energy – Harmonic Mean over Kernels. The RISC processor consumes $23\times$ more energy. The data register organization reduces the cost of supplying operands by $17\times$, and instruction registers reduce the cost of supplying instructions by $49\times$. The address and index registers eliminate overhead instructions, which improves datapath utilization and contributes to the $3.6\times$ increase in datapath efficiency because fewer instructions are executed. The exposed pipeline eliminates pipeline registers, which reduces the clock load, and pipeline control logic, such as the comparators that control the bypass network in the RISC datapath.

III. Evaluation

To illustrate how an Ensemble Processor achieves efficiency, we compare its efficiency to that of an embedded RISC processor. The RISC processor is derived from a synthesizable embedded processor with a SPARC V8 compliant 32-bit integer unit [3]. The integer pipeline, caches, and pipeline control logic were retained and optimized for the comparison; all other components, such as the MMU and TLB, and the associated control logic were removed. Both processors implement equivalent 32-bit arithmetic and logic instructions with identical latencies. For comparison, we estimate that the RISC processor would achieve an efficiency of 180 pJ/op when implemented in a 90 nm technology. Table I summarizes the configurations used in the evaluation.

Energy consumption was measured when executing the following seven kernels on gate-level netlists of the processors: AES encryption, two-dimensional convolution filtering, DCT, FIR filtering, CRC32 calculation, Huffman decoding, and Viterbi decoding. The netlists were generated from RTL models of the processors; the processors were synthesized, placed, and routed using the same design flow. Results are presented for hand-coded assembly. Similar efficiency trends are observed when the kernels are compiled, with both processors exhibiting a $1.7\times$ reduction in average throughput and efficiency. Fig. 3 presents the harmonic mean of the energy expended executing the kernels and explains how the Ensemble Processor achieves an energy efficiency that is $23\times$ greater than the RISC processor. Fig. 4 and Fig. 5 provide addition details about the energy expended supplying data and instruction. To provide context, an ASIC version of the FIR filter, implemented in the same technology and using the same design flow, is $1.5\times$ more energy-efficient than the Ensemble Processor and $72\times$ more efficient than the RISC processor.

IV. Related Work

Data parallel architectures such as Imagine [12] and SCALE [8] amortize instruction issue and control overhead by issuing the same instruction to multiple functional units. However, the efficiency of data parallel architectures declines when applications lack sufficient SIMD parallelism. The efficiencies of tiled architectures, such as RAW [15], are limited by the RISC processors from which they are constructed; the additional ALUs increase

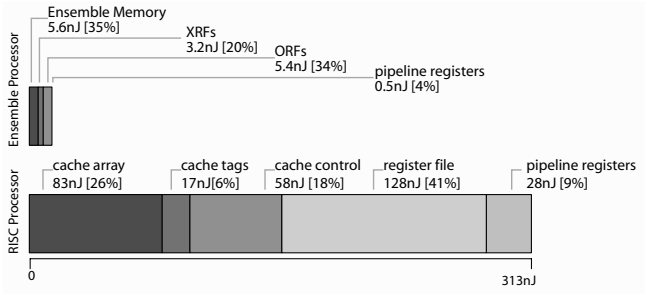

Fig. 4. Data Supply Energy – Harmonic Mean over Kernels. The small, inexpensive ORFs eliminate references to more expensive levels of the data supply hierarchy, which results in the $15\times$ reduction in register file energy. The XRFs are able to capture indirect access patterns that would normally require data be allocated in memory, which contributes to the $15\times$ reduction in memory array energy. The exposed pipeline achieves a $56\times$ reduction in the energy expended forwarding operands and results through pipeline registers.

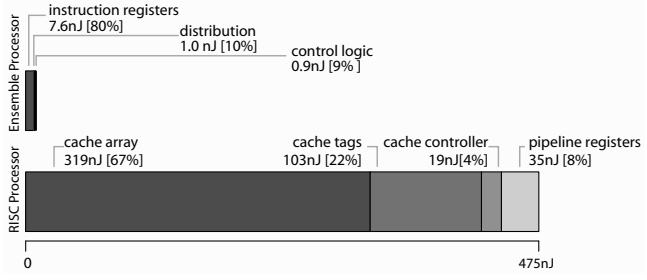

Fig. 5. Instruction Supply Energy – Harmonic Mean over Kernels. The instruction cache dominates in the RISC processor. The instruction registers, which are significantly less expensive to access, reduce the energy expended fetching instructions by $58\times$. Placing the IRFs within the functional units and exposing the pipeline eliminates the pipeline registers that are used to distribute instruction bits to control points in the RISC processor, further reducing the cost of delivering instructions to the functional units.

throughput, but efficiency is not improved beyond a conventional embedded processor.

Unlike reactive mechanisms such as filter caches [7], instruction registers are software-managed to reduce instruction movement, and are distributed to reduce the cost of transferring instructions to control points. Loop buffers [9], which are loaded on backwards branches, require an instruction cache to handle misses, and cannot capture complicated control flow. Pre-loaded loop caches [1] require a dynamic instruction cache to execute code outside of the loop cache. Instruction register loads are scheduled in parallel with execution to avoid stalls when instructions are brought into the lowest level of the instruction storage hierarchy. The instruction registers described in [5] are dissimilar as they provide a limited form of code compression and cannot be loaded while code executes. The register organization described in this work differs from other hierarchical [16] and distributed [13] organizations by distributing the register files at every level of the hierarchy and by using small (4-entry) register files, each associated with and located within a single functional unit, at the lowest level. Like index registers, the vector pointer registers described in [11] can be automatically updated when used to access a register file. The register organization described in this work differs because it allows the XRF registers to be directly accessed, bypassing the index registers, when data are accessed irregularly; this allows the XRF registers to hold working sets that contain both scalar and vector operands.

V. Conclusion

This paper has described the architecture of an efficient processor for compute-intensive embedded applications. Instruction registers, a distributed and hierarchical data register organization, and exposed pipelines reduce the cost of supplying instructions and data to functional units so that many programmable processors can be integrated in a system. The architecture described in this paper achieves energy efficiencies that are $23\times$ greater than embedded RISC processors, and can approach within $1.5\times$ of ASIC efficiency on computationally demanding tasks such as FIR filtering. Leveraging custom circuit design techniques will allow us to reduce or eliminate the energy efficiency advantages ASIC implementations exhibit for many embedded tasks. We have developed a compiler, and are implementing the programming systems for mapping entire systems onto our architecture.

References

[1] A. Gordon-Ross, S. Cotterell, and F. Vahid, "Tiny instruction caches for low power embedded systems," *Transactions on Embedded Computer Systems*, vol. 2, no. 4, pp. 449–481, 2003.

[2] E. Grochowski and M. Annavaram, "Energy per instruction trends in Intel microprocessors," *Technology@Intel Magazine*, pp. 1–8, Mar. 2006.

[3] S. Habnic and J. Gasisler, "Status of the LEON2/3 processor developments," in *DASIA '07: DAta Systems In Aerospace 2007*. The Association of European Space Industry, 2007.

[4] T. R. Halfhill, "MIPS threads the needle," *Microprocessor Report*, February 2006.

[5] S. Hines, G. Tyson, and D. Whalley, "Reducing instruction fetch cost by packing instructions into register windows," in *Proc. of ACM/IEEE International Symposium on Microarchitecture*, 2005.

[6] S. Hsu, V. Venkatraman, S. Mathew, H. Kaul, H. Anders, S. Dighe, W. Burleson, and R. Krishnamurthy, "A 2GHz 13.6mW 12×9b multiplier for energy efficient FFT accelerators," in *Proc. of the 31st European Solid-State Circuits Conference*, 2005, pp. 199–202.

[7] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *MICRO 30: Proc. of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 184–193.

[8] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *ISCA '04: Proc. of the 31st annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 52.

[9] L. H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *ISLPED '99: Proc. of the 1999 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 1999, pp. 267–269.

[10] Y. Mathys and A. Châtelain, "Verification strategy for integration 3G baseband SoC," in *DAC '03: Proc. of the 40th conference on Design automation*. New York, NY, USA: ACM Press, 2003, pp. 7–10.

[11] J. H. Moreno *et al.*, "An innovative low-power high-performance programmable signal processor for digital communications," *IBM J. Res. Dev.*, vol. 47, no. 2-3, pp. 299–326, 2003.

[12] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *MICRO 31: Proc. of the 31st annual ACM/IEEE international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 3–13.

[13] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *HPCA 6: Proc. of the Sixth International Symposium on High-Performance Computer Architecture*, 2000, pp. 375–386.

[14] O. Silven and K. Jyrkkä, "Observations on power-efficiency trends in mobile communication devices," *EURASIP Journal of Embedded Systems*, vol. 2007, no. 1, pp. 17–17, 2007.

[15] M. B. Taylor, *et al.*, "Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams," in *ISCA '04: Proc. of the 31st annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 2.

[16] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, "Two-level hierarchical register file organization for VLIW processors," in *MICRO 33: Proc. of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. New York, NY, USA: ACM Press, 2000, pp. 137–146.