

A Transformational Approach to High Performance Embedded Computing

Wim Böhm
Colorado State University
Fort Collins, CO

Jeffrey Hammes
SRC Computers, Inc.
Colorado Springs, CO

1. Introduction

This paper describes a transformational, high level language approach to High Performance Embedded Computing on the SRC-6 machine and its MAPTM reconfigurable hardware. A program is initially written in pure C and compiled by the MAP C Compiler. Then, using feedback from the MAP C compiler, the program is successively transformed manually to achieve better performance. These transformations avoid certain inefficiencies, such as re-reading values from memory, loop slowdown caused by loop carried dependencies, and underutilizing memory bandwidth. We discuss the transformations in the context of the Wavelet Versatility Benchmark and the Gauss-Seidel iterative linear equation solver.

FPGAs use a large number of pins to connect to memories. They do not have caches, but they have on-chip block RAM, allowing the programmer to decide what data stays on chip. Also, fine grain operation level parallelism combined with pipelining makes it possible for FPGAs to execute an inner loop body in one clock cycle. These characteristics provide a simple, deterministic performance model, allowing the programmer to work towards a well defined goal: store hot data structures on chip either in block RAM or in registers, create inner loop bodies that execute in one clock cycle and use the full memory bandwidth of the machine by loop unrolling.

2 The SRC-6 and MAP Compiler

The SRC-6 machine contains a pair of dual-processor Pentium IV boards, running Linux, and two SRC-developed FPGA-based reconfigurable processors called MAPs. Each MAP contains two Xilinx Virtex-*II*TM FPGAs, six banks of dual-ported SRAM on-board-memory (OBM) totaling 24 Mbytes, and a control FPGA containing a DMA engine that manages memory transfers into and out of the OBM. DMAs can take place concurrently with executing FPGA code. Both user FPGAs have access to the OBM banks, though only one can access a given memory at a time. Each of the six memory banks contains 512K 64-bit words (4 Mbytes). The user FPGAs are clocked at a fixed frequency

of 100 MHz. Each OBM bank can handle one write or read from a user FPGA in each clock.

Using a simple directive the user allocates off chip arrays onto OBM banks. The MAP Compiler allocates local arrays to the FPGA's block RAM and local scalar variables in registers. The MAP compiler front end produces a control flow graph (CFG) of basic blocks and directed control flow edges between the blocks. Next the MAP Compiler translates each block into its own dataflow graph (DFG) that exposes instruction-level parallelism. It then merges the DFG fragments that compose an innermost loop into a single pipelined code block that includes a driver module for firing loop iterations. The driver will fire one loop iteration on each clock, unless data dependencies or multiple accesses to a bank force it to run slower. The DFGs for the code blocks are then mapped to Verilog, using straightforward instantiations of pre-defined macros. The Verilog is synthesized and place-and-routed using commercial software.

The MAP Compiler allows a user to create "user-macros" in Verilog. Their semantics differs from functions in C in that they can retain state between calls. This allows for program transformations providing code optimization beyond straight forward C compilation.

3 Wavelet Versatility Benchmark

The Wavelet Versatility Benchmark is part of a benchmark suite for evaluating configurable computing systems. This suite was produced by Honeywell as part of the DARPA/ITO ACS (Adaptive Computing Systems) program [2]. The Wavelet Benchmark consists of four phases: Wavelet Transform, Quantization, Run-Length Encoding and Entropy Encoding.

The Wavelet Transforms perform a 5x5 convolution of an input image stepping by 2 in both horizontal and vertical directions, reading from one OBM and writing to four. In the initial pure C implementation of this convolution, values are re-read on average 2.5 times. This can be avoided by using system-macros that implement a delay queue mechanism [1]. In a next transformation, four pixels are packed in one word, and two horizontally adjacent convolutions are

performed in parallel.

The initial implementation of the Quantization, Run-Length Encoding and Entropy Encoding phases read/write pixels (packed after the Wavelet phase was optimized) from/to OBMs. To avoid unnecessary memory traffic and to fully benefit from pipelining, the three phases are loop fused. The MAP compiler indicates loop slowdown, caused by loop carried data dependencies in Run-Length Encoding and Entropy Encoding in the form of (min) reductions and (summing and shifting) accumulations. These can be avoided by using stateful reduction and accumulator macros.

A final transformation distributes the program in a coarse grain parallel fashion over the two user FPGAs. Its execution on the MAP produces bit-identical results to the Honeywell reference code and achieves a speedup of 38 when compared to the reference code executed on a 2.8 GHz Pentium IV.

4 Gauss Seidel

Gauss Seidel is an iterative linear system solver of diagonally dominant systems $Ax = b$. The inner loop of the code recomputes $x[i]$ using a vector inproduct:

```
for(i=0;i<n;i++) {  
  s = 0.0;  
  for(j=0;j<n;j++)  
    if (j != i) s += A[i*COL+j] * x[j];  
  x[i] = b[i] - s;  
}
```

All A, b and x values are in single precision floating point. In a first pure C implementation the x vector is allocated in block RAM, while A and b are allocated in one OBM. When this code is compiled, the MAP compiler indicates a loop slowdown because s is both read and written in the inner loop. This can be avoided by using a floating point accumulator macro. This accumulator needs to be able to accept a new input every clock cycle. It will have a larger than one latency, as a floating point add takes 10 cycles on the MAP. This implies that the accumulator will have to be parallelized internally. At the time of writing this abstract, this and other floating point macros have not been integrated in the MAP compiler yet.

In a next program transformation, k values $x[i], x[i + n/k], \dots, x[i + (k - 1)n/k]$ are updated in the inner j loop in parallel. Because we can row block partition A and b over six OBMs, a good value for k is 6. The j loop reads one value from each OBM and performs 6 multiplies and 6 adds.

Because we are using single precision floating point we can pack two adjacent values in one word, thereby doubling the amount of computation per communication. The inner loop now performs 12 multiplies and 12 adds in each iteration. However, the MAP compiler now reports a loop

slowdown: because the inner loop is unrolled, two consecutive x values are read from block RAM. This can be avoided by stripe partitioning the odd and even x elements over two block RAMs.

The code currently runs in debug mode on a host machine. The MAP compiler backend for floating point operations will soon become available, and we will assess the hardware performance of the Gauss Seidel codes. If the most parallel version of the code with the 12 single clock floating point accumulators can be placed and routed on the FPGA, its inner loop will execute 24 floating point operations per clock cycle. At 100 Mhz, this will represent 2.4 GFlops.

5 Conclusions and Future Work

In this paper we have argued that high performance embedded computing can be achieved on the SRC-6 machine and its MAPTM reconfigurable hardware by starting from a pure C code and transforming this code stepwise using system or user macros. For the codes we have studied, the transformations 1) employ delay queues to avoid re-reading from OBMs, 2) pack data items in words and 3) unroll loops to increase bandwidth, 4) use accumulator macros to avoid loop slowdown caused by loop carried dependencies, 5) fuse loops to avoid memory traffic, 6) partition arrays to avoid multiple memory accesses in the same loop body, and 7) perform coarse grain task parallelization to shorten the critical path of the complete application. In future work we will implement sparse solvers and the NAS Parallel Benchmark suite.

References

- [1] J. Hammes. Methodology for pipelining and fusing stenciled loops. Technical Report SWP-009-00, SRC Computers, Inc., November 2003.
- [2] R. Kohler. Benchmark specification document – versatility stressmark. Technical Report CDRL A001, Rome Laboratory, November 1997. Submitted by Honeywell, Inc.