Reconfigurable Instruction Set Processors: An Implementation Platform for Interactive Multimedia Applications

Francisco Barat, Murali Jayapala, Pieter Op de Beeck, Geert Deconinck
ESAT/ACCA, K.U.Leuven, Belgium.
f-barat@ieee.org

Abstract

Future interactive multimedia applications are characterized by a large variety of compression algorithms with highly parallel nested loops. It will not be efficient to design custom processors suitable for this wide range of applications due to the uncertainty on what is going to be executed. Instead, we must find ways to cope with such dynamic and compute intensive tasks. Reconfigurable instruction set processors can cope with this dynamism by specializing the hardware to the algorithm at hand at runtime. They achieve this thanks to a flexible fabric of coarse-grained processing elements that can be reconfigured to perform different complex algorithms. This paper analyzes the performance improvements obtained by such programmable structures and discusses some of the critical issues, such as reconfiguration times.

1 Introduction

Future interactive multimedia applications will be based on standards like MPEG-4 [1]. Using an object-based approach to describe and composite an audiovisual scene, MPEG-4 combines many different coding tools not only for natural audio and video but also for synthetic objects and graphics. Objects are coded and transmitted separately and composed at the decoder side, letting the receiver interact and influence the way the scene is presented on the receiving display and speakers. Due to this user interaction, the number and type of decoders that needs to be implemented on the system is not known at design time, but rather at run time [2].

This fact forces the designer of platforms for these applications to use new approaches. Traditionally, multimedia applications have been implemented on custom VLIW processors which provide enough parallelism to accelerate this computation intensive applications [3], while at the same time retaining a low power consumption (when compared to other parallel approaches such as superscalar processors). Furthermore, in order to increase even further the computational power of these devices, they have been enhanced with custom hardware for acceleration of the most common multimedia operations. An example of this is the Trimedia processor [4], which contains specialized units for DCT (Discrete Cosine Transform) and motion estimation.

Unfortunately, due to the variety of algorithms that can be used in new interactive applications and the fact that the actual number and type of objects is not known till run time, it is no longer economically viable to make specialized functional units for each algorithm. The picture is further complicated if we also take into account that a platform designed for these applications may have to decode an object encoded with an algorithm for which it was not conceived. Therefore, in order to maintain power efficiency and the real time constraints, we need a platform that can be specialized at run time to the algorithm at hand. A platform based on a reconfigurable instruction set processor provides this run time specialization.

When designing such a reconfigurable processor, loops are the initial optimization target since typical multimedia algorithms spend most of the time on nested loops. Figure 1 shows the percentage of time spent inside the inner loops of some applications taken from the Mediabench [5] set of benchmarks when compiled with our experimental compiler. Only loops that could be software pipelined with our compiler are shown. With adequate program transformations and a better quality compiler, the percentage is even greater. As can be seen on the figure, typically more than 50% of the time is spent in inner loops.

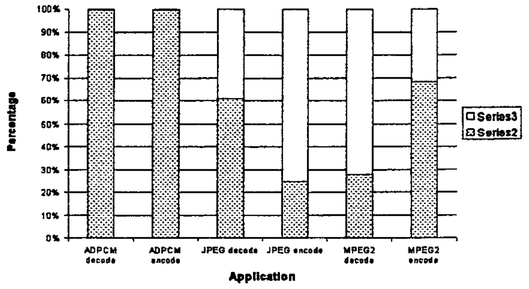

Figure 1 Percentage of loops in some applications

Another interesting detail is the amount of parallelism available inside these loops. If we analyze the loops of the previous applications, we can calculate the maximum instruction per cycle. A typical value is over 20 instructions per cycle, much larger than what current VLIW processors can exploit.

For the rest of the code, the instructions are mostly sequential, with little or no instruction level parallelism. These correspond to data dependent control code. An example of this is the 32% of non-loop code on MPEG decoding seen on the previous figure. This corresponds mainly to the variable length decoding of symbols, which is a purely sequential task. Typical VLIW processors have enough parallelism for these parts of the application.

In order to optimize these applications, a processor that can exploit all the parallelism in the loops and at the same time reduce the power consumption is needed. Such an approach can be based on a reconfigurable processor. In this paper we present CRISP, a reconfigurable processor designed for these applications.

A reconfigurable processor executing an interactive application needs run time reconfiguration (also known as dynamic reconfiguration) in order to adapt to the algorithm that is required at the moment. This means that the processor must be reconfigured for each part of the application, and this must be done at run time. However, reconfiguration times in most reconfigurable processors are not negligible [6]. Hence, a processor designed for interactive applications needs a mechanism to reduce or hide these times. We show that a small configuration memory significantly reduces the configuration times.

The paper is organized as follows. Section 2 describes the processor. The next section explains how the reconfiguration penalty can be solved or hidden with a simple mechanism, the configuration cache. Results from some simulations are shown in section 3 and conclusions are given on section 4.

2 A reconfigurable processor for interactive multimedia applications

In this section we describe CRISP, which stands for Configurable and Reconfigurable Instruction Set Processor. CRISP is a VLIW instruction set processor that is configurable at design time. It is configured by fixing parameters such as the number and type of functional units or the size of the register files. The processor is also reconfigurable at run time thanks to a coarse grained reconfigurable fabric. This flexibility allows us to perform experiments and optimize the processor to a set of applications or application domain.

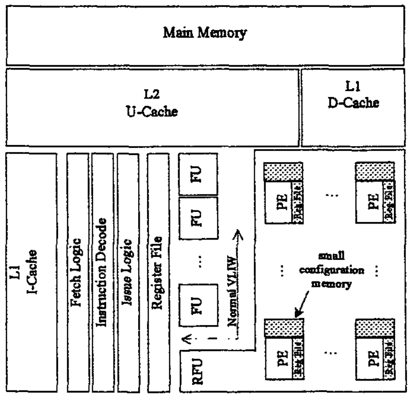

Figure 2 Typical CRISP processor instance

CRISP is composed of fixed functional units (FUs) and a reconfigurable functional unit (RFU). An example instance can be seen on Figure 2. As all VLIW processors, the processor executes instructions that are composed of parallel operations. Operations are executed in the functional units (both fixed and reconfigurable).

The fixed functional units are the functional units found in typical VLIW processors, such as integer units, multipliers or a load/store units. These are used outside loops, where not much parallelism is needed. In a sense, if we only used the fixed functional units, CRISP would behave like a standard VLIW processor.

2.1 The reconfigurable logic

The reconfigurable unit contains an array of coarse-grained processing elements (PEs). Since the array is designed for multimedia applications, it must efficiently perform the basic operations found in this type of algorithms. The processing elements therefore operate on 8, 16 and 32 bit data. The basic operations are addition, subtraction, multiplication and shifting. This means that the processing elements have a complexity similar to the integer functional units.

In order to support loops with control code inside it, the processing elements are predicated. Predication [7] is a method for translating control constructs into simple dataflow operations. It allows the removal of costly branches in conditional code. A predicated operation executes if the predicate (one of the inputs of the operation) is true.

The processing elements are connected together through a full crossbar (not depicted on Figure 2). This crossbar can connect the output of any processing element to the input of any other processing element. It is also possible to connect a processing element to a register from the main register file through the RFU ports. Parameters and results are passed in this manner. By limiting the interconnect to a less complex structure, savings in area and power consumption can be obtained at a cost in compiler complexity.

Each processing element has a small register file at its output (see Figure 2) that can be optionally bypassed, just like flip-flops in traditional FPGAs. By bypassing the register file, it is possible to connect the output of one processing element to the input of another processing element and thus perform spatial computation. Elements in a data flow chain are connected together through the crossbar. The processing element at the end of the chain is registered to combine temporal and spatial computation. By using spatial computation it is possible to reduce the critical path in the code at no extra cost than the register bypass and an increase in compiler complexity.

2.2 The reconfigurable decoder

The reconfigurable fabric provides enough computational power to execute the inner loops of multimedia applications. The processing elements of this fabric can be considered as extra functional units of a VLIW processor. There is, however, a major difference in how the elements are controlled. When the RFU needs to be used, the compiler inserts a special reconfigurable operation (ROP) in the main instruction word that specifies which configuration must be used. When the ROP is executed, the processor loads the adequate configuration into the RFU and executes it. In this manner, the RFU is only used when necessary. The ROP specifies the configuration that must be used, which must be loaded from a specified memory location. This translation from ROP to a much longer control word can be viewed in two manners: as a reconfigurable instruction set or as a programmable micro coded instruction set.

The compiler (see section 2.3) optimizes the code in such a manner that a loop might require more than one configuration. In this case, loading the configurations from the external memory every cycle of the loop would incur in a huge time and power consumption overhead. To solve this, instead of fetching the configuration from the external memory, the configuration is fetched from a local configuration memory. This configuration memory is a memory adapted for the execution of loops. It is very shallow, typically 32 words depth, enough to hold the configurations needed for the most interesting loops, and it is very wide, proportional to the number of processing elements in the RFU.

The configuration memory is now accessed by the ROP code in the main instruction word. The ROP specifies a configuration memory location, which in turn is going to specify the control lines of the processing elements. Figure 3 shows this scheme.

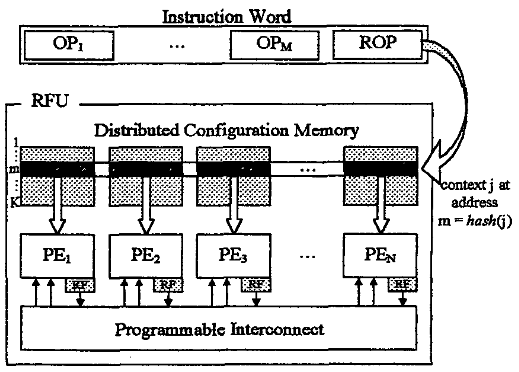
Instruction Word
OP_1
...
OP_M
ROP
RFU
Distributed Configuration Memory
1
m
K
...
context j at address $m = hash(j)$
PE_1
PE_2
PE_3
...
PE_N
RF
Programmable Interconnect

Figure 3 Control decoding in the RFU

The configuration memory is distributed all over the reconfigurable fabric. This allows the length of the control lines from the memory to the processing elements to be minimized. Thanks to these short lines and the shallowness of the configuration memory, we can achieve a great saving in power consumption in the control logic of the RFU.

With this scheme, the user needs to preload the configuration memory with data before it is actually used. An alternative mechanism that can be used is to use a configuration cache instead of a configuration memory. The ROP now specifies a configuration number (that can be much bigger than the depth of the configuration cache). If the configuration is in the configuration cache, it is accessed. If there is a cache miss, the configuration is loaded from a slower but bigger memory (placed in a lower level in the memory hierarchy). The time spent in this operation is the reconfiguration time.

By making the configuration cache very shallow we are able to reduce the power consumption. In some applications, the number of misses on this cache can be quite high, since not all the loops in an application will be able to fit in this memory. Adding a second level of configuration cache allows us to maintain a small reconfiguration time and at the same time reduced power consumption. The bigger and less power efficient second level cache is only accessed at the beginning of the loops.

2.3 Code generation

Code generation for a reconfigurable instruction set processor involves two tasks: generation of the different configurations for the reconfigurable array and generation of the fixed functional units of the processor. In the case of CRISP, with processing elements of similar complexity, common VLIW techniques have been used. On our research compiler (based on Trimaran [8]), code generation for loops is based on software pipelining [9]. In software pipelining, iterations are initiated at regular intervals and execute simultaneously but in different stages of the computation. This allows mapping the available parallelism onto the huge number of resources of CRISP.

The code generated for a loop will contain as many configurations as the iteration interval of the loop. It is therefore important to check that an iteration does not last more than the number of available configurations in the configuration cache. If this was not the case, the generated code would need constant reconfiguration.

Furthermore, software pipelining can also be modified to exploit the ability to perform spatial computation by chaining operations [9]. This allows a reduction of the critical path length of inner loops, with the corresponding decrease in execution time. The process of code generation with spatial computation requires a proper model of the timing delay of the processing elements and the interconnect, since the process is similar to the place and route stage in FPGAs.

3 Results

We compared the execution time of a VLIW processor with that of a CRISP processor. Both processors had the following functional units:

- 5 integer units
- 2 load/store units
- 1 branch unit

In addition, the CRISP processor had a reconfigurable functional unit with 32 processing elements, which could perform the same operations as the integer units.

Both processors had the following memory hierarchy:

- 16 KB L1 I-cache, 2 cycles access time.
- 16 KB L1 D-cache, 2 cycles access time.
- 2 MB L2 unified I and D cache, 8 cycles access time.
- 64 bit bus to external bus, pipelined with setup time 18 cycles and sustained transfer 2cycles/transfer

The CRISP processor also included a L1 configuration cache of 4Kbytes with 32 sets and a L2 configuration cache of 256KB.

Figure 4 compares the total execution time of some multimedia decoders. The benchmarks were taken from the Mediabench set of benchmarks [5]. Even though this are not real interactive applications, they have similar workloads. Figure 5 shows the execution improvement on the inner loops, which is where the optimization target was set. As we can see from the figures, for one of the benchmarks (ADPCM encode) no improvement in execution speed was observed. This was due to the lack of enough parallelism in this particular benchmark. On average, the improvement is around 15%. After closely examining the results, we found out that the limiting factor was the data memory bandwidth. With only two load store units, the RFU did not get enough data to exploit all the available parallelism. Rewriting the code to improve the way data is accessed will improve the execution speed. Since multimedia applications usually operate on contiguous data, widening the data memory transfers would also improve the execution speed.

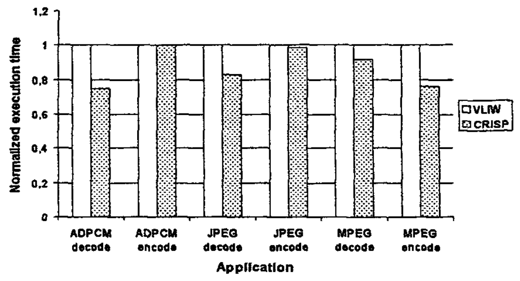

Figure 4 Overall execution improvement

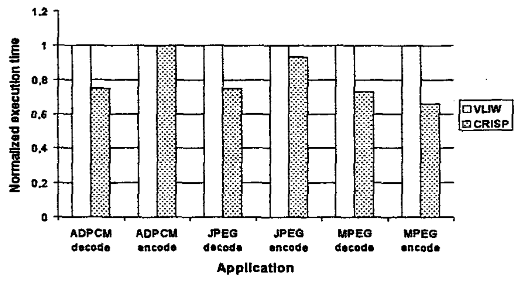

Figure 5 Loop execution improvement

In order to measure the effects of the reconfiguration time, we measured the applications with three different configuration memory hierarchies. The first one did not contain any configuration memory on chip. The second one contained a small L1 configuration cache (4KB with a depth of 32 words). The last contained the complete configuration memory hierarchy: level 1 and level 2 (256KB). Figure 6 compares the execution times of these three approaches.

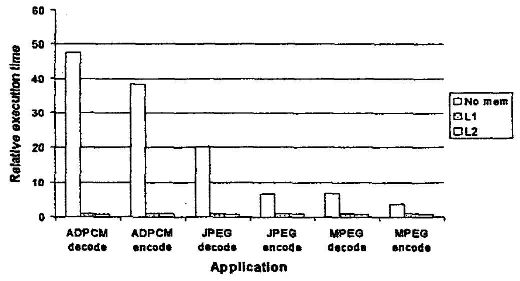

Figure 6 Effects of configuration memory hierarchy

From the figures, we see that at least a first level of configuration cache is needed if we want to obtain a fast execution (without this memory, the execution times were a lot worse than the standard VLIW processor seen before). Adding a second level of configuration cache does not drastically improve the performance (about 0.5% on average). The reason for this is twofold. First, the reconfiguration time is small due to the small size of the configuration stream (1024 bits). And secondly, once a loop is entered, the reconfiguration cost is quickly hidden by the execution time of the loop. With a more efficient compiler and with extra data memory bandwidth, it might be necessary to include the second level of cache. Also, it might be interesting to include it for power consumption reduction.

4 Conclusions

In this paper we have shown that a reconfigurable instruction set processor based on coarse-grained reconfigurable logic can reduce the execution time of interactive multimedia applications. Such a processor is optimized for two types of code:

- Highly parallel loops: these loops can profit from a big number of processing units and represent a big fraction of the total execution time. CRISP exploits the temporal locality in the loops to reduce the power consumption by using a low power configuration cache. The high number of processing units allows reducing the execution times of these loops.
- Sequential code: The amount of parallelism in this type of code does not require a big number of functional units. For this reason, during these pieces of code, the reconfigurable functional unit is not used at all. The processor behaves as a standard VLIW processor.

Loading the configurations into the configuration memory is a time consuming task. Using an L1 configuration cache almost eliminates the reconfiguration latency. Reconfiguration delays are small because coarse-grained reconfigurable logic has small configuration size. A level two of configuration cache does not drastically improve the performance. It can be used to reduce the off-chip memory traffic and reduce the power consumption.

By combining the large amount of processing elements in the reconfigurable logic, the characteristics of multimedia applications and the ability to quickly reconfigure the processor, it can be seen that coarse-grained reconfigurable processors are ideally suited for interactive multimedia applications. Future work will try to find out what the ideal type and numbers of processing elements that should be placed on the reconfigurable logic. The interconnect structure is also of great importance, as it will define important parameters such as speed and power consumption.

Acknowledgements

This work is supported in part by MESA from the MEDEA+ project.

References

[1] ISO/IEC 14496, "Information Technology – Coding of audio-visual objects", 1999.

[2] Johannes Kneip, Bernd Schmale, Henning Möller, "Applying and Implementing the MPEG-4 Multimedia Standard", IEEE Micro November/December 1999 (Vol. 19, No. 6)

[3] M.F. Jacome and G. de Veciana, "Design Challenges for New Application-Specific Processors" IEEE D&T Computers, Vol. 17, No. 2, April 2000, pp. 40-50.

[4] Trimedia Technologies Inc., "Trimedia 32 CPU Handbook", http://www.trimedia.com

[5] C. Lee, M. Potkonjak and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", Proceedings of the 37th IEEE/ACM Symposium on Microarchitecture, December 1997.

[6] Zhiyuan Li, Katherine Compton and Scott Hauck, "Configuration Caching Techniques for FPGA", IEEE Symposium on FPGAs for Custom Computing Machines, 2000.

[7] W.W. Hwu, "Introduction to Predicated Execution", IEEE Computer, January 1998, pp. 49-50.

[8] Trimaran, an Infrastructure for Research in Instruction-Level Parallelism, 1999. http://www.trimaran.org

[9] F. Barat, M. Jayapala, P. Op de Beeck and G. Deconinck, "Software Pipelining for Coarse-Grained Reconfigurable Instruction Set Processors", Proceeding of the Asian and South Pacific Design Automation Conference 2002, to be published.