Polymorphic Architectures (II)

Lecture #8: Thursday, April 24, 2003
Lecturer: Joel Coburn, John Kim
Scribe: David Bloom, Amin Firoozshahian

1. Introduction

Continuing the study of reconfigurable architectures, two other architectures with finer grain reconfiguration are covered: PipeRench [1] and MIT RAW processor [2]. MIT RAW processor consists of sixteen single issue cores connected via a low latency communication network over the chip. PipeRench is a reconfigurable array of processing elements and programmable interconnection networks between them, targeted to exploit parallelism mostly in data parallel applications. Compared to the Smart Memories [3] and TRIPS [4] architectures, these two have finer basic blocks (PEs in PipeRench and single issue cores in RAW), which expose more architectural aspects to the compiler and demand more static scheduling of the code.

2. PipeRench: A Reconfigurable Architecture and Compiler [1]

2.1. Summary

PipeRench is a co-processor for streaming multimedia applications. It is different from the other polymorphic architectures since it is an attached processor. Also, unlike the other polymorphic architectures, PipeRench is a more fine-grain polymorphic architecture and resembles an FPGA with a coarser granularity. The granularity chosen for this architecture is 8-bit processing elements as this width provides the best tradeoff between data path utilization and complexity. Since most data elements for streaming multimedia are 8-16 bits wide, it made more sense to use a smaller granularity instead of a conventional 32bit data path.

The architecture consists of network of configurable logic and storage elements within each processing element (PE). A row of PE's creates a "stripe" within the architecture and stripes are stacked on top of each other with local interconnects between them to create the configuration fabric. Each stripe also represents a pipe stage of the hardware.

A unique feature of this architecture is that it is a "pipelined reconfigurable architecture". Even though the number of physical stripes on silicon might be limited, through the

process of virtualizing hardware, the number of virtual stripes can exceed the number of physical stripes. This virtualization brings several benefits:

1) The compiler is isolated from the hardware since compiler does not need to know how many physical stripes there are.

2) Performance that you can obtain from a given number of stripes is much greater (i.e. if you have 16 physical stripes and 128 virtual stripes, you can get 8x the performance with just 16 stripes). However, this does bring complication since it restricts the model of computation to pipelined data path, where each pipeline stage corresponds to a stripe.

Like most other polymorphic architectures, one of the critical elements of obtaining high performance is relying on the compiler. For PipeRench, a dataflow intermediate language (DIL) that is a single-assignment language with C operators was used for compilation.

The results illustrated by this paper showed that there is possibly a 10x-200x speedup on various kernels, but those results did not account for the I/O limitations of the architecture. On one particular application (IDEA), the architecture obtained 10x performance increase over a general-purpose processor as well as a custom hardware.

2.2. Critique

The advantage of this architecture is that it resolves many of the disadvantages of the FPGAs, which includes forward compatibility, rapid reconfiguration, and compilability. This architecture is also probably easier to implement in VLSI since only a single type of PE and the interconnections need to be designed. However, there are several limitations for this architecture, which is mainly related to the limited bandwidth between the PipeRench and the main processor and main memory. As a result, there are very few applications that are suitable for this architecture.

3. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine [2]

3.1. Problem:

This paper operates on the premise that future architectures will have distributed resources in order to meet the demands of greater parallelism and faster clock rates. Distributed resources require non-uniform access latencies. Instruction scheduling becomes both a spatial and temporal problem. The authors propose a compiler, RAWCC, for general-purpose sequential programs on the RAW machine. Through space-time scheduling, the compiler can exploit ILP within basic blocks.

Original Contributions:
The authors make several contributions with this paper. Most importantly, the paper illustrates a method for space-time scheduling of ILP on the RAW architecture. This method borrows ideas from the partitioning and scheduling of tasks on MIMD architectures. The paper presents a control flow model, called asynchronous global branching, based on asynchronous local branches on a machine supporting multiple independent instruction streams. Since RAW architecture has independent instruction streams, it also has the ability to tolerate timing variations due to dynamic references.

3.2. Critique:

The primary strength of this paper is the RAW compiler and its ability to compile general-purpose programs written in C and FORTRAN. By conforming to a generic programming model, designers could easily use this compiler to target the RAW machine. Another important advantage is that RAWCC allows a fully distributed processor to provide scalable ILP and handle control flow. This is only possible because the RAW architecture is fully exposed to the compiler.

A significant weakness of RAWCC is that it is only advantageous for programs heavy in ILP. The RAW architecture and compiler actually exploit TLP (rather than ILP) by using independent instruction streams across the tiles. It is important to note that this paper shows improvements only for DLP applications. Also, the compiler focuses strictly on static references and does not examine dynamic references as a means to achieve better performance. There is a phase ordering problem with the compiler where the event scheduler runs before register allocation. In this way, register pressure is not considered and may underestimate the instruction cost. The benchmarks that are used to evaluate the compiler are highly suspected because they are chosen explicitly to match the qualities of the RAW machine. Almost all the programs consist of dense matrix operations. The size of the data sets in the benchmarks is kept small in order to exploit the low-overhead communication available in RAW machine.

3.3. Future Work:

Since the evaluation methods for the RAW compiler seem biased, a natural extension of this work would be to diversify the benchmark suite and expose potential areas of improvement in the compiler design. This work could also be expanded by varying the granularity of localization of control flow to extract more parallelism in applications. Also, it might be possible to combine the merging and placement phases of the compiler to take advantage of the processor topology. Exploring the ordering phases in basic block orchestration could be an interesting area of research.

4. Discussion Notes

4.1. PipeRench

As in the previous lecture, the first point to discuss was the granularity of the reconfiguration. Namely, discussing PipeRench, it is obvious that it has finer configuration. This imposes more overhead on the hardware while trying to implement more complicated functions than PEs provide and also makes the compiler's job more difficult, since now instead of only using a functional unit, the compiler is also responsible for synthesizing it out of PEs and the interconnection network between them. In other words, now the compiler also has the responsibility of virtualizing the targeted architecture and then using it.

On the other hand, compared to FPGAs, having ALUs in form of PEs and specific registers at each PE reduces the hardware overhead and speeds up configuration time a lot. This observation led to some questions about common algorithms for placement and routing in FPGAs and how they can be accelerated. One of the suggested ways was to look up for some predefined configurations in a table and load it into the FPGA whenever such a configuration is encountered. Another suggestion to program an array of re-configurable elements was try to map simple VLIW cores at each row and instead of having specific architectural registers, pass the results directly between them.

One main question brought up during this part of discussion was the difference between configuration and programming. Two main definitions given were:

- Configuration means setting up the environment (data path) while programming means how to use it.
- Changing a state every cycle means programming while keeping it for several cycles is defined as configurations.

It was noted that the difference between these two is often not very clear, and they are in fact often confused. For instance, the configuring that is described in the PipeRench paper actually controls changes in the configuration on a cycle-by-cycle basis, so it should probably more correctly be referred to as programming.

As mentioned before, in PipeRench, functionality of each pipe stage changes in every cycle. This adds to the complexity of the compiler, since it has to decide about all these changes statically at compile time and know which stage is going to perform which part of the functionality before running code on the fabric. On the other hand, in systems like Smart Memories, the distinction between compiling the program and configuration concept is more realizable.

Another issue brought into attention was exception handling in architectures like PipeRench or RAW. Or, a more fundamental question would be if it is needed to support precise exceptions or not. The difficulty is by spreading out execution cores over the chip and requiring explicit communication between them, original order of the execution is

lost. But the advantage in such architectures is that since they have ignored virtual memory issues, precise exceptions are not explicitly required.

Also, in PipeRench paper, the authors did not mention anything about memory and its hierarchy. The importance is that feeding all the processing elements with data requires enough memory bandwidth and it can potentially be a bottleneck. While in some architectures like Imagine this bandwidth and its hierarchy are emphasized, it is not mentioned in the PipeRench architecture at all.

4.2. RAW

Returning to discussion about exceptions, we notice that currently precise exceptions are supported at the instruction level. An interesting experiment would be to change the level of preciseness, e.g. supporting precise exceptions at basic block boundaries. For supporting such a scheme couple of requirements comes to mind at first glance:

- Recording exception boundaries
- Not overwriting inputs of the block until it is guaranteed that block commits (more like a Macro-History buffer)
- Number of load/store instructions in the basic block can be a limiting factor

Similar to the granularity discussion about the various configurations, it was noted that with coarser granularity, implementations may be simpler (in this case, precise exception handling is more feasible if done at block level, as opposed to the instruction level, since the whole block can be replayed in case of exception).

Discussing different aspects of the RAW architecture, one of the interesting observations was that RAW somehow acts as a dual for the TRIPS. While TRIPS tries to turn thread and data level parallelism to instruction level parallelism by executing them in the frames, RAW supports data and instruction level parallelism by explicitly distributing computation to different processing cores and run them as different threads. This implies fast communication between processing cores, which is taken care of by the low latency on-chip network. The compiler puts all statically predictable communications on the static on-chip network and ordering is never changed in this network. The difficulty then would be adapting this method of exploiting parallelism (converging DLP and ILP to TLP) to applications with more dynamic code. In fact, the authors do not evaluate ILP programs to see how much this scheme might be capable for applications with considerable data dependent branches or pointer chasing sections.

As a side discussion, decoupled architectures were brought into attention by the class. The main idea behind decoupled architectures is to separate the stream of instructions in the program to two semi-independent instruction streams: Load/Store instruction stream and arithmetic instruction stream and make the first stream run faster in order to bring in data required for computation before it is actually required. As figure 1 shows, the communication between these two instruction streams are done via queues. This leads to an out of order issue machine, which uses queues instead of register renaming. This

approach is realizable in RAW, since processors are not lock step and one can run ahead of others in executing instructions. This is a cheap latency hiding technique, much simpler to implement than a scheme like Tomosulo's algorithm.

Another interesting question was about necessity of adding storage elements, namely registers, along with processing elements like ALUs. While adding more ALUs to an architecture means that we're trying to run more operations in parallel, providing operands for these operations puts more traffic and load on the register file. Therefore it is more preferred to distribute this register file and put it close to the execution units rather than having a central register file. For example, such an approach is seen in super scalar machines by adding reservation stations in front of functional units.

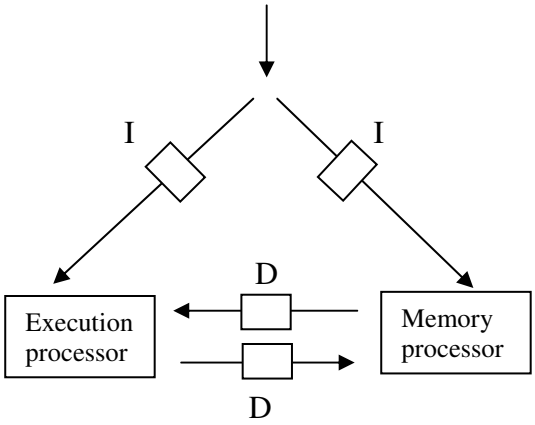

Figure 1. An illustration of decoupled architecture. Structures with label "I" are instruction queues while structures labeled "D" are data queues

Returning back to the software issues, order of optimizations/analysis done by the RAW compiler was an interesting point mentioned. The conclusion was that compiler can first assign data to processing cores and then try to schedule code on each of them, or vice versa. In fact, compilers today try to do register allocation and instruction scheduling at the same time or repeat instruction scheduling after doing register allocation in order to do better overall scheduling.

4.3 Overall summary of reconfigurable architectures

In general, it is seen that all the architectures pay the price of configurability in communications between elements by burning more power. Also, it was made clear that they can never reach the performance/efficiency levels of the custom hardware, while efforts concentrate on reducing this gap as much as possible.

Among the specific architectures studied, conclusions was as follows:

- PipeRench: Mostly suitable for data parallel applications.

- TRIPS: Mostly extracts ILP, has mechanisms for speculation and running single threaded applications efficiently. It mostly requires a VLIW like compiler for code generation.
- RAW: Efficient for thread parallel applications, requiring a MIMD/parallelizing compiler.
- Smart Memories: It's not yet well determined which type of parallelism it can exploit better.

5. References

1. S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. IEEE Computer, Volume: 33 Issue: 4, April 2000.

2. W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, CA, October 1998.

3. 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.

4. 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.