FPGAs and CPUs

By: Steven Stanek
Robert El-Soudani

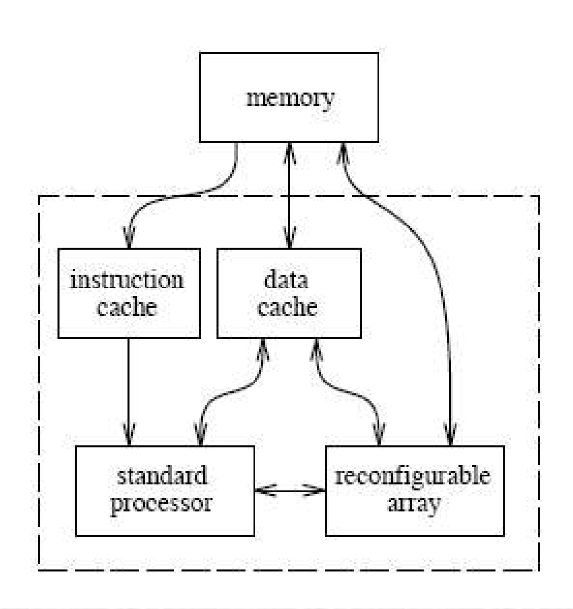
In which we inspect how technological changes over the past 10 years affect performance results of FPGAs paired with CPUs.

Motivation

- FPGAs can be used to create application specific circuits for specific problems quickly.
- Most programs are far too large to be put entirely on FPGAs.
- Many programs spend much of their time in certain time consuming sections
  - We can find this code using profiling tools such as gprof and Shark
- We can pair FPGAs with CPUs:
  - FPGAs execute small time consuming functions
  - CPUs execute the large amount of code which consumes little time
  - For some small algorithms, the entire algorithm can be implemented on the FPGA

The Garp (1997) Implementation

- Designed to interact with a MIPS-II CPU through both ISA and memory
- 2 bit logic blocks
  - Supported several modes including: Several kinds of LUTs, 3 Adder, variable shift and MUX
  - 64 bits of configuration data per blocks
- 24 blocks wide, at least 32 tall
  - Supported a scheme for expanding in vertical
  - 16 Center blocks (32 bits) were used for computation
  - 7 Blocks for overflows, rounding, “whatever is needed”
  - 1 Special control block for I/O
- Interconnect was asymmetric: designed for data to flow from top to bottom more or less linearly



Design Points

- Even when staying with a GARP like design, there are many potential design points:
  - Increasing either vertical or horizontal interconnect
    - Makes it easier to shift, permute or use multiple rows
  - Increasing the width or height of the array
    - More Hardware = array can handle more complex problems
  - Modifying permitted connections within a clock cycle (forced superpipelining?)
    - Can affect cycle time
  - Modifying memory access methods
    - 64 bit memories today
  - Changing the contents of a block
    - Do more (or less?) with a block. Cycle time constraints.
  - Configuration techniques for the array
    - As Array size grows, time to configure increases

Our Proposed G05 Implementation

- Modern processors have 2^6 to 2^7 times the transistor count of those in 1997
  - Consistent with predictions of Moore's Law
- Our choices
  - 4 x Width: 72 cells (total), 64 (128 bit) for computation plus a few others
    - We needed at least 32 blocks (64 bits) for 64 bit memory accesses
    - Examination of our applications showed that the extra 64 bits were often useful
  - 128 blocks High (4 times the minimal GARP 1995) value
    - Support for larger amounts of code
  - Wider (and thus more) horizontal interconnect
    - Original horizontal interconnect scheme doesn't scale well: shifts and permutations require many more wires at 128 bits than 32 bits
  - Minor Blocks Changes

Control, Contexts and Memory

- Contexts, allow for many states to be stored in array
  - State in each blocks=64 bits of configuration data + 4 bits in registers
  - Most hardware is invested in interconnect and static logic, not state
  - Can be used for either interthread context switches or intrathread blocks
- Like original GARP, ISA includes instructions for:
  - Programming the Array
  - Starting and stopping the Array
  - Switching Array Contexts
  - Storing Values in registers on FPGA
- Memory Accesses: left 32 blocks generate 64 bits addresses
  - The Array is connected to the L1 cache
  - Blocks can be marked to read into both, one or neither of their registers
  - Stalls on L1 misses or too many outstanding accesses

CPU vs. FPGA Then & Now

- CPUs have also evolved over the past ten years
  - More ILP since 97 (that was the age of the original Pentium)
  - The clock speed ratio of FPGA to CPU in the original paper (a little < than 1:1) is now closer to 1:8
- Compiler techniques may also allow CPUs to better exploit parallelism
- Our FPGA: More blocks better interconnect
- Original GARP result: “speedups [range] from a factor of 2 to as high as a factor of 24 for some useful applications”
- FPGA Hidden Costs:
  - Stalls on Memory Accesses
  - Load times into the array

Application #1: Blowfish Encryption

- The algorithm: 16 iterations of a loop with 5 memory accesses per iteration.
- Our implementation of inner loop: 10 pipeline stages, 21 rows per iteration.
  - Total of 20 memory accesses per cycle, results in stalls
    - If we can service 20 memory accesses per cycle: 1 block every 4 cycles
    - If we can service 8 memory accesses per cycle: 1 block every 12 cycles
    - If we can service 4 memory accesses per cycle: 1 block every 20 cycles
- Benchmark results (using author's own code):
  - 1 block every 250 cycles on G4
  - 1 block every 450-500 cycles on a Athlon XP

Application #2: GZIP Compression

- The Algorithm: repeatedly scan for longer pattern matches within a window – in hindsight, not the most ideal candidate for fpga
- Our Implementation:
  - 2 stage pipeline to examine 16 bytes at a time
  - no real algorithmic changes, very simple, could definitely do better
  - Uses 8 rows and around 142 fpga cycles to program
- Analysis:
  - Executed gzip on freeBSD machine with AthlonXP 1.53Ghz. Used cycle counters to estimate lone CPU performance and wrote C sim code for FPGA.
  - Compressed bmps, tifs, large text, small text, highly compressible, not very compressable and took the average

Application #2: GZIP Results:

- 90.4% of execution spent in particular loop
- Averaged 4.98 fpga_cycles per 180.21 cpu cycles
- Using 1:8 CPU:FPGA clock
- Raw speedup of loop averaged 4.53
- Expected speedup of entire program over 50MB with 1 time programming averaged 4.0908. With 100 programs 4.0897
- Expected speedup of entire program over 408B text with 1 time programming averaged 3.14. At 9 programs speedup was 1.075.
- For 11MB repetitive text, speedup even at 100 reprograms was 10.70.