

Polymorphic Architectures I

Lecture #7: Tuesday, April 22 2003
Lecturer: Jing Jiang and Honggo Wijaya
Scribe: Chi Ho Yue and Rohit Gupta

We are entering an era of ubiquitous computing, as technology scales, more and more applications demand ever-growing performances. Yet the design complexity grows as well. In addition, high non-incurring fabrication cost and manufacturing delays demand chips to be sold in large volume, thus targeting a larger market to be cost effective. How can we achieve performances comparable to customized solutions in a single chip design? The answer is Polymorphous Architecture.

One thing is for sure, interconnect is going to be a big issue and therefore any kind of architecture needs to be scalable in terms of wires. We will look at 2 particular solutions today, Smart memories and TRIPS.

1 Paper 1: The TRIPS Multiprocessor

TRIPS processor consists 4 out-of-order, 16-wide-issue Grid processor cores, which can be partitioned to exploit different types of parallelism. It uses software scheduler to optimize for point-to-point communication.

It's a block-oriented system in all modes of operations, namely hyperblocks. Programs are compiled into large blocks of instructions with single entry point, no internal loops and possible multiple exit points. Each block has a set of state inputs and a potentially variable set of state outputs that depend upon the exit point from the block. The compiler is responsible for statically scheduling each block of instructions onto the computation engine.

Each node of the grid processor consists of an integer ALU, A floating-point unit, a set of reservation states. Each node can forward the result to any of the operands in the local of remote reservation states within the ALU.

TRIPS processor has the following resources to achieve configurability. First frame space is the reservation stations with the same index across all nodes. Next is the register file banks, which are used for speculation or multithreading etc, depending on the mode of operation. Block Sequencing controls has various policies for different modes. For example, deallocation logic maybe configured to allow a block to execute more than

once, as is useful in streaming applications. Also memory tiles can be configured as scratch pad memory, synchronization buffers etc.

The strength of this paper is that the processor can deal with a mix load of parallelism, at least it claims. However, the performance numbers are done with a perfect memory in mind which is not usually the case in a real world. The overhead of speculation hardware can not be underestimated either.

2 Paper 2: Smart Memories

2.1 Summary

This paper proposes Smart Memories as a partitioned, explicitly parallel, reconfigurable architecture for use as a future universal computing element. By using Smart Memories, the appearance of the on-chip memory, interconnection network and processing elements can be tailored to better match the application requirements.

Smart Memories contains an array of processor tiles and on-die DRAM memories connected by a packet-based, dynamically routed network. In order to get more computation power than what is contained in a single processing tile, four processor tiles are clustered together into a "quad" and a low-overhead, intra-quad interconnection network is provided. By grouping the tiles into quads also makes the global interconnection network more efficient by reducing the number of global network interfaces.

A Smart Memories tile consists of a reconfigurable memory system, a crossbar interconnection network, a processor core and a quad network interface. Having a reconfigurable memory system is important since different applications have different memory access patterns. The crossbar interconnection is used to connect memory mats to processors or the quad interface port. The processor itself contains integer and floating point clusters, local register files and shared FP register file to provide the necessary bandwidth. Each tile can sustain up to two independent threads. Smart Memories also allow for reconfigurable instruction format and decode.

2.2 Results

Smart Memories mapped really well to two different machines on far ends of the architectural spectrum which require very different memory systems and arrangement of compute resources. The first machine is Imagine, a highly-tuned SIMD/vector machine optimized for media applications with large amount of data parallelism. The second one is Hydra, a speculative multiprocessors that supports application with irregular accesses and communication patterns.

2.3 Critique

Smart Memories is a modular architecture that can address the scalability issues due to wiring delay. Its reconfigurability made Smart Memories able to map two machines at opposite ends of the architectural spectrum with only modest performance degradation.

The main critique is this paper doesn't describe how complex it is to map certain architecture to Smart Memories.

2.4 Future Work

The future work of this paper is to create a more complete simulation environment to look at the overall performance of some complete applications and to investigate the architecture for inter-tile interactions.

3 Class Discussion

What grain of configurability is optimal? Fine-grained configuration is good for limited numbers of applications. The extra flexibility comes at the cost of extra configuration and complexity overhead. The decision on deciding the granularity is essentially a tradeoff between these two factors.

Is there a specific example of what architecture is better? Both coarse and fine grain approaches can exploit all three kinds of parallelism: ILP, DLP, and TLP, i.e. there is no obvious reason to choose any given architecture. To take the advantage of random ILP, you need wide issue VLIW processors (TRIPS attempts to do this). Smart memories is geared toward DLP. The next two sections discuss the TRIPS and Smart memories design in more detail.

3.1 TRIPS

One of the most notable points of this paper is the assumption of having a "magic" memory block that guarantee perfect memory fetches and handles all memory disambiguation. Obviously, this is an unrealistic assumption - often, one of the major performance bottleneck in modern system is the memory latency.

Further, the paper claims that the TRIPS processor is capable of exploiting DLP and TLP, whereas in the implementation, everything is converted to ILP. The performance results presented are mostly for applications which exhibit DLP. This kind of applications will run better on a vector or stream processor such as Imagine.

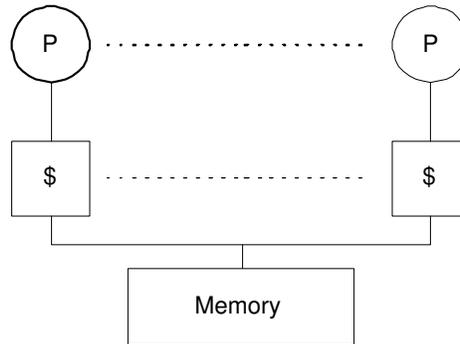


Figure 1: Processor/Cache/Memory Configurations

3.2 Smart Memories

How does one support different combinations of the above processor/cache/memory organizations? Is that necessary?

Depending on each application, the ideal configuration is different. It would ideal to support all possible formations. But beyond a certain point, it becomes a problem of diminishing return with the extra topologies only providing minimal performance gains

As an example, we discuss how SMART memories would have to be adopted to handle a second level of coherence caches. Currently, L2 cache is present on a per-quad basis. How would this change if L2 is shared across multiple quads? In a re-configurable scheme, we need some sort of mapping for every access scenario since the processors must know how to handle memory requests. This limits the complexity and number of mappings that can be supported.

In order to support distributed L2 cache, the inter-quad connections need to provide the required bandwidth for inter-processor connection. It also needs to support some kind of coherency protocol. An "easy" way out would be to use message passing to explicitly share data without worrying coherency. One way to implement coherency is to use a directory based approach. Every processor can search the directory to locate the required page. Generally speaking, the directory size is proportional to the main memory size. In order to avoid each quad have a large directory that can take up significant portion of the L2 cache, we can implement some sort of distributed directory scheme. Every quad would be assigned a range of addresses for which that quad is responsible to store directory information. By this scheme, every processor knows where to send its memory requests and the system can avoid duplicating pages. An alternate to distributed directory would be to store the directory in a memory access controller at the memory interface. This could potentially simplify the solution at the cost of increasing latency.

3.3 FPGA

Perhaps the FPGA can be viewed as the "ultimate" polymorphic architecture since most FPGAs are bit-re-configurable. However, FPGAs today are moving toward a more block-oriented approach with shorter programming times at the cost of reduced flexibility. Amongst the major challenges faced by the FPGA is having a good programming model and associated compilers that can efficiently utilize its resources.

3.4 Compiler

Compilers used in multithreaded environments require the capability to extract hyper blocks and perform some (hopefully aggressive) amount of speculation. A hyper block is defined as a chunk of code that is executed sequentially and there is only one point of entry into that particular block. There may be multiple exit points. The advantage to this scheme is that we only have to figure the dependencies (data and control) within a single hyper block. Further, this should reduce I cache misses since a large number of instructions is already known. Ideally, we want to make hyper blocks large, but it is hard to find such contiguous sections of code in normal programs. One method to make commonly executed streams of code into a large block, and the rare cases where execution comes in mid-way through the block has to be handled by replicating relevant sections of code. Again, this is a tradeoff between code-size and potential performance gains.

In order to efficiently utilize DLP, we need parallelizing compilers that go beyond the usual 4-way (or larger) VLIW design and can parallelize arbitrary loops and tasks. An important consideration for any compiler design is obviously the programming model. For example, depending on programming model, one can use message passing or memory-based synchronization. This will affect the compiler's design and its abilities.

The programming model is highly dependent on the kind of applications running. Furthermore, any given application of medium complexity may have different sections that exhibit very different behavior. For example, a visual or speech recognition application will have a DSP module and recognition module. A DSP algorithm typically exhibits a lot of DLP, whereas the recognition may be done using table-lookup matching type schemes which involve random memory accesses. Efficiently programming these components would typically use drastically different programming models

Ideally, the compiler should be able to figure out all types of parallelism. However, compilers today are not smart enough to do that. Even optimized compilers are dependent on good programming semantics to extract ILP for VLIW processors even though these compilers have been used for decades. Needless to say, we need a good programming model that will explicitly expose TLP and DLP. Popular high-level languages such as C do not do that. Further, a question that needs to be addressed is what level of familiarity

a programmer should have with the hardware architecture. Even now, "serious" parallel programmers have to know everything about the underlying hardware

4 References

1. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. Proceedings of the 27th International Symposium on Computer Architecture, Vancouver, BC, June 2000.
2. D. Burger, S. Keckler, et al. Exploiting ILP, DLP, and TLP Using Polymorphism in the TRIPS Processor. Proceedings of the 30th International Symposium on Computer Architecture (ISCA), San Diego CA, June 2003.