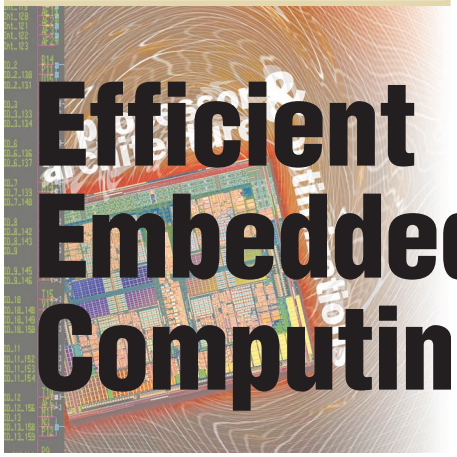

Efficient Embedded Computing

William J. Dally, James Balfour, David Black-Shaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield, Stanford University

Hardwired ASICs—50X more efficient than programmable processors—sacrifice programmability to meet the efficiency requirements of demanding embedded systems. Programmable processors use energy mostly to supply instructions and data to the arithmetic units, and several techniques can reduce instruction- and data-supply energy costs. Using these techniques in the Stanford ELM processor closes the gap with ASICs to within 3X.

Embedded computing applications demand both efficiency and flexibility: The bulk of computation today happens not in desktops, laptops, or data centers, but rather in embedded media devices. More than one billion cell phones are sold each year, and a 3G cell phone performs more operations per second than a typical desktop CPU.

Media devices like cell phones, video cameras, and digital televisions perform more computations than all but the fastest supercomputers at power levels orders of magnitude lower than general-purpose desktop and laptop machines. For example, a 3G mobile phone receiver requires 35 to 40 giga operations per second (GOPS) of performance to handle a 14.4-Mbps channel, and researchers estimate the requirements for a 100-Mbps orthogonal frequency-division multiplexing (OFDM) channel at between 210 and 290 GOPS.[1]

In contrast, a typical desktop computer system has a peak performance of a few GOPS and sustains far less on most applications. A cell phone's computing challenges are even more impressive when we consider that these performance levels must be achieved in a small handheld package with a maximum power dissipation of about 1W. Simple arithmetic gives a required efficiency of 25 mW/GOP or 25 pJ/op for the 3G receiver and 3-5 pJ/op for the OFDM receiver.

Demanding performance and efficiency requirements drive most media devices to perform their computations with hardwired logic in the form of an application-specific integrated circuit. A carefully designed ASIC can achieve an efficiency of 5 pJ/op in a 90-nm CMOS technology.[2] In contrast, very efficient embedded processors and DSPs require about 250 pJ/op[3] (50X more energy than an ASIC), and a popular laptop processor requires 20 nJ/op[4] (4,000X more energy than an ASIC). The efficiencies of these programmable processors is simply inadequate for demanding embedded applications—forcing designers to use hardwired logic to keep energy demands within limits.

While ASICs meet the energy-efficiency demands of embedded applications, they are difficult to design and inflexible. It takes two years to design a typical ASIC, and the cost is $20 million or more. This high cost places ASIC efficiency out of reach for all but the highest-volume applications. The long design cycle causes ASICs to lag far behind the latest developments in algorithms, modems, and codecs. Inflexibility increases an ASIC's area and complexity. If a system must support several air interfaces, for example, an ASIC implementation instantiates separate hardwired modems for each interface—even though only one will be used at any time. If it meets the efficiency requirement, a programmable processor can use a single hardware resource to implement all the interfaces by running different software.

As media applications evolve and become more complex, the problems of ASICs become larger. The

0018-9162/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

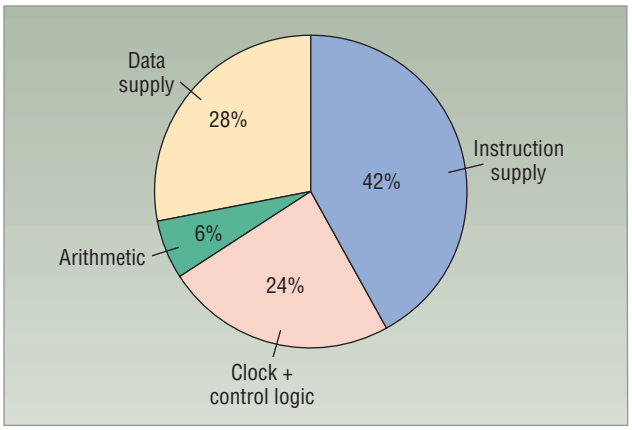

Figure 1. Embedded processor efficiency. Supplying data and instructions consumes 70 percent of the processor's energy; performing arithmetic consumes only 6 percent.

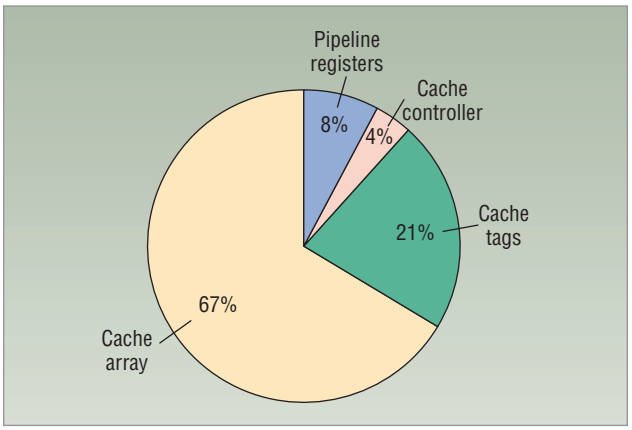

Figure 2. Instruction-supply energy breakdown. The 8-Kbyte instruction cache consumes the bulk of the energy, while fetching each instruction requires accessing both directions of the two-way set-associative cache and reading two tags.

increasingly complex applications are harder to implement as hardwired logic and have more dynamic requirements—for example, different modes of operation. Algorithms are also evolving more rapidly, making it problematic to freeze them into hardwired implementations. Increasingly, embedded applications are demanding flexibility as well as efficiency.

An embedded processor spends most of its energy on instruction and data supply. Thus, as a first step in developing an efficient embedded processor, seeing where the energy goes in an efficient embedded processor can be instructive. Figure 1 shows that the processor consumes 70 percent of the energy supplying data (28 percent) and instructions (42 percent). Performing arithmetic consumes only 6 percent. Of this, the processor spends only 59 percent on *useful* arithmetic—the operations the computation actually requires—with the balance spent on overhead, such as updating loop indices and calculating memory addresses. The energy spent on useful arithmetic is similar to that spent on arithmetic in the hardwired implementation: Both use similar arithmetic units.

A programmable processor's high overhead derives from the inefficient way it supplies data and instructions to these arithmetic units: for every 10-pJ arithmetic operation (a weighted average of 4 pJ adds and 17 pJ multiplies), the processor spends 70 pJ on instruction supply and 47 pJ on data supply. This overhead is even higher, though, because 1.7 instructions must be fetched and supplied with data for every useful instruction.

Figure 2 shows a further breakdown of the instruction supply energy. The 8-Kbyte instruction cache consumes most of the energy. Fetching each instruction requires accessing both *ways* of the two-way set-associative cache and reading two tags, at a cost of 107 pJ of energy.

Table 1 lists each component's energy costs. Pipeline registers consume an additional 12 pJ, passing each instruction down the five-stage RISC pipeline. Thus the total energy of supplying each instruction is 119pJ to control a 10-pJ arithmetic operation. Moreover, because of overhead instructions, 1.7 instructions must be fetched for each useful instruction.

Figure 3 shows the breakdown of data supply energy. Here the 8-Kbyte data cache (array, tags, and control) accounts for 50 percent of the data supply energy. The 40-word multiported general-purpose register file accounts for 41 percent of the energy, and pipeline registers account for the balance. Supplying a word of data from the data cache requires 131 pJ of energy; supplying this word from the register file requires 17 pJ of energy. Two words must be supplied and one consumed for every 10-pJ arithmetic operation.

Thus, the energy required to supply data and instructions to the arithmetic units in a conventional embedded RISC processor ranges from 15 to 50 times the energy of actually carrying out the instruction. It is clear that to improve the efficiency of programmable processors we must focus our effort on data and instruction supply.

Instruction supply energy can be reduced 50X by using a deeper hierarchy with explicit control, eliminating overhead instructions, and exposing the pipeline. Since most of the instruction-supply energy cycles an instruction cache, to reduce this number the processor must supply instructions without cycling a power-hungry cache. As Figure 4 shows, our efficient low-power microprocessor (ELM) supplies instructions from a small set of distributed instruction registers rather than from the cache. The cost of reading an instruction bit from this instruction register file (IRF) is 0.1 pJ versus 3.4pJ for the cache, a reduction of 34X.

In many ways, the IRF is just another, smaller, level of the instruction memory hierarchy, and we might ask why such a level has not been included in the past. Historically, caches were used to improve performance, not

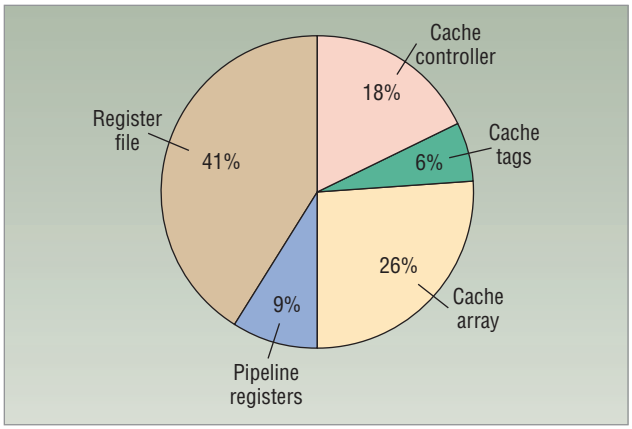

Figure 3. Data-supply energy breakdown. The 8-Kbyte data cache—array, tags, and control—accounts for 50 percent of the data-supply energy. The 40-word multiported general-purpose register file accounts for 41 percent of the energy; pipeline registers account for the balance.

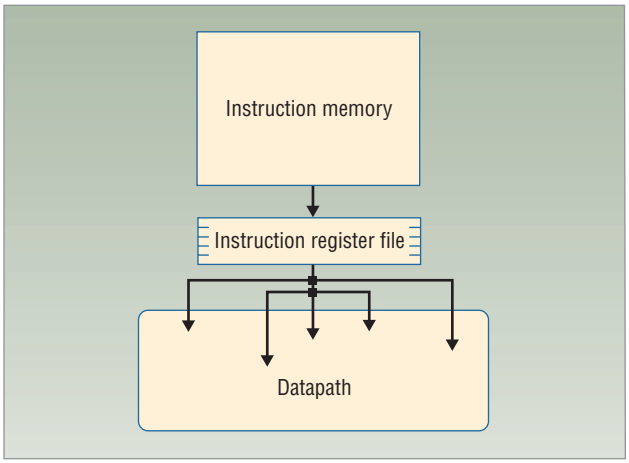

Figure 4. Stanford efficient low-power microprocessor. The processor supplies instructions from a small set of distributed instruction registers rather than from the cache.

to reduce energy. To maximize performance, the hierarchy's lowest level is sized as large as possible while still being accessible in a single cycle. Making the cache smaller would only decrease performance by increasing the miss rate, without affecting cycle time. For this reason, level 1 instruction caches are typically 8 to 64 Kbytes. Optimizing for energy requires minimizing the hierarchy's bottom level while still capturing the critical loops of the kernels that dominate media applications. The ELM has an IRF with 64 registers that can be partitioned so that smaller loops need only cycle the registers' bit lines as needed to hold the loop.

The ELM processor manages the IRF as a register file, with the compiler allocating registers and performing transfers—not as a cache, where hardware performs allocation and transfers reactively at runtime. Explicit management of the IRF has two main advantages. First, it avoids stalls by prefetching a block of instructions into the IRF as soon as the processor identifies the block to be executed. In contrast, a cache waits until the first instruction is needed, then stalls the processor while it fetches the instruction from backing memory. Some caches use hardware prefetch engines to avoid this problem, but they burn power, often fetch unneeded instructions, and rarely anticipate branches off the straight-line instruction sequence. In addition to being more efficient, explicit management better manages cases where the working set does not quite fit in the IRF.

With an explicitly managed IRF reducing the cost of fetching each bit of instruction from 3.4 pJ to 0.1 pJ, the 0.4-pJ cost of moving this bit down the pipeline now appears large. The ELM processor eliminates these pipeline instruction registers by exposing the pipeline. With a conventional, hidden pipeline, the processor fetches an instruction that takes many cycles all at once, then delays it via a series of pipeline registers

Table 1. Storage hierarchies.

RISC instruction cache	**8 Kbytes (2-way)**
Read – tags	26
Read – array	81
Read – total	107 pJ
RISC data cache	**8 Kbytes (2-way)**
Read – tags	26
Read – array	81
Read – total	107 pJ
Write – tags	27
Write – array	94
Write – total	121 pJ
RISC register file [2R + 1W]	**40 x 32-bit**
Read	17 pJ
Write	22 pJ
ELM instruction memory	**8 Kbytes**
Read 128-bits	66 pJ
ELM instruction registers	**64 x 128-bit**
Read 128-bits	16 pJ
Write 128-bits	18 pJ
ELM data memory	**8 Kbytes**
Read	33 pJ
Write	29 pJ
ELM XRF [1R + 1W] – 2 files	**16 x 32-bit**
Read	14 pJ
Write	9 pJ
ELM ORF [2R + 2W] – 1 file per ALU	**4 x 32-bit**
Read	1.3 pJ
Write	1.8 pJ

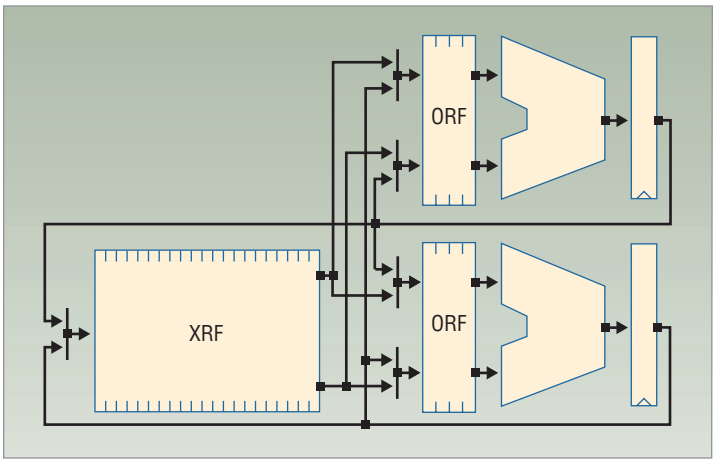

Figure 5. Reducing data-supply energy. The ELM reduces the data-supply energy by placing a small, four-entry operand register file (ORF) on each ALU's input and by using an indexed register fille (XRF).

until each part is needed. The system uses the register read addresses during the register read pipeline stage, the opcode during the execute pipeline stage, and so on. This is convenient but costly in terms of energy. An exposed pipeline splits up instructions and fetches each part of the instruction during the cycle when it is needed. This requires a little more bookkeeping on the compiler's part, but eliminates the power-hungry instruction pipeline.

The IRF and exposed pipeline reduce the cost of supplying each instruction bit. Eliminating overhead instructions reduces the number of instruction bits the system needs to supply. The processor uses most overhead instructions to manage loop indices and calculate addresses. We modify the instruction set so that the system performs the most common cases of these overhead functions automatically—as side effects of other instructions. For example, our load instruction allows loading a word from memory with the address postincremented by a constant and wrapped to a start address when it reaches a limit. This allows implementing a circular buffer using a single load instruction rather than a sequence of five instructions.

Adding these side effects to instructions represents a selective return to complex instruction sets. When energy is the major concern, such CISC constructs make sense, and the ELM architecture introduces them in ways that make them easy for an optimizing compiler to use. Over our suite of benchmarks, the ELM fetched 63 percent fewer dynamic instruction bits than the bits fetched for the RISC.

Data supply energy can be reduced 21X by using a deeper storage hierarchy with indexed register files: Most data in the RISC processor is supplied from the general register file at a cost of 17 pJ/word. Accessing this register file is costly because of its 40-word size and multiple read and write ports. As Figure 5 shows, the ELM reduces the data supply energy by placing a small, four-entry operand register file (ORF) on each arithmetic logic unit's (ALU's) input. Because of its small size and port count, reading a word from an ORF requires only 1.3 pJ, a savings of 13X. The ORF's small size also allows reading the ORF in the same clock cycle as the arithmetic operation, without appreciably lengthening the cycle time. This helps simplify the exposed pipeline.

Figure 5 also shows the use of an indexed register file (XRF) as the hierarchy's next level. The system can access the XRF either directly—a field of the instruction specifies the register to access—or indirectly—a field of the instruction specifies an index register which in turn specifies the register to access. Allowing indirect, or indexed, access to this register file eliminates the need for many references to the data cache or memory. Many media kernels are characterized by accesses to small arrays that can fit in the register file but require indirect access. On ELM, these arrays can be kept in the indexed register file, greatly reducing their access energy. Across our benchmark suite, the ELM data memory reads 77 percent fewer words than the RISC data cache.

COMPILATION FOR EXPOSED COMMUNICATION ARCHITECTURES

Exposing the movement of instructions and data with IRFs, ORFs, and XRFs requires that the compiler perform many new tasks. In particular, it must manage the transfer of instruction blocks into the IRFs; coordinate the movement of data between XRFs, ORFs, and ALUs; and map arrays into XRFs. The exposed pipeline creates new compilation challenges, particularly at the boundaries of basic blocks where the compiler can overlap operations from multiple blocks—but lets the compiler precisely control the movement of data through the data path and avoid unnecessarily cycling data through costly register files. While not part of conventional compilers, all these tasks are well within the reach of current technology.

To quantify our current compiler's efficiency, we ran our benchmark suite using only compiled code, then compared the results to hand-optimized code for both the ELM and RISC processors. The results showed a degradation in performance of 1.7X for both the ELM and RISC when moving from hand-scheduled code to compiled code. Much of the performance degradation on the ELM processor results from the current compiler performing only basic scheduling optimizations at the boundaries of basic blocks; the current scheduling algorithms sometimes introduce brief bubbles in the pipeline when a branch instruction jumps to a new instruction.

CLOSING THE GAP

To quantify how closely the ELM processor approaches the ultimate goal of ASIC efficiency, we compared our processor to ASIC implementations of several kernels in our benchmark suite. These implementations used the same technology and design flow. On kernels such as AES encryption and discrete cosine transfer computation, where the ELM processor stores part of the data working set in its local memory, the ELM processor consumes about 3X the energy of an ASIC. On compute-intensive kernels such as FIR filtering, where the data register hierarchy captures the working sets, the ELM processor consumes no more than 1.5X the energy of an ASIC.

These results are promising. We chose the arithmetic operations and register hierarchies implemented in the ELM processor to allow a fair comparison against an embedded RISC processor, and room remains for further optimization along these two dimensions. Despite this, the ELM processor's efficiency is close enough to that of an ASIC for us to expect that we can close the remaining gap using a combination of minor improvements to the ISA and microarchitecture along with more efficient custom circuits and layout, particularly in the instruction and data storage hierarchies.

THE FUTURE IS COOL

Increasingly complex, modern media applications demand high performance and low power. Historically, developers have used hardwired logic to meet these performance and power demands. The increasing complexity and fixed costs of ASICs, however, call for a programmable solution. Conventional programmable processors do not have the efficiency required for these applications because of the energy they consume to supply data and instructions to arithmetic units. Our ELM processor optimizes instruction and data supply to improve energy efficiency by 23X compared to an embedded RISC processor, while closing the gap with ASICs from 1.5X to 3.0X.

The ELM design represents only a starting point in our quest for more energy-efficient programmable processing. Considerable opportunities for additional energy savings exist. In the area of instruction supply, we can add additional levels to the hierarchy, factor instructions so that common instruction parts can be shared between instructions, and compress out no-operation fields of instructions. We can view instruction supply as a data- or instruction-compression problem. However, rather than trying to represent an instruction stream with the minimum number of bits, we seek to deliver the dynamic instruction stream with a minimum amount of energy.

Opportunities exist to improve data-supply energy as well. For example, we can construct compound operations that perform several instructions as a unit without incurring the cost of cycling intermediate data through even the smallest of register files.

While aimed at embedded computing, the techniques used in the ELM might be valuable in reducing energy in other applications. For example, the servers used in large data centers are rapidly becoming power limited. The energy cost of operating these machines usually exceeds their purchase price within two years. In 2007, the US expended one percent of its electricity supply operating large servers. The applications that run on these servers differ markedly from the embedded media-processing applications we have considered. In particular, they have far less instruction locality, making IRFs less attractive. Even so, some of the same techniques might apply and could yield significant power savings.

Whether in embedded processors, data centers, or personal computers, the world of computing is becoming energy limited. By carefully focusing on where energy is consumed in computers—instruction supply and data supply—we can make these machines much more efficient, ensuring that future computing is cool in both senses of the word.

References

1. O. Silven and K. Jyrkkä, "Observations on Power-Efficiency Trends in Mobile Communication Devices," *EURASIP J. Embedded Systems*, vol. 2007, no. 1, 2007, p. 17.
2. S. Hsu et al., "A 2GHz 13.6mW 12x9b Multiplier for Energy Efficient FFT Accelerators," *Proc. 31st European Solid-State Circuits Conf.*, IEEE Press, 2005, pp. 199-202.
3. T.R. Halfhill, "MIPS Threads the Needle," *Microprocessor Report*, Feb. 2006, vol. 20, part 2, pp. 1-8.
4. E. Grochowski and M. Annavaram, "Energy per Instruction Trends in Intel Microprocessors," *Technology@Intel Magazine*, Mar. 2006, pp. 1-8.

William J. Dally *is the Bell Professor of Engineering and chairman of the Computer Science Department at Stanford University and founder, chairman, and CTO of Stream Processors. His research interests include computer architecture, interconnection networks, compilers, and circuit design. Dally received a PhD in computer science from Caltech. Contact him at dally@stanford.edu.*

James Balfour *is a PhD candidate at Stanford University. His research interests include computer architecture, compilers, interconnection networks, and circuit design. Balfour received an MS in electrical engineering from Stanford University. Contact him at jbalfour@cva.stanford.edu.*

David Black-Shaffer *received a PhD in electrical engineering from Stanford University. His research focused on programming systems for embedded applications on many-core processors and efficient instruction delivery. He will be joining the GPGPU group at Apple Computer to work on high-performance programming systems. Contact him at davidbbs@cva.stanford.edu.*

James Chen *is a PhD candidate at Stanford University. His research interests include energy-efficient circuit design. Chen received an MS in electrical engineering from Stanford University. Contact him at jameschen@cva.stanford.edu.*

R. Curtis Harting *is a research assistant at Stanford University. His research interests include the design and implementation of low-power architectures. Harting received a BSE in electrical and computer engineering from Duke University. Contact him at charting@stanford.edu.*

Vishal Parikh *is a research assistant at Stanford University. His research interests include computer architecture, on-chip networks, and programming systems. Parikh received a BS in electrical engineering from the University of Texas at Austin. Contact him at vparikh@stanford.edu.*

Jongsoo Park *is a PhD candidate at Stanford University. His research interests include compilers and energy-efficient computer architectures. Park received an MS in electrical engineering from Stanford University. Contact him at jongsoo@stanford.edu.*

David Sheffield *is a research assistant at Stanford University. His research interests include computer architecture, operating systems, and computer-aided design. Sheffield received an ScB in engineering from Brown University. Contact him at dsheffie@stanford.edu.*