

Reconfigurable Computing 3: Programmable-Reconfigurable systems

ECE 497NC Lecture 8, 2/13/2004

Lecture Readings

- *CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit* by Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. Appeared in the 2000 International Symposium on Computer Architecture
- *Garp: A MIPS Processor with a Reconfigurable Coprocessor* by John R. Hauser and John Wawrzynek. Appeared in the IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
- *Clustered Programmable-Reconfigurable Processors* by Derek B. Gottlieb, Jeffrey J. Cook, Joshua D. Walstrom, Steven Ferrera, Chi-Wei Wang, Nicholas P. Carter. Appeared in the 2002 IEEE International Conference on Field-Programmable Technology.

Lecture Overview

This lecture presented the rationale behind systems that combine programmable processors and reconfigurable units, usually on the same chip. Of the various different ways to do this, three paradigms as illustrated by research prototypes Garp, CHIMERA and Amalgam were discussed.

Reconfigurable-Programmable processors: Why?

Reconfigurable hardware has shown impressive performance, sometimes orders of magnitude better than comparable programmable processors, on applications ranging from RSA decryption to Signal Processing. However, it is doubtful that such hardware will completely replace programmable processors for general-purpose computing. There are a number of reasons for this:

- FPGAs are usually not large enough to implement whole programs without the need for reconfiguration. Reprogramming an FPGA, at least for current generation devices, is very time consuming, specially when the function to be programmed on the FPGA will be executed for a very short period of time (i.e., when the configuration time is not amortized by the number of times the code is run)
- Programmable processors use a very dense encoding for instructions compared to the encoding in FPGAs. This is because most processor instructions are SIMD-like instructions that perform the same function on a large number of bits. While specifying the configuration for FPGAs, each logic block has to be programmed individually. For example, a 32-bit add on a processor will typically take a 32 bit instruction, while the same add will need 60 bits of configuration data on an

FPGA. However, coarse-grain reconfigurable systems (like PipeRench) may not suffer from this drawback.

- Certain instructions, like floating-point operations, are slow and take up a lot of area when implemented in an FPGA. It may be better to implement these in custom ALUs in a processor
- Some functions (notably I/O) don't lend themselves amenable to implementation in reconfigurable hardware

While reconfigurable hardware as a complete computing machine is not very promising, combining it with programmable processors in a single system has a lot of potential:

- Reconfigurable systems seem to execute certain compute-intensive kernels very well, specially when such kernels fit on an FPGA without the need for reconfiguration.
- Most real-world programs follow the 90-10 rule: 90% of the time is spent executing 10% of the code. Speeding up this 10% of the code can reduce the overall execution time greatly. Further, the 90-10 rule seems to be recursive; out of the important 10% of the code, one-tenth accounts for a large proportion of the 90% of execution time that this code takes.

Therefore, mapping the critical regions of programs on reconfigurable logic seems likely to improve the execution time.

Paradigms for Reconfigurable-Programmable systems

In general, there are two ways to achieve a partitioning of code between reconfigurable and programmable processors:

1. Fine-grained partitioning:

In this technique, small groups of instructions (sometimes as small as two instructions) are mapped on the reconfigurable part. Such a partitioning means that the reconfigurable and the programmable processors will be talking often to each other (i.e, lots of communication). This approach can be expected to have good speedup on a wide range of applications. For example, almost all applications are likely to have groups of instructions with data-dependencies that will be benefited when there is no need of storing intermediate data explicitly. A typical example of such a system is CHIMERA.

2. Coarse-grained partitioning:

In this method, large, computationally-intensive parts of the program are mapped onto the reconfigurable array. This has the potential of reducing communication between the programmable and reconfigurable units. However, programs whose kernels are not amenable to being mapped onto reconfigurable logic will not show a lot of speedup with this approach (for example, programs where the most often executed loops have a complicated control flow). While this approach will show staggering speedups on programs that map well to the idea, it may not show speedups across all applications. A typical example of such a system is Garp.

CHIMAERA

Overview

CHIMAERA is a typical example of a reconfigurable-programmable processor with fine grained partitioning between the two processor components. One of the driving ideas behind the project was the fact that multimedia applications are becoming increasingly important in the general-purpose computing domain. To speed up the execution of such programs, specialized vector-like instruction set extensions have been used by the industry (eg, Intel's MMX and SSE, Motorola's AltiVec). However, the need for backward-compatibility implies that changes to the ISA are permanent. Even when software needs change and the ISA extensions are not needed, the architecture will need to support them to allow legacy code to run. CHIMAERA offers an alternative to introducing ISA extensions to speed up multimedia applications: instead of adding custom vector-like hardware to the processor and introducing special instructions to use this hardware, the processor can have a small reconfigurable unit on chip. Since FPGAs are well suited to the data-parallel and bitwise operations in multimedia applications, this unit will speed up the execution of such programs. A compiled program will have instructions that call upon the reconfigurable unit to perform some computation. The difference vis-à-vis ISA extensions is that as software evolves, the design of the reconfigurable unit is allowed to change, provided its still capable of executing older instructions.

Architecture

CHIMAERA consists of a Reconfigurable Functional Unit (RFU) that is treated as an extra ALU in a superscalar. It communicates with the rest of the machine through a shadow register file.

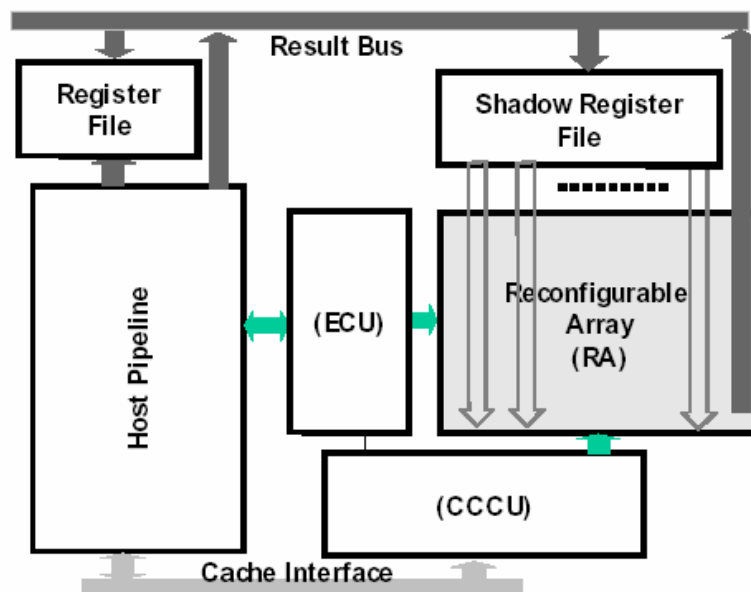


Figure 1: CHIMAERA architecture¹

¹Figure Credit - CHIMERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit, Zhi Alex Ye et al

Compilation

During compilation, special instructions called RFUOPs are inserted in the machine code at places where two or more processor instructions could be efficiently fused into a single operation on the RFU. Three specific techniques used to do this are:

- *Instruction Combination*: This compilation pass combines chains of dependent instructions that produce one output into a single RFUOP. The compiler does this by looking at the Data Flow Graph of the program, finding parts which have no control flow and produce just one output that can be mapped efficiently in the RFU. The one-output-only restriction is present because the RFU is capable of executing any operation with 9 registers as input and one output register.
- *Control Localization*: This optimization transforms pieces of code with simple control flow into a single RFUOP (using *instruction combination*). The RFU executes all control paths, and chooses the correct output at the end. It is very similar to the concept of predicated instructions, and helps programs run faster on a superscalar by reducing the number of branches.
- *SIMD within a register (SWAR)*: SWAR attempts to recognize instructions that operate on different bits of the same registers such that they can be executed in parallel. This is a “suggested” optimization; the CHIMAERA compiler doesn’t use this because it lacks an alias analysis phase to ensure correctness of this transformation

Note that the programmer still writes code in C. The compiler maps it in the best possible way to the CHIMAERA hardware. Also, the front-end of the pipeline of the superscalar remains the same. The RFUOP enters the instruction window as any regular instruction: it is signaled as being ready-to-execute when its sources have been written to. When an RFUOP is ready to execute and the RFU is not appropriately configured to execute the RFUOP, the processor stalls and fetches the configuration from memory (the configuration of all RFUOPs is a part of the program executable).

Simulation Results

Simulations of real programs on CHIMAERA show some interesting results. One of these is that the number of different RFUOPs used by a single program is rarely more than 8. This presents an excellent opportunity for caching RFUOPs so the processor doesn’t need to go to memory to fetch RFU configurations.

In terms of performance, while the speedups shown by CHIMAERA vary across different applications, most applications show *some* speedup, the average speedup being somewhere between 20 and 30 percent. Since all of these are simulation results, the designers made different assumptions about the delay in the RFU with respect to the clock cycle time of the superscalar processor; the speedup depends a lot on the actual assumptions made.

There are some reservations about the efficiency of CHIMAERA on a multitasked system, where processes are swapped in and out frequently. Since the reconfiguration time of the RFU is likely to be appreciable, and so is the time to fetch RFU configurations from memory, swapping is a cause for concern.

Garp

Overview

Berkeley's Garp processor is a typical example of a system that divides work between the reconfigurable unit and the programmable processor at a coarse granularity. It consists of a small MIPS processor loosely coupled with a large reconfigurable unit (essentially a

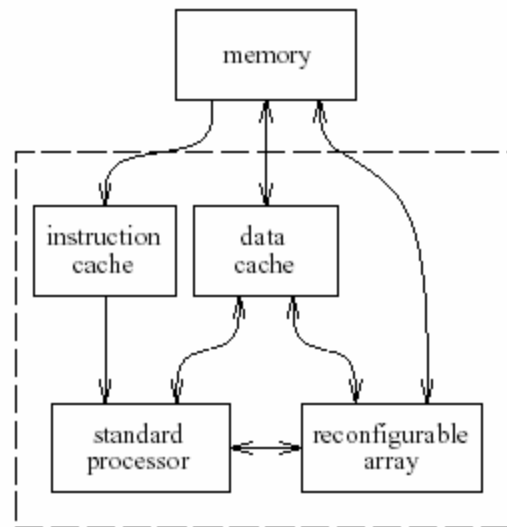


Figure 2: Garp architecture²

slave processor). These two units share the data cache, and are both capable of communicating with memory. One of the features of Garp is that the reconfigurable array has access to memory. Also, the design of the array is slightly non-conventional, with the finest granularity operations supported being 2-bit operations. The design is such that while the architecture of a single row is fixed, the number of rows in the array can grow in future generations without losing backward-compatibility. Another interesting feature is that two wires can be connected to each other only through a logic block. This has the potential to reduce interconnect area in the reconfigurable unit, at the cost of using some logic blocks exclusively for routing. The use of logic blocks for routing is one of the reasons for the presence of redundant logic blocks in each row (i.e, there are more logic blocks than would be needed for a 32 bit operation).

Compilation

Garp is programmed in C. The compiler tries to discover loops which are compute intensive and map well to the reconfigurable array. These loops are then replaced in the C code by inline assembly that initializes the array. The assembly code includes instructions that specify the configuration to be mapped to the array, and (if needed) instructions that move values from (or to) processor registers to (or from) specific places in the array (reg-move instructions). These instructions can also store a value in a special "counter" in the reconfigurable array, which is decremented once every clock cycle. The reg-move instructions *stall* till the counter becomes zero before moving data and storing a value in

²Figure Credit - *Garp: A MIPS Processor with a Reconfigurable Coprocessor*, John R. Hauser et al.

the counter. This potentially allows the programmable processor and the reconfigurable processor to operate in parallel.

The Garp compilation infrastructure is tailored for a multitasked system and has support for swapping processes. It also fits in very nicely with the regular compilation flow where programmers write code in C and use libraries extensively

Simulation results

Results of simulating Garp show impressive speedup on the applications selected for analysis. The comparison is between an UltraSPARC machine, and a Garp processor which occupies approximately the same amount of die area as the UltraSPARC, except that it has a single-issue MIPS-II processor and a reconfigurable unit. The speedups vary from about 24x on DES (Data Encryption Standard) and 9x on Image Dithering to 2x for QuickSort on an array of about 1 million entries.

Amalgam

Overview

Amalgam is a *clustered* reconfigurable-programmable processor that takes a middle-path approach to the granularity issue. *Clustered* processors are like Chip Multiprocessors (CMPs) in many ways: they group essential computation units, like registers and ALUs, locally into “clusters”. A single processor on a CMP is such a cluster: it has an independent set of registers, ALUs etc. The advantage of such a system over a monolithic processor is that clock-cycle times are determined by the critical path in a single “cluster”, long wires going across the chip don’t affect the cycle time. While clustered processors also have individual processing units and some on-chip shared memory, they differ from CMPs in having a *logically global* register file. Physically, individual processors have their own register files, but a processor can write to any register in any other processor in the chip. While reading values from registers, however, a processor is restricted to its own register file. Note that writes to registers on other processors have to go through the chip-wide network, and are expected to have latencies in excess of a single cycle. This approach has been shown to be superior to a CMP architecture where processors can talk to each other only through memory.

Architecture

Amalgam consists of 4 reconfigurable processors and 4 programmable processors on the same chip. All processors have a 32 entry register file, all programmable processors have individual instruction caches. The data cache is a banked system which is shared by all processors on the chip. Processors communicate to the data cache, off-chip memory, and to each other through the chip wide network.

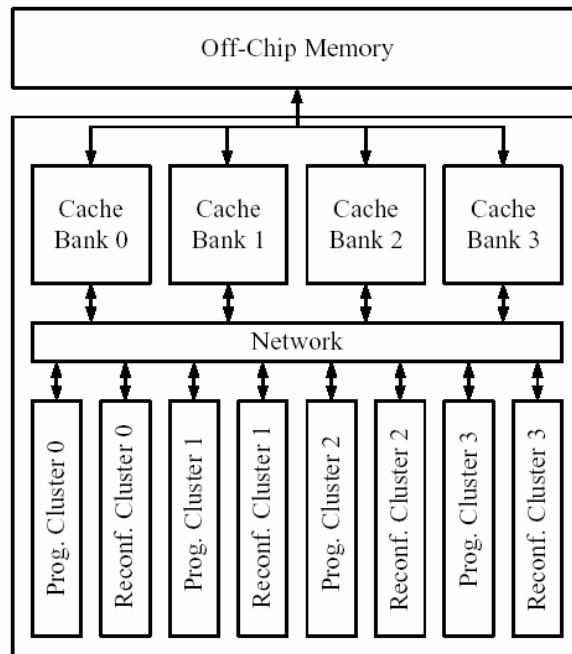


Figure 3: Amalgam architecture³

The reconfigurable units in Amalgam are interesting because of their novel design. The register file is “interleaved” within the CLB’s, instead of being present as a central resource. While this is likely to decrease the access time of the register file, compilation becomes harder for such a design, since all registers are not equidistant from all logic blocks.

Simulation results

Various applications were programmed for Amalgam by hand-mapping code to the programmable and reconfigurable processors. The relevant comparison is between a machine using $2n$ programmable clusters, and a machine using n programmable clusters and n reconfigurable clusters. All of the applications studied show an appreciable speedup when half of the clusters are reconfigurable compared to the case where all clusters were programmable. In some cases, the difference is staggering, like for the n -queens problem, where the entire chessboard could be stored in the reconfigurable cluster, dramatically reducing the number of memory accesses. Average speedup over the purely-programmable cluster architecture is 2.84.

Conclusion

Reconfigurable-programmable processors present an interesting point in the design space of general-purpose and embedded processors. It seems reasonable to assume that reconfigurable units, if and when they become a part of standard computer architecture, will need to be supported by programmable processors. Because of the need for

³Figure Credit - *Clustered Programmable-Reconfigurable Processors*, Derek Gottlieb, Jeffrey Cook, Joshua Walstrom, Steven Ferrera, Chi-Wei Wang, Nicholas P. Carter

backward-compatibility at the level of machine code, there are reservations about whether such processors will be used for general purpose computing in the near future; the embedded systems domain seems more likely to adopt this paradigm first. Exactly what the architecture of these machines will be is still an open question, three candidate-architectures were discussed in the lecture. Compilers that automatically map HLL programs onto these machines will be essential for their success in the industry, given the requirements of short time-to-market and simplified testing.