An Introduction to Reconfigurable Architectures

Mike Phillips

March 28, 2000

6.911: Architectures Anonymous

with Tom Knight

Reconfigurable architectures are computer architectures which employ reconfigurable hardware, which can be changed after fabrication to perform a desired task. The dynamic nature of these hardware applications is generally achieved with some sort of "reconfigurable array," such as a Field Programmable Gate Array (FPGA). These devices have the property that they can be programmed to create arbitrary combinational logic circuits. The benefit of such technology, and the reason that it is of interest in computer architecture research, is that the array can be altered to create an application-specific piece of hardware to quickly solve the problem at hand. Although not as fast as an Application-Specific Integrated Circuit (ASIC) designed to perform the same task, reconfigurable logic has the advantage of being able to change so that it is the right application-specific piece of hardware for any number of applications rather than just one.

The Concept

The key concept here is maximizing computational density, the ratio of raw computing power per unit area of silicon. Traditional microprocessors devote most of the available area to caches and to the control of a small number of functional units. These functional units are the pieces of hardware which actually perform computation. A great deal of fuss is made in such an architecture about reusing these small (area-wise) units to make computation happen. Computation is pipe-lined temporally– a computation is broken in to sequential pieces which take turns executing on the hardware. Results from earlier parts are passed on to successive parts via registers, memory, etc.

Another approach is to pipeline the computation spatially. Split your problem into parts and place hardware to solve each part on your silicon. Results from previous stages are passed to the next stages using wires. If your silicon is actually a reconfigurable array, then you can customize the silicon in this manner for each computation that needs to be performed. Thus, the potential exists to maximize the area being used for actual computation, resulting in an order of magnitude more raw compute per unit area over microprocessors.

Suitability

Of course, this approach does not work that well for all problems. It is necessary that the problem being solved matches the architectural assumptions. This approach is all about bandwidth– if the problem can be split into enough pieces of relatively small size, then the system can be clocked at a high rate, and data will pour forth. If the design cannot be adequately pipelined, then performance will suffer. Additionally, the hardware needed to solve the problem must actually fit into the reconfigurable array spatially. Also, the solution to the problem should involve regular, repeated operations. The array, once configured, should run for several cycles, always performing the same computation. Reconfiguration is an expensive operation. If it is the case that there are a large number of infrequently used operations, then a lot of area may be spent on implementing these functions, but rarely used, drastically decreasing the computational density.

Another issue is the width of the data being acted upon. Modern microprocessors with their wide data paths waste a lot of area shuttling unused bits around when asked to act on data at the bit or byte level. Reconfigurable solutions are ideal in such cases, though, since only as much logic is as needed for the width of the data need be implemented.

The most important factor in determining if a reconfigurable architecture will do well trying to solve a problem is the presence of sequential dependencies. If there are large number of operations that must be completed in sequence, then spatial pipelining may be difficult to accomplish, in terms of breaking the problem up into pieces and in terms of fitting all the pieces together. In these situations, traditional microprocessors perform very well, while reconfigurable architectures do not.

Examples

Field-Programmable Gate Arrays

FPGAs are commercially-available reconfigurable arrays produced by companies such as Xilinx and Altera. They can be visualized simply as a "sea of wires and logic bits" that can be wired together. More specifically, there are programmable switches for interconnect, and look-up tables and multiplexors for performing bit-level computations. The majority of the area spent in implementing an FPGA is devoted to interconnect.

Over time, the densities and speeds of these arrays have been steadily increasing, and have reached the point where non-trivial amounts of hardware can be programmed into the array to solve a problem.

Typically an FPGA is used as follows. A hardware design is compiled down to a set of configuration bits that tell the FPGA how to wire itself up. This compilation is often very compute-intensive, as current FPGA designs require alot of work to get designs to "fit" into the array in a way that everything can be wired up to whatever it needs to be connected to. Programming the array is a matter of sending these bits into the array, typically via a 1-bit or maybe 8-bit wide path. Thus configuration ends up being a relatively expensive operation.

Please note that FPGAs are not the only way to implement reconfigurable arrays. There are several other custom implementations, as we shall see, which are variations on this theme.

High Speed Reconfigurable Array

The goal of the HSRA project was to overcome a feature found in many FPGA designs: low clock frequency. This array's key innovation is the pipelining of the interconnect paths to allow an increase in clock rate. The designers also added more flip-flops to their array versus a typical FPGA so that the hardware needed to do the pipelining would be available. The design was implemented in a 0.4 micron process and clocked at 250MHz. The result was that at a cost of less twice the area, a factor of between two and seventeen improvement in performance was observed, as compared to FPGAs. One catch, of course, is that if there was not a lot of parallelism in the logic design, then the extra hardware to do the pipelining slowed things down as compared to the unpipelined (FPGA) version. The designers concluded that this was a "pessimistic upper bound" because they felt that they could further decrease the latency and area cost of the design in a future version.

MATRIX

The designers of MATRIX started by observing two things. First, microprocessors have low computational density and wide data paths. Second, FPGAs are efficient went functional diversity is low due to a lack of resources for distributing instructions. They then decided to pursue a design where it is possible to tailor how much chip resources are spent on control of computation resources versus active computation to the application and available hardware.

The fundamental building block in the MATRIX array is the Basic Functional Unit (BFU). Each BFU is a 256x8bit memory, an ALU, and control logic. The array can be configured as a register file, in which state is is a dual-ported 128x8bit memory that can perform two reads and a write each cycle. The ALU is equipped with a configurable carry chain to speed up computations on data that is several bits wide. The key concept is that each BFU can be deployed as needed– as computational datapath, as memory/cache, or as control circuitry. A further optimization is the ability for each BFU to cache a few configurations, so that the cost of reconfiguring the array for a few common tasks is greatly reduced. One other interesting feature of MATRIX is that BFUs could reconfigure each other, rather than always being programmed from an external source.

Coprocessor Approaches

The GARP and Chimera projects mate a typical microprocessor with a reconfigurable array. The idea here is to program the array, or at least part of it, to perform some common task. When it is time to execute that task in the instruction stream, the array executes the task. The Chimera project actually allows a programmer to allocate N rows in the array to implement a particular operation, allowing the array to have several different operations available at any moment. The array will also cache configurations, like MATRIX does, to reduce reconfiguration costs. The GARP design adds several instructions to the ISA of a typical MIPS processor to directly interact with the array. Both approaches require that the array be reprogrammed with some regularity, as it is used to compute some portion of a problem.

Tools

In reconfigurable architectures, at some point the array needs to be configured in order to have to perform the task at hand. In some systems, such as with FPGAs and with the Teramac, a hardware design is compiled down and fed into the array. A compiler for a reconfigurable architecture could actually take as input a program in a high level language and decide how to setup the array. GARP's compiler does this, selecting certain loops to implement in the array, and generating the bit-stream that will need to be sent to the array when it is time to program the array. The Teramac, which is a reconfigurable peripheral accessible via a SCSI bus, included software tools to debug a design on the array, by setting breakpoints, stepping the array one cycle at a time, et cetera. In general, tools for converting high level language descriptions of a computation to a hardware design or bit configuration for an array are not yet mature.