Compiling ATR Probing Codes for Execution on FPGA Hardware *

W. Böhm, R. Beveridge, B. Draper, C. Ross, M. Chawathe and W. Najjar
Colorado State University and University of California Riverside

Abstract

This paper describes the implementation of an automatic target recognition (ATR) Probing algorithm on a reconfigurable system, using the SA-C programming language and optimizing compiler. The reconfigurable system is 800 times faster than a comparable Pentium running a C implementation of the same probing task. The reasons for this are analyzed.

1. SA-C and Probing

The goal of the Cameron project is to make application development on FPGAs easier by raising the abstraction level from hardware circuits to software algorithms. To this end, we have developed a high-level language, similar to C, called SA-C, and an optimizing compiler that maps SA-C programs directly onto FPGA configurations. More information on SA-C, its compiler, and other applications can be found at www.cs.colostate.edu/cameron.

SA-C is a single assignment language with data parallel loop constructs allowing for allowing easy detection of expression level and loop level parallelism. The SA-C compiler performs both conventional and FPGA-specific optimizations. ***Full or partial loop unrolling*** spreads iterations in code space rather than in time. ***Array value propagation*** replaces array references with constant indices with the array elements. The SA-C compiler not only performs standard common sub-espression elimination (CSE), but also ***temporal CSE***, replacing a computation in one loop iteration with a result computed in a previous iteration. This paper compares the timing of the probing algorithm written in SA-C and running on an AMS WildStar board with a C code version of the same algorithm running on a Pentium PC.

A probe is a pair of pixels and an associated true/false question. Typically, a probe returns true when the absolute value of the difference in pixel values exceeds a threshold, answering the question: "Does this pixel straddle a boundry?" A probeset depicts the silhouette of an object viewed from a particular viewpoint. When a probeset is placed in the correct image location over an object, most of its probes should return true. When a probing algorithm is applied to an image to recognize targets, each probeset is evaluated at every image position, and the location and identity of the highest scoring probeset is returned.

The exhaustive application of probesets for all possible objects, viewing angles, and image positions is a computation that it is ripe for optimization. It is therefore both an algorithm of considerable practical interest and a powerful demonstration of what can be done using FPGAs and the SA-C compiler. Here is a pseudo code version of the Probing algorithm.

```
for each window in image {
 best_score, probe_set_index =
  for all probe_sets {
   hit_count =
    for each probe in probe_set
    return(sum(threshold(probe)))
   score = hit_count/probe_set_size
  } return(max(score),probe_set_index)
} return(array(best_score),array(probe_set_index))
```

The two inner loops computing the scores and probeset indices can be fully unrolled, because the probesets are statically known. This turns the code into a singly nested loop driven by one window generator. The loop body becomes an expression consisting of threshold operators computing hits, sum trees adding the hits for each probeset, division operators computing the scores for each probeset, and max trees selecting the winner. This giant expression allows for standard and temporal CSE. The computation of the score of a probeset in a window requires the hit count to be divided by the probeset size. Floating point division is replaced by a table lookup that maps hit counts and probeset sizes onto ranks. Scores below a threshold (e.g. 80%) can be given rank zero. This reduces the number of bits in the rank, and therefore the size of the lookup table.

The test suite for the probing application consists of three vehicles (an M60 tank, an M113 armored per-

*This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

	Unoptimized			Optimized		
	Pbs	Adds	Win.	Pbs	Adds	Win.
m60	2832	2751	12x34	151	1413	12x4
m113	2315	2234	11x26	106	967	11x4
m901	2426	2345	13x25	143	1196	13x4
total	7573	7330	13x34	400	3576	13x4

Table 1. DFG level statistics: Probes , Additions, Window sizes before and after optimization

sonnel carrier, and an M901 armored personnel carrier with missile launcher), each represented by 81 probe sets (27 aspect angles times three depression angles), totalling 7573 probes in windows of sizes up to 13x34. The input is a 512x1024 LADAR image of 12 bit pixels. The WildStar has three XCV2000E Virtex FPGAs, capable of operating at frequencies from 25 MHz to 180 MHz. It communicates over the PCI bus with the host computer at 33 MHz. In our system, the board is housed in a 266-MHz Pentium-based PC.

We will compare the performance of SA-C running on the WildStar to the performance of C running on an 800 MHz Pentium III. The XCV2000E and 800 MHz PIII are of similar age, and were both fabricated at .18 microns. In both cases, execution times do not include the time required to read the image from the disk or host into local memory, but do include I/O time between the processor and local memory. The SA-C code is partitioned in the most straightforward way: each vehicle is mapped onto one FPGA. Each FPGA scans the input image and produces an image of winning scores and probeset indices for its particular vehicle. The host gathers the resulting data and creates an 8 bit version of the input image with the probeset of the highest scoring winner superimposed over it.

Table 1 provides dataflow graph level statistics for the test suite. It shows the number of probes, the number of additions in the sum trees, and the window size, before and after optimization. This shows that optimization reduces the number of probes about nineteen fold and the number of additions about two fold. About 50% of these additions are one bit additions. The number of columns in the window is compacted from the width of the largest probe set (34) to the horizontal width of the widest probe (4).

Program execution time on the Wildstar is 81 milliseconds. Executing the equivalent C code on the Pentium, using the Microsoft VC++ compiler optimized for speed, takes 65 seconds. Hence the Wildstar is about 800 times faster than the Pentium. These times can be explained as follows.

For the configuration generated by the SA-C compiler for the probing algorithm, the FPGAs run at 41.1 MHz. The program is completely memory IO bound: every clock cycle each FPGA reads one 32 bit word, containing two 12 bit pixels. As there are $(512-13+1)*(1024)$ 13 pixel columns to be read (see table 1), the FPGAs perform $(512-13+1)*(1024)*(13/2) = 3,328,000$ reads. At 41.1 MHz this takes 80.8 milliseconds.

The Pentium performs $(512-13+1)*(1024-34+1)$ window accesses. Each of these window accesses involves 7573 threshold operations. Hence the inner loop that performs one threshold operation is executed $(512-13+1)*(1024-34+1)*7573 = 3,752,421,500$ times. Using the optimizing VC++ compiler, the inner loop takes 16 instructions (much better than the 22 instructions gcc -O6 produces). The total number of instructions executed in the inner loop is therefore $3,752,421,500*16 = 60,038,744,000$. If one instruction were executed per cycle,this would bring the execution time to about 75 seconds. As the execution time of the whole program is 65 seconds, the Pentium (a super scalar architecture) is actually executing more than one instruction per cycle!

The factors contributing to the speed difference between the Wildstar and the Pentium can be broken down as follows. (1) **Analysis and optimization:** the compiler has reduced the number of probes nineteen fold. (2) **Coarse grain parallelism:** the Wildstar executes three processes, each process corresponding to a vehicle, in parallel without any interference. (3) **Massive fine grain parallelism:** each FPGA performs all its threshold operations, hit summations, table lookups, and comparisons in parallel, whereas the Pentium performs slightly more than one instruction per cycle. (4) **Clock frequency:** the Pentium runs at a 20 times higher clock rate than the FPGAs.

It can be argued that an optimizing and parallelizing compiler for a parallel von Neumann machine could achieve the same improvements in terms of the first two factors: analysis and optimization, and coarse grain parallelism. However, the largest factor, the fine grain parallelism, is a defining FPGA characteristic.

2 Conclusion

In this paper we have studied Probing, its implementation in SA-C, and its FPGA performance compared to a C version running on a Pentium. We have analysed the factors comprising the difference in FPGA and Pentium performance. We have shown that the FPGAs show a considerable speedup as compared to the Pentium, even when programmed in a high level language.