International Journal of Ethics in Engineering & Management Education
Website: www.ijeee.in (ISSN: 2348-4748, Volume 9, Issue 8, August 2022)

Efficient Embedded Computing

M. Hari Krishna
Professor, Dept of ECE
Sree Vahini Institute of Science & Technology
Tiruvuru, Andharapradesh

T. Naga Raju
Assistant Professor, Dept of ECE
Siddhartha Institute of Engineering and Technology
Ibrahimpatnam, Hyderabad

Abstract: Embedded computer systems are ubiquitous, and contemporary embedded applications exhibit demanding computation and efficiency requirements. Most of the effort and expense arises from the non-recurring engineering activities required to manually lower high-level descriptions of systems to equivalent low-level descriptions that are better suited to hardware realization. They also offer the flexibility required to upgrade previously deployed systems as new standards and applications are developed. However, programmable systems are less efficient than fixed-function hardware. This significantly limits the class of applications for which programamble processors are an acceptable alternatives to application-specific fixed- function hardware, as efficiency demands often preclude the use of programmable hardware. This dissertation describes Elm, an efficient programmable system for high-performance embedded applications. Elm is significantly more efficient than conventional embedded processors on compute-intensive kernels. Elm allows software to exploit parallelism to achieve performance while managing locality to achieve efficiency. This dissertation proposes and critically analyzes concepts that encompass the interaction of computer architecture, compiler technology, and VLSI circuits to increase performance and efficiency in modern embedded computer systems. A central theme of this dissertation is that the efficiency of programmable embedded systems can be improved significantly by exposing deep and distributed storage hierarchies to software.

Key words: **Elm, Embedded, Computing, Energy, Efficiency, Architecture**

1. INTRODUCTION

Embedded computer systems are everywhere. Most of my contemporaries at Stanford carry mobile phone handsets that are more powerful than the computer I compiled my first program on. Embedded applications are adopting more sophisticated algorithms, evolving into more complex systems, and covering broader application areas as increasing performance and efficiency allow new and innovative technologies to be implemented in embedded systems.

Embedded applications will eventually exceed general-purpose computing in prevalence and importance. I argue that efficient programmable embedded systems are a critical enabling technology for future embedded applications. Improving the energy efficiency of computer systems is presently one of the most important problems in computer architecture. Perhaps the only other problem of similar importance is easing the construction of efficient parallel programs. As this dissertation demonstrates, data and instruction communication dominates energy consumption in modern computer systems. Consequently, improving efficiency necessarily entails reducing the energy consumed delivering instructions and data to processors and function units.

A central theme of this dissertation is that the efficiency of programmable embedded systems can be improved significantly by exposing deep and distributed storage hierarchies to software. To improve efficiency, Elm allows software to explicitly schedule and orchestrate the movement of instructions and data, and affords software significant control over the placement of instructions and data. Although this dissertation mostly contemplates efficiency, the relative complexity of mapping applications to Elm informed many of the architectural decisions. The Elm architecture attempts to simplify the complex and arduous task of compiling and mapping software that must satisfy realtime performance constraints to programmable architectures. Elm is designed to allow applications to be compiled from high-level languages using optimizing compilers and efficient runtime systems, eliminating the need for extensive development in low-level languages. The Elm architectures is specifically designed to allow software to embed strong assertions about dynamic system behavior and to construct deterministic execution and communication schedules.

1.1 *Embedded Computing*

It has become rather difficult to differentiate clearly between embedded computer systems and general- purpose computer systems. Embedded systems continue to increase in complexity, scope, and sophistication. Advances in semiconductor and information technology made massive computing capability both inexpensive and pervasive. The following paragraphs describe ways in which embedded applications and embedded computer systems differ significantly from general-purpose computer systems.

1.2. Power and Energy Efficiency — Embedded systems have demanding power and energy efficiency constraints. Active cooling systems and packages with superior thermal characteristics increase system costs. Many embedded systems operate in harsh environments and cannot be actively cooled. The limited space within enclosures may preclude the use of heat sinks. In mobile devices such as cellular telephone handsets, energy efficiency is a paramount design constraint because devices rely on batteries for energy and must be passively cooled.

1.3. Performance and Predictability — Contemporary embedded applications have demanding computation requirements. For example, signal processing in a 3G mobile phone handsets requires upwards of 30GOPS for a 14.4 Mbps

International Journal of Ethics in Engineering & Management Education
Website: www.ijeee.in (ISSN: 2348-4748, Volume 9, Issue 8, August 2022)

channel, while signal processing requirements for a 100 Mbps OFDM [110] channel can exceed 200GOPS [131]. Future applications will present even more demanding requirements as more sophisticated communications standards, compression techniques, codecs, and algorithms are developed. In addition, many applications impose real-time performance requirements on embedded computer systems. For example, a digital video decoder must deliver frames to the display at a rate that is consistent with the video stream, as there is little buffering in such systems to decouple decoding from presentation. Control systems, such as those found in automotive and telecommunications applications, impose even more demanding real-time constraints.

Designing systems to satisfy real-time performance constraints is challenging. Designers must reason about the dynamic behavior of complex systems with many interacting hardware and software components. The design process is simplified significantly when hardware and software is constructed to allow reasonably strong assertions to be made about the performance of components perform and how they interact. Predicting the performance of modern processors is notoriously difficult because structures such as caches and branch prediction units that improve performance introduce significant amounts of variability into the execution of software. Embedded processors often provide mechanisms that allow software to eliminate some of the variability, such as caches that allow specific cache blocks and ways to be locked. However, such mechanisms are typically exposed to software in such a way that programmers must program at a very low-level of abstraction. Application-specific fixed-function hardware can often be designed to deliver deterministic behavior that is readily analyzed and understood, though this comes at the expense of designers explicitly specifying the behavior of the hardware in precise detail.

For most embedded applications with real-time constraints, delivering performance and capabilities beyond those required to satisfy the real-time performance requirements offers little benefit. Rather than optimizing for computational performance, the design objective is often to minimize costs and reduce energy consumption.

1.4. Cost and Scalability — Embedded applications present a broad range of performance needs and cost allowances. Components within a cellular network such as base stations may be expensive because the systems are expected to be deployed for many years. Network operators prefer for these systems to be extensible and upgradeable so that deployed equipment can be updated as new technologies and standards are developed, and operators are accordingly willing to pay a premium for upgradeable systems that preserve their capital investments. Components in edge and client devices such as cellular handsets must be inexpensive as they are typically discarded after a few years.

Cost and energy efficiency considerations favor integrating more components of a system on a single die. Historically, costs associated with fabricating, testing, and packaging chips have been dominant in most embedded systems. The modest non-recurring engineering and tooling costs associated with the design, implementation, verification, and production of mask sets were amortized over the large production volumes needed to meet the demand for the consumer products that account for most embedded systems. Low volume systems, such as those used in defense and medical applications, exhibited different cost structures, and often had considerably greater selling prices.

However, it has become increasingly more difficult and expensive to design, implement, and verify chips in advanced semiconductor processes. Because programmable systems are presently not efficient enough for demanding embedded applications to be implemented in software and intensive hardware design and validation efforts. The implementation complexity and effort associated with developing application-specific hardware impose significant engineering costs, and the time and effort required to implement and verify application-specific fixed-function logic increases with the scale and complexity of a system. Furthermore, the inflexibility of deployed systems that rely on application-specific fixed-function hardware increases the cost of implementing new standards and slows their adoption because existing infrastructure must be replaced.

The significant cost of designing new systems often discourages the development, adoption, and deployment of innovative applications, technologies, algorithms, protocols, and standards. For example, the expenses associated with developing novel medical equipment can limit the deployment and accessibility of important medical devices, as the potential markets for new devices may not be large enough to adequately amortize development costs in addition to those costs associated with regulatory approval and clinical trials. Because designs require very large volumes to justify such engineering costs, designs will be limited to individual products with large volumes and families of related products with large aggregate volumes.

This favors configurable and programmable systems, such as programmable processors, field programmable gate arrays, and systems that integrate fixed-function hardware and programmable processors on a single die. Unfortunately, partitioning applications between software and hardware requires programming at low-levels of abstraction to expose the hardware capabilities to software, which increases application software development costs and development times: the design and verification of a complex system-on-chip can require hundreds of engineer-years [102] and incur non-recurring engineering costs in excess of $20M-$40M.

2. CONTEMPORARY EMBEDDED ARCHITECTURES

Contemporary embedded computer systems comprise diverse collections of electronic components. Advances in semiconductor scaling allow complex systems to be integrated one a single system-on-chip. Most system-on-chip components comprise heterogeneous collections of microprocessors, digital signal processors, application-specific fixed-function hardware accelerators, memory controllers, and input-output interfaces that allow the chip to communicate

International Journal of Ethics in Engineering & Management Education
Website: www.ijeee.in (ISSN: 2348-4748, Volume 9, Issue 8, August 2022)

with other devices. In most contemporary embedded systems, the majority of the computation capability is provided by a few system-on-chip components.

It is reasonable to wonder whether we could replace the fixed-function hardware with collections of simple processors similar to the RISC processors that are prevalent in contemporary embedded systems. The processors would perhaps be organized as a multicore or manycore processor system-on-chip. It is possible for such an architecture to deliver sufficient arithmetic bandwidth to satisfy the demands of most applications, as a tiled collection of small processors can provide a large aggregate number of function units. However, I argue in this chapter that the energy efficiency of simple RISC processors is insufficient for such a system to deliver acceptable power and energy efficiency.

This chapter provides a brief examination of existing embedded computer systems. We begin with a case study of a simple and efficient embedded RISC processor that is representative of contemporary commercial processors; we consider its efficiency, and examine where and how energy is expended. We then discuss strategies and technologies that have been proposed for improving energy efficiency. We conclude with a survey of a few relevant architectures that have used these technologies to improve efficiency.

2.1 *The Efficiency Impediments of Programmable Systems*

Analyzing how area and energy is consumed in simple RISC processors provides significant insight into the efficiency limitations of conventional programmable processors. Table 2.1 lists significant attributes of a very simple embedded processor. The processor is derived from a synthesizable implementation of 32-bit SPARC processor designed for embedded applications [51]. We modified the design to remove everything except the integer pipeline, instruction and data caches, and those parts of the memory controller that are needed

3. THE ELM ARCHITECTURE

This chapter introduces the Elm architecture. It motivates and develops important concepts that informed the design of Elm, and presents many of the ideas that are developed . Embedded applications have abundant task-level and data-level parallelism, and there is often considerable operation-level parallelism within tasks. Components of embedded systems such as application-specific integrated circuits exploit this abundant parallelism by distributing computation over many independent function units; both performance and efficiency improve because instructions and data are distributed close to function units. Efficiency is further improved by exploiting extensive reuse and producer-consumer locality in embedded applications. Individual components in embedded systems tend to perform the same computation repeatedly, which is why they can benefit from fixed-function logic. This behavior may be used to expose extensive instruction and data reuse.

Programmability allows tasks to be time-multiplexed onto collections of function units, which both improves area efficiency and energy efficiency. Systems that use special-purpose fixed-function logic usually provision dedicated hardware for each application and tasks. Depending on application demand, some fixed- function hardware will be idle, resulting in resource underutilization and poor area efficiency. For example, cellular telephone handsets that support multiple wireless protocols typically implement multiple baseband processing blocks even though multiple blocks cannot be used concurrently. Time multiplexing tasks onto programmable hardware allows a collection of function units to implement different applications and different tasks within applications as demanded by the system. Programmability improves energy efficiency when data communication dominates by allowing tasks to be dispatched to data. Rather than requiring that data be transferred to and from hardware capable of implementing some specific task, the task may be transferred to the data, reducing data communication and aggregate instruction and data bandwidth demand.

The Elm architecture exploits the abundant parallelism, reuse, and locality in embedded applications to achieve performance

.

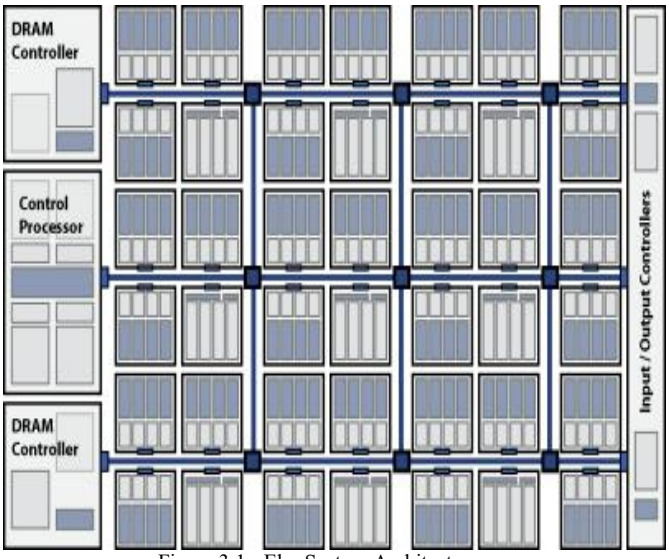

Figure 3.1 - Elm System Architecture

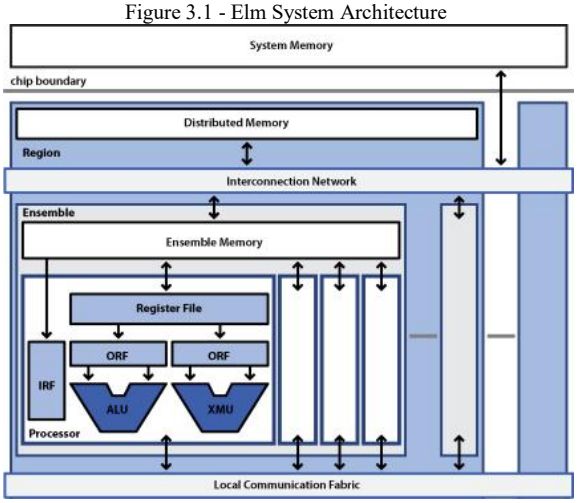

Figure 3.2 - Elm Communication and Storage Hierarchy.

International Journal of Ethics in Engineering & Management Education
Website: www.ijeee.in (ISSN: 2348-4748, Volume 9, Issue 8, August 2022)

Instructions are issued from instruction register files (IRFs) distributed among the function units. Distributed operand register files (ORFs) provide inexpensive locations for capturing short-term reuse and locality close to function units. Collections of processors are grouped to form Ensembles, which share a local Ensemble Memory. Collections of Ensembles are grouped with a distributed memory tile to form a region. The local communication fabric provides low-latency, high- bandwidth communication links and allows processors to communicate through registers. The system comprises a collection of processors, distributed memories, memory controllers, and input-output controllers. The modules are connected by an on-chip interconnection network and efficiency. Figure 3.1 provides a conceptual illustration of an Elm system-on-chip. The latency-optimized control processor.

3.2 System Architecture

Handles system management tasks, while the computationally demanding parts of embedded applications are mapped to the dense fabric of efficient Elm processors. Efficiency is achieved by keeping a significant fraction of instruction and data bandwidth close to the function units, where hierarchies of distributed register files and local memories deliver operands and instructions to the function units efficiently. The processor complexity is low enough that custom circuit design techniques could be used throughout to improve performance and efficiency, and small enough for large amounts of arithmetic bandwidth to be implemented economically. We estimate that an Elm system implemented in a 45 nm CMOS process would deliver in excess of 500 GOPS at less than 5 W in about 10 mm x 10 mm of die area. Figure 3.2 illustrates the distributed and hierarchical register and memory organization. The Elm processors are statically scheduled dual-issue 32-bit processors. Instructions are issued in pairs, with one instruction sent to the ALU pipeline and one to the XMU pipeline. The ALU executes complex arithmetic, logic, and shift instruction; the XMU executes simple arithmetic, memory, control, communication, and synchronization instructions. Instructions are issued from registers to reduce the cost of delivering instructions. The distributed and hierarchical register organization reduces the cost of accessing local instructions and data.

The Elm processors use block instruction and data transfers to improve the efficiency of transferring instructions and data between registers and memory.

The memory and communication systems near the processors provide predictable latencies and bandwidth to simplify the process of compiling to real-time performance and efficiency constraints. The compute, memory, and communication resources are exposed to software to provide fine-grain control over how computation unfolds, and the programming systems and compiler may explicitly orchestrate instruction and data movement. The result is that the complexity and expense of developing large embedded systems is in the development of compilers and programming tools, the cost of which is amortized over many systems, rather than in the design and verification of special-purpose fixed-function hardware which must be paid for in each new system.

Ensembles

Clusters of four processors are grouped into Ensembles, the primary physical design unit for compute resources in an Elm system. The processors in an Ensemble are loosely coupled, and share local resources such as a pool software-managed memory and an interface to the interconnection network. The Ensemble organization is exposed to software, and the software concept of an Ensemble provides a metaphor for describing and encoding spatial locality among the processors within an Ensemble.

The compiler can exploit task-level parallelism by executing parallel tasks on the processors in an Ensemble, capturing fine-grain producer-consumer data locality within the Ensemble to improve efficiency. Reuse among the processors reduces the demand for instruction and data bandwidth outside the Ensemble. For example, executing parallel iterations of a loop on different processors in an Ensemble increases performance by exploiting data-level parallelism; amortizing the cost of moving the instruction and data to Ensemble memory increases efficiency by exploiting instruction and data reuse across iterations.

Remote memory accesses are performed over the on-chip interconnection network. The Ensemble memory can be used to stage transfers. Hardware at the network interface supports bulk memory operations to assist the compiler in scheduling proactive instruction and data movements to hide remote memory access latencies. The Ensemble organization effectively concentrates interconnection network traffic at Ensemble boundaries and allows multiple processors share a network interface and the associated connection to the local router. This amortizes the network interface and network connection across multiple processors, which provides an important improvement in area efficiency because the processors are relatively small.

The local communication fabric connects the processors in an Ensemble with high-bandwidth point- to-point links. The fabric can be used to capture predictable communication, and allows the compiler to map instruction sequences across multiple processors. The fabric can also be used to stream data between Ensembles deterministically at low latencies. The fabric is statically routed, and avoids the expense cycling large memory arrays incurs when communicating through memory.

4. INSTRUCTION REGISTERS

This chapter introduces instruction registers, distributed collections of software-managed registers that extend the memory hierarchy used to deliver instructions to function units. Instruction registers exploit instruction reuse and locality to improve efficiency. They allow critical instruction working sets such as loops and kernels to be stored close to function units. Instruction registers are implemented efficiently as distributed collections of small instruction register files. Distributing the register files allows instructions to be stored close to function units, reducing the energy

International Journal of Ethics in Engineering & Management Education
Website: www.ijeee.in (ISSN: 2348-4748, Volume 9, Issue 8, August 2022)

consumed transferring instructions to data-path control points. The aggregate instruction storage capacity provided by the distributed collection of register files allows them to captured important working sets, while the small individual register files are fast and inexpensive to access.

The remainder of this chapter is organized as follows. I explain general concepts and describe the operation of instruction registers. I then describe microarchitectural details of the instruction register organization implemented in Elm. This is followed by examples illustrating the use of instruction registers. I then describe compiler algorithms for allocating and scheduling instruction registers. I conclude this chapter with a comparison of the performance and efficiency of instruction registers to similar mechanisms for reducing the amount of energy expended delivering instructions.

Embedded applications are dominated by kernels and loops that have small instruction working sets and extensive instruction reuse and locality. Consequently, a significant fraction of the instruction bandwidth demand in embedded applications can be delivered from relatively small memories.

The instruction caches and software-managed instruction memories found in most embedded processors have enough capacity to accommodate working sets that are larger than most embedded kernels. Often, first-level instruction caches are made as large as possible while still allowing the cache to be accessed in a single cycle at the desired clock frequency. Consequently, the instruction caches found in common embedded processors are much larger than a single kernel or loop body: it is not uncommon to find embedded processors with instruction caches ranging from 4 KB to 32 KB, while many of the important kernels and loops in an instruction stored in an instruction register does not require a tag check, and consequently instruction register files are faster and less inexpensive to access than caches of equivalent capacity. Instruction register files are much smaller than typical instruction caches. The instruction register files implemented in Elm provide 512 B of aggregate storage, enough to accommodate 64 instruction pairs, per processor. This allows the register files to be implemented close to function units, which reduces the cost of transferring instructions from instruction registers to data-path control points.

5. OPERAND REGISTERS AND EXPLICIT OPERAND FORWARDING

Operand registers and explicit forwarding expose to software the spatial distribution of registers among function units, which increases software control over the movement of data between function units and allows software to place data close to where they will be consumed, reducing the energy expended transporting operands to function units. Operand registers and explicit operand forwarding allow the compiler to keep a significant fraction of the data bandwidth within the kernels of embedded applications close to the function units, which reduces the amount of energy expended staging data in registers.

The remainder of this chapter is organized as follows. I begin by introducing the basic concepts motivating explicit operand forwarding and operand registers. I then describe how explicit operand forwarding and operand registers extend a conventional register organization and affect instruction set architecture, using Elm as a specific example of a machine that uses both explicit operand forwarding and operand registers. I then discuss microarchitecture, again using Elm as an example of an architecture that implements explicit operand forwarding and operand registers. After describing the architecture, I discuss compilation issues such as instruction scheduling and register allocation. An evaluation follows. I conclude with related work and then summarize the chapter.

Operand registers extend a conventional register organization with distributed collections of small, inexpensive general-purpose operand register files, each of which is integrated with a single function unit in the execute stage of the pipeline. Essentially, operand registers extend the pipeline registers that deliver operands to function units in a conventional pipeline into shallow register files.

6. INDEXED AND ADDRESS-STREAM REGISTERS

This chapter introduces the concepts of indexed registers and address-stream registers, and describe an architecture that uses indexed registers and address-stream registers to expose hardware for performing vector memory operations to software.

Registers are used to store intermediate values and to stage data transfers between function units and memory. In an architecture such as Elm that provides operand registers, general-purpose registers are used to capture working sets that exhibit medium-term reuse and locality. Providing additional general-purpose registers allows software to capture larger working sets in registers, reducing the amount of data that must be transferred between registers and memory. This improves energy efficiency and performance, as registers are less expensive and faster to access than memory. However, exposing additional architectural registers to software requires the use of instruction encodings that dedicate more bits for encoding register names. This leaves fewer bits available for encoding operations or increases the number of bits required to encode an instruction, neither of which are desirable.

Perhaps more significantly, many compute-intensive kernels, and particularly those with enough structured reuse to benefit from large register files, access data in ways that makes it difficult to exploit large register files without using software techniques, such as loop unrolling, to expose additional scalar variables that can be assigned to registers. A kernel that operates on the elements of an array serves to illustrate this point. Before assigning the elements of the array to registers, the compiler must typically replace each reference to an element of the array with an equivalent reference to a scalar variable that is introduced explicitly for this purpose, a transformation commonly referred to as scalar replacement. If a kernel accesses the array in a loop using an index that is a non-trivial affine function of the loop induction variable, the loop must be unrolled to enable the scalar replacement

International Journal of Ethics in Engineering & Management Education
Website: www.ijeee.in (ISSN: 2348-4748, Volume 9, Issue 8, August 2022)

transformation. Because loop unrolling replicates operations within the bodies of loops, loop unrolling increases the number of instructions in the critical instruction working sets and kernels of an application. Reductions in the energy consumed staging operands in registers may be offset by increases in the energy consumed loading instructions from memory. Consequently, software techniques such as loop unrolling that exploit additional registers but introduce additional instructions are unattractive for architectures such as Elm because the additional instructions increase instruction register pressure.

Before continuing, we should observe that the aggregate number of general-purpose registers that may be accessed by the instructions residing in the instruction registers will be limited to a small multiple of the number of instruction registers. This follows from each instruction being able to reference at most 3 general-purpose registers, though many instructions will reference fewer. There are techniques for allowing software to access a larger number of physical registers. For example, vector processors are able to provide a large number of physical registers and offer compact instruction encodings by associating a large number of physical registers with a single vector register identifier. Similarly, architectures that provide register windows allow software to access different sets of physical registers by rotating through distinct windows, though these are typically used to avoid saving and restoring registers at function call boundaries, and when entering and terminating interrupt and trap handlers [114].

Index registers let software access some subset of the architectural registers indirectly. The registers that can be accessed indirectly through the index registers are referred to as indexed registers. Some of the indexed registers may be architectural registers that can be accessed using a conventional register name; others may be dedicated indexed registers that can only be accessed through index registers. Index and indexed registers improve efficiency by allowing larger data working sets to be efficiently stored in the register hierarchy; they let software increase the fraction of intra-kernel data locality that can be compactly captured in registers without resorting to software techniques such as loop unrolling that increase the instruction working set of a kernel. Thus, index and indexed registers simultaneously address the problem of allowing a processor to increase the number of registers that are available to software without requiring instruction set changes and the problem of allowing software to exploit addition registers without resorting to techniques such as loop unrolling that increases the number of instructions in the working sets of the important kernels within embedded applications. Like vector registers, index and indexed establish an association between a large number of physical registers and a register identifier that is exposed to software. However, index and indexed are more general and more flexible: index and indexed registers expose the association to software and let software control the assignment of physical registers to register identifiers, and allow the physical registers to be accessed using multiple register identifiers.

Vector memory operations transfer multiple data words between memory and the indexed registers, which improves code density by allowing a single vector memory instruction to encode multiple memory operations. Techniques that improve code density improve efficiency in general, and are particularly attractive for architectures such as Elm that achieve efficiency in part by delivering a significant fraction of instruction bandwidth from a small set of inexpensive instruction registers. Vector memory operations provide a means for software to explicitly encode spatial locality within the operation, which allows hardware to improve the scheduling and handling of the memory operations. Decoupling allows the memory operations that transfer the elements of a vector to complete while the processor continues to execute instructions, which lets software schedule computation in parallel with memory operations. Decoupling vector memory operations also assists software in tolerating memory access latencies, as vector memory operations can be initiated early to allow the operations more time to complete.

The Elm architecture exposes the hardware that generates sequences of addresses for vector memory operations as address-stream registers. The address-stream registers deliver sequences of addresses and offsets that can be combined with other registers to compute the effective addresses of memory operations. Software controls the address sequences computed by the address-stream registers by loading configuration words from memory. Software can save the state of an address sequence by storing the address-stream register to memory, and can later restore the sequence by loading the register from memory. This lets software map multiple address sequences onto the address-stream registers. Software can move elements from the address sequences to other registers, which allows software to implement complex address calculations using a combination of address-stream registers and general-purpose and address registers.

Exposing the address generators as address-stream registers allows software to use the hardware to accelerate repetitive address calculations, improving both performance and efficiency by eliminating instructions that perform explicit address computations from the instruction stream. It also simplifies the task of decomposing a large transfer that can be compactly expressed using vector memory operations and address-stream registers into a sequence of smaller operations that better exploit the storage available in the register organi-zation: the address-stream registers maintain the state of the operation between transfers.

7. ENSEMBLES OF PROCESSORS

This chapter introduces the Ensembles of processors organization used in an Elm system. An Ensemble is a coupled collection of processors, memories, and communication resources. An Elm system is a collection of Ensembles. The Ensembles share a distributed memory system, and external interfaces for communicating with other chips. The Ensemble is the fundamental compute design unit in an Elm system, and is extensively replicated throughout an Elm system. This extensive replication and reuse of a small design unit allows

International Journal of Ethics in Engineering & Management Education
Website: www.ijeee.in (ISSN: 2348-4748, Volume 9, Issue 8, August 2022)

aggressive custom circuits and design efforts to be used effectively within an Ensemble.

Figure 7.1 illustrates the organization of the processors and memories comprising an Ensemble. The processors in an Ensemble are assigned a unique processor identifier in the range [0,3]. We usually refer to the processors using the names EP0, EP1, EP2, and EP3 when we need to identify a particular processor.

Ensemble Memory

The Ensemble memory is a local software-managed instruction and data store that is shared by the processors in the Ensemble. The Ensemble memory serves several important functions. Perhaps most importantly, it provides local memory for storing important instruction and data working sets that exceed the capacity of the register files close to the processors. The Elm compiler uses the local Ensemble to store instructions that may be loaded to the instruction registers during the execution of a kernel, which reduces the performance and energy impact of loading instructions from memory during the execution of a kernel. The Elm compiler also uses the local Ensemble memory to temporarily spill data when it is unable to allocate registers to all of the live variables in a kernel.

The Ensemble memory also provides local storage for staging data and instruction transfers between the processors in the Ensemble and memory beyond the Ensemble. The storage capacity provided by the local Ensemble memory allows software to use techniques such as prefetching and double buffering that exploit communication and computation concurrency to effectively hide remote memory access latencies.

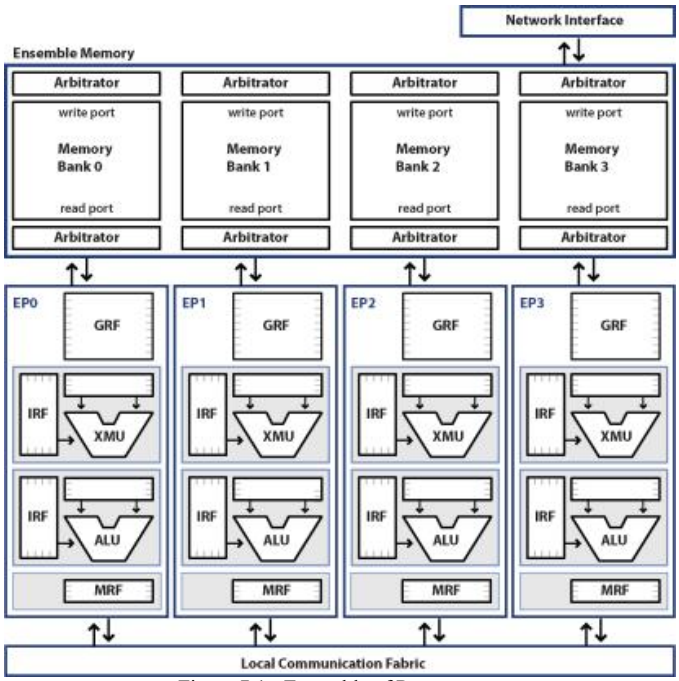

Figure 7.1 - Ensemble of Processors.

The Ensemble organization allows the expense of transferring instructions and data to an Ensemble to be amortized when multiple processors within the Ensemble access the instructions and data. The Ensemble memory is a local software-managed instruction and data store that is shared by the processors in the Ensemble. The local communication fabric provides a low-latency, high- bandwidth interconnect for efficiently transferring data between threads executing concurrently on different processors. These links may be used to stream data between threads that are mapped to different processors within an Ensemble, avoiding the expense of streaming data through memory.

The Ensemble organization allows the expense of transferring instructions and data to an Ensemble to be amortized when multiple processors within the Ensemble access the instructions and data. This provides animportant mechanism for significantly improving the efficiency of instruction and data delivery when data parallel codes are mapped to the processors in an Ensemble.

The Ensemble memory is banked to allow it to service multiple transactions concurrently. Each processor within an Ensemble is assigned a preferred bank. Instructions and data that are private to a single processor may be stored in its preferred bank to provide deterministic access times. The arbiters that control access to the read and write ports are biased to establish an affinity between processors and memory banks.

8. THE ELM MEMORY SYSTEM

This chapter introduces the Elm memory system. Elm exposes a hierarchical and distributed on-chip memory organization to software, and allows software to manage data placement and orchestrate communication. Additional details on the instruction set architecture appear in Appendix C. The chapter is organized as follows. I first explain general concepts and present the insights that informed the design of the memory system. I then describe aspects of the microarchitecture in detail. This is followed by an example and an evaluation of several of the mechanisms implemented in the memory system. Finally, I consider related work and conclude.

Elm is designed to support many concurrent threads executing across an extensible fabric of processors. To improve performance and efficiency, Elm implements a hierarchical and distributed on-chip memory organization in which the local Ensemble memories are backed by a collection of distributed memory tiles. The architectural organization of the Elm memory system is illustrated. Elm exposes the memory hierarchy to software, and lets software control the placement and orchestrate the communication of data explicitly. This allows software to exploit the abundant instruction and data reuse and locality present in embedded applications. Elm lets software transfer data directly between Ensembles using Ensemble memories at the sender and receiver to stage data. This allows software to be mapped to the system so that the majority of memory references are satisfied by the local Ensemble memories, which keeps a significant fraction of the memory bandwidth local to the Ensembles. Elm provides various mechanisms that assist software in managing reuse and locality, and assist in scheduling and orchestrating

communication. This also allows software to improve performance and efficiency by scheduling explicit data and instruction movement in parallel with computation.

9. CONCLUSION

This dissertation presented the Elm architecture and contemplated the concepts and insights that informed its design. The central motivating insight is that the efficiency at which instructions and data are delivered to function units ultimately dictates the efficiency of modern computer systems. The major contributions of this dissertation are a collection of mechanisms that improve the efficiency of programmable processors. These mechanisms are manifest in the Elm architecture, which provides a system for exploring and understanding how the different mechanisms interact.

REFERENCES

[1] D. Abts, S. Scott, and D.J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *International Parallel and Distributed Processing Symposium (IPDPS),* 2003.

[2] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing,* pages 159-170, New York, NY, USA, 1993. ACM.

[3] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT alewife machine: A large-scale distributed memory multiprocessor. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1991.

[4] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The mit alewife machine: architecture and performance. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture,* pages 2-13, New York, NY, USA, 1995. ACM.

[5] Anant Agarwal and Markus Levy. The kill rule for multicore. In *DAC '07: Proceedings of the 44th annual Design Automation Conference,* pages 750-753, New York, NY, USA, 2007. ACM.

[6] T. Agerwala and J. Cocke. High Performance Reduced Instruction Set Processors. *IBM Thomas J. Watson Research Center Technical Report,* 558845, 1987.

[7] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transfom. *IEEE Transactions on Computing,* 23(1):90-93, 1974.

[8] A. Allen, J. Desai, F. Verdico, F. Anderson, D. Mulvihill, and D. Krueger. Dynamic frequencyswitching clock system on a quad-core itanium processor. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International,* pages 62 - 63,63a, Feb. 2009.

[9] Arvind and Vinod Kathail. A multiple processor data flow machine that supports generalized procedures. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture,* pages 291-302, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[10] Krste Asanovic. *Vector Microprocessors.* PhD dissertation, University of California at Berkeley, Berkeley, CA, USA, 1998.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[12] James Balfour, William J. Dally, David Black-Schaffer, Vishal Parikh, and Jongsoo Park. An energy- efficient processor architecture for embedded systems. *Computer Architecture Letters,* pages 29-32, 2008.

[13] James Balfour, R. Curtis Harting, and William J. Dally. Operand registers and explicit operand forwarding. *Computer Architecture Letters,* 2009.

[14] Max Baron. Silicon a la carte. *Microprocessor Report,* January 2004.

[15] Luiz Andrd Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture,* pages 282-293, New York, NY, USA, 2000. ACM.