Smart Memories: A Configurable Processor Architecture for High Productivity Parallel Programming

A. Solomatnikov, A. Firoozshahian, F. Labonte, M. Horowitz, C. Kozyrakis, K. Olukotun, K. Mai
Computer Systems Laboratory
Stanford University
Stanford, CA, 94305
{sols,aminf13,flabonte, horowitz}@stanford.edu, christos@ee.stanford.edu, kunle@ogun.stanford.edu, demon@stanford.edu

Abstract: *With single processor systems running into instruction-level parallelism (ILP) limits and fundamental VLSI constraints, multiprocessor chips provide a realistic path towards scalable performance by allowing one to take advantage of thread-level (TLP) and data-level parallelism (DLP) in emerging applications. Nevertheless, parallel architectures are limited by the difficulty of parallel application development. This challenge has led to the invention of new programming models to simplify the way in which parallel programs are developed correctly and efficiently. Smart Memories is a scalable, hierarchical architecture which, using a modular design, addresses the process technology issues, such as power consumption and wire latency. Its reconfigurability allows executing applications described in different programming models with high performance. Simulations have shown that considerable speed ups (2x to 10x) can be achieved over a broad range of applications, while a small amount of power and area penalty is tolerated for reconfiguration.*

Keywords: Smart Memories, Reconfiguration, Parallel programming, streaming, memory system, transactional programming

Introduction

Over the past two decades, microprocessors have doubled in performance every 18 months without the need for substantial changes to the underlying instruction-set architecture. Historically, faster transistors, deeper pipelining, and increased instruction level parallelism (ILP) have been the major factors in increasing processor performance. However, for performance to continue to scale over the coming decades, significant architectural changes are required, since two of these factors--increased ILP and deeper pipelining--have been exhausted [4]. Architectural change is also being driven by process technology that is becoming wire and power-limited, rather than device-count limited [2]. Future processors should exploit explicit parallelism, be modular, minimize the use of global wires, and exploit locality for power efficiency. Fortunately, most emerging applications, such as cognitive reasoning, computational biology, and large-scale enterprise services, have large amounts of explicit task- and/or data-level parallelism that can be mapped on such parallel architectures.

One of the key challenges of parallel systems is how to program them. Existing parallel programming approaches are counterproductive for all but a few expert users, since they require the programmer to manage concurrency directly by creating and synchronizing parallel threads. This difficulty stems from the need to achieve the often conflicting goals of functional correctness and high performance. For example, with shared-memory programming, a small number of coarse-grain locks make it simpler to correctly order accesses to shared objects, while many fine-grain locks allow higher performance by reducing the amount of time wasted in accessing locks.

Moreover, conventional architectural techniques for increasing performance--caching and pre-fetching--are not very successful for most of data-intensive applications. The little global data reuse and producer-consumer patterns in memory accesses which are not captured by caches are considered the main reasons for this poor performance.

Stanford researchers have proposed two new models to address parallel programming productivity. The first model is “stream programming,” [3] which constructs programs as collections of computational kernels, linked in data-flow patterns. This type of programming is appropriate for data-intensive, regular applications, such as signal processing, telecommunications, and scientific computing. Stream programming exposes the data parallelism and explicit communication as well as locality patterns. The second model is “transactional programming,” [6] which views programs as collections of database-like, atomic transactions. Transactions provide a single abstraction for the parallelism, synchronization, communication, and failure recovery. Therefore, transactional programming is appropriate for irregular applications with unstructured control and communication patterns, such as enterprise services and cognitive tasks. Considering the substantial dissimilarity between these programming models, as well as the different levels of parallelism they aim to exploit, designing an architecture that can reuse hardware resources

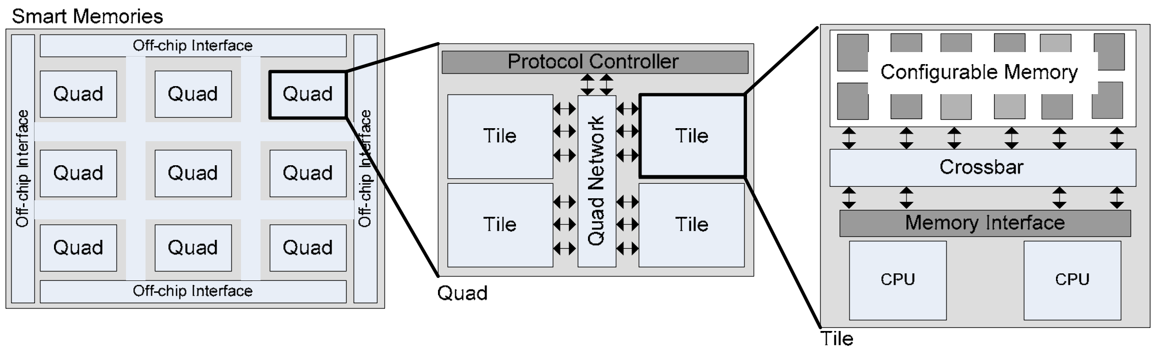


Figure 1. Smart Memories architecture hierarchy

to efficiently support them presents an interesting challenge.

Smart Memories Architecture

Figure 1 illustrates Smart Memories hierarchical architecture, which integrates a large number of processors and memory blocks on a single chip. The system consists of Tiles, each with two VLIW cores, several reconfigurable memory blocks, and a crossbar connecting them. Tiles are placed in groups of four, forming Quads. Tiles in the Quad are connected via a Quad network and share a common interface for communicating to the outside world. A shared protocol controller provides support for the Tiles by moving data in and out of the local memory blocks and implementing memory protocols (such as cache coherence) in different execution modes. Quads are then connected to each other and to the off-chip interfaces using a mesh-like network.

The key feature of Smart Memories that addresses technology issues lies in its hierarchical architecture. The modularity of the system addresses the design complexity concerns and makes the architecture an excellent target for scalability. The system is expanded simply by adding elements at the top level of the hierarchy (Quads), increasing both the processing and memory resources. Using fast local memories and high-bandwidth interconnects to connect elements at every level of the hierarchy tolerates the increasingly important wire-delay problem in modern architectures. Power efficiency is achieved using simple VLIW cores and selectively turning off the unnecessary components in the system.

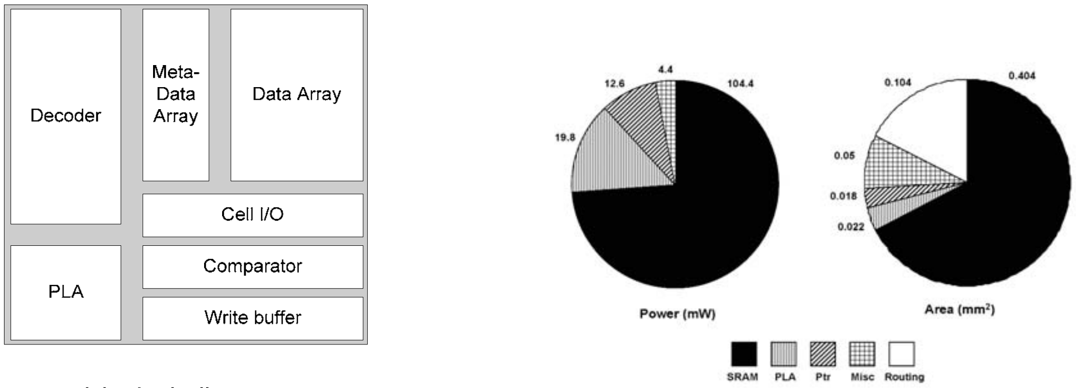
Smart Memories attempts to exploit different levels of parallelism in the application, which might be orthogonal to one another: VLIW cores tackle the conventional levels of instruction level parallelism (ILP) in most applications, while multiple execution engines target the more explicit levels of parallelism at the data and task levels. Data-level parallelism is extracted by mapping computational kernels on the CPU cores, which follow producer-consumer patterns to pass streams of data, while task-level parallelism can be exploited by either assigning standard threads to each processor or by running atomic transactions, with or without an explicit order being assigned to them.

Reconfigurable Memory System

With the exception for ILP, which is utilized within the processors, the other two more explicit levels of parallelism cannot be exploited efficiently without appropriate support from the memory system. The memory system is responsible for not only keeping the execution units busy by moving data in and out of the local memories, but also for tracking of the data sharing and dependencies and detecting dependence violations. In Smart Memories, the memory system can configure itself as a hierarchy of caches for capturing temporal and spatial locality in memory accesses, which is used when running threads or transactions in task-parallel applications. The system also provides a mechanism for tracking data dependencies and detecting violations. Alternatively, the memory system can be configured as a series of local memories or FIFOs, supported with DMA engines, which is suitable for running data-parallel applications with producer-consumer locality.

The memory system consists of three major reconfigurable blocks, highlighted in Figure 1. The memory interface coordinates accesses from processor cores to local memories and allows reconfiguration of basic memory accesses; a basic operation, such as a Store instruction, can treat a memory word differently in transactional mode than normal multi-threaded mode. This interface also can broadcast accesses to a set of local memory blocks; for example, when accessing multiple ways of a cache, the access is concurrently sent to all the blocks forming the cache ways.

Figure 2.a shows the block diagram of a local memory mat. Each memory mat has an array of data words, each associated with a few meta-data bits. Meta-data bits store the status of that data word in any particular configuration of the memory system and their state is considered in every memory access; access can be discarded based on the status of these bits. For example, when mats are configured as a cache, these bits are used to store the status of a cache line and access is discarded if the status indicates an invalid line. A programmable logic updates the state of these bits



(a) Block diagram (b) Power and area breakdown

Figure 2. Reconfigurable memory mat

atomically, simultaneous with access to the data word. A built-in comparator and a set of pointers allow the mat to be used as a tag storage (for cache) or as a FIFO. Mats are connected to each other through an inter-mat network that communicates control information when they are accessed as a group.

Local memory blocks require a different kind of support depending on the programming model. The protocol controller is a reconfigurable control engine which allows executing a sequence and/or a combination of basic operations to support memory mats. These operations include: loading and storing data words (or cache lines) into mats, manipulating meta-data bits, keeping track of outstanding requests from each Tile, and broadcasting data or control information to Tiles within the Quad. This controller is connected to a network port for sending and receiving requests to/from other Quads or off-chip interfaces.

Support for Multiple Programming Models

Smart Memories architecture supports three major programming models that use the underlying memory system differently: conventional multi-threading in a cache-coherent shared-memory environment, streaming and transactional programming

In the conventional shared-memory execution mode, memory mats are configured as instruction and data caches, with support for the fine-grain synchronization operations. The protocol controller acts as a cache coherence engine, which refills the caches and enforces coherence. Low latency in servicing cache misses is achieved by merging requests from different processors whenever possible and trying to satisfy the requests by conducting local cache-to-cache transfers between the Tiles, without crossing the Quad boundary. High bandwidth is achieved by keeping track of a relatively high number of outstanding requests and processing them in parallel.

Streaming [3] is the second execution model supported by the Smart Memories. In this mode, memory mats are used as Stream Register Files (SRF) to store streams of data locally or as FIFO buffers to pass the streams from one processor to the other. They also may be used as scratchpads or local instruction memory. The protocol controller acts as a high-throughput DMA engine with support for multiple DMA transfer modes, such as stride or indexed gather/scatter.

Transactional programming [6] is the third major programming model supported by the Smart Memories. In this mode, memory mats are configured as local caches for processors which can store uncommitted transaction state and trace the data dependencies. Meta-data bits in memory mats are used for marking words to detect dependency violations that might occur between the transactions. At commit time, the protocol controller broadcasts all the modifications of the committing transaction to other Tiles and detects any possible dependence violations.

Status, Results and Summary

Currently, we have developed a functional simulator for the system, which can simulate an arbitrary configuration of processors, contexts and memory hierarchy. We have also completed Verilog RTL for the Tile, which at this time is being verified. We are also designing the protocol controller and the rest of the system.

Figure 3 shows the speed ups for two probabilistic inference applications developed in multi-threaded mode, while Figure 4 shows the performance of applications developed using the transactional model. As demonstrated, Smart Memories provides significant speed ups (2x to 10x) over sequential architectures for a variety of applications [1, 6]. Furthermore, using a chip prototype, it has been shown that compared to normal custom memory arrays, configurable memory blocks can be efficiently implemented with only a small area and power penalty (Figure 2.b, [5]).

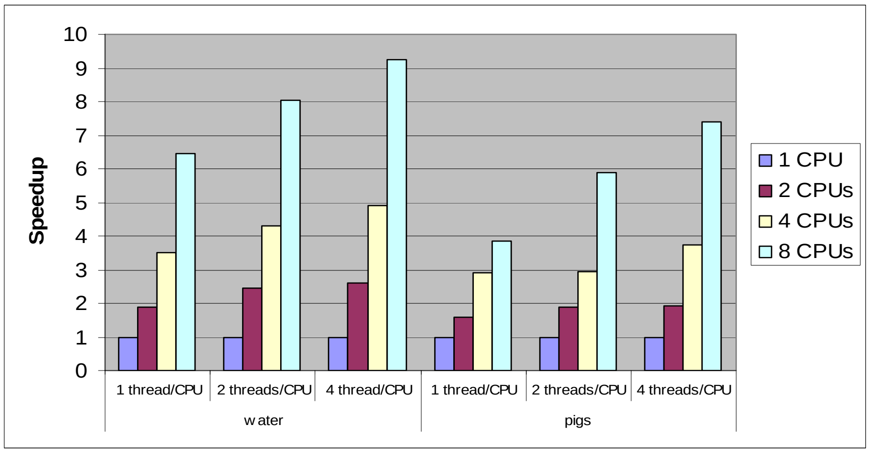


Figure 3. Speed ups for cognitive tasks

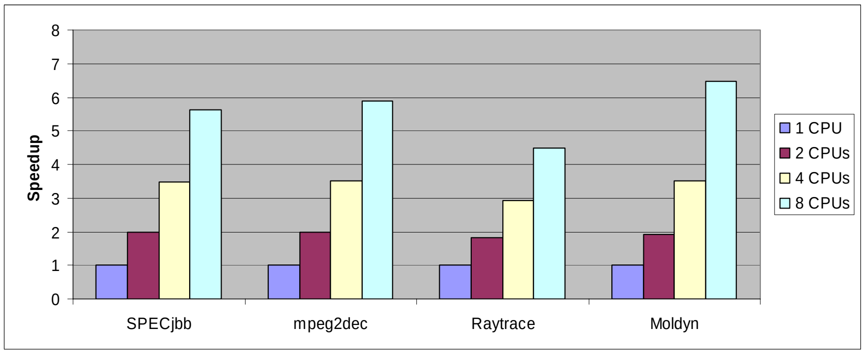


Figure 4. Transactional programming performance

In summary, Smart Memories is a scalable multiprocessor system designed to efficiently support a wide range of applications. Its modularity and hierarchical design address modern technology concerns, such as design complexity, power efficiency and wire delay. Its reconfigurable memory system is the key feature in supporting different programming models, ranging from the conventional shared-memory model to more innovative, recently developed models, such as streaming and transactions, each suitable for a large class of emerging applications. Overall, the results suggest that Smart Memories is a promising architecture for next-generation, general purpose computing systems that can achieve high productivity and high performance.

The Smart Memories project has been supported by DARPA IPTO through the PCA program grants F29601-01-2-0085 and F29601-03-2-0117.

References

1. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz, “Smart Memories: A Modular Reconfigurable Architecture,” *27th International. Symposium on Computer Architecture*, June 2000.
2. R. Ho, K. Mai, and M. Horowitz, “Efficient on-chip global interconnects,” *IEEE Symposium on VLSI Circuits*, June 2003.
3. U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. Ahn, P. Mattson, J. Owens, “Programmable Stream Processors,” *IEEE Computer*, vol.36, no. 8, pp 54-62, August 2003.
4. M. Horowitz, W. Dally, “How Scaling Will Change Processor Architecture,” *International Solid States Circuits Conference*, February 2004.
5. K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, M. Horowitz, “Architecture and Circuit Techniques for a Reconfigurable Memory Block,” *International Solid States Circuits Conference*, February 2004.
6. L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, J. Davis, M. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun, “Transactional Memory Coherence and Consistency,” *31st International. Symposium on Computer Architecture*, June 2004.