Using EDIF for Software Generation

MJ van der Westhuizen Student Member IEEE, RG Harley Fellow IEEE, DC Levy* Member IEEE, DR Woodward Student Member IEEE

Department of Electrical Engineering, University of Natal, South Africa
e-mail: mwesth@elaine.ee.un.ac.za

Department of Electrical Engineering, University of Sydney, Australia

With the advent of the FPGA and parallel microprocessors the need for practical codesign methods is becoming increasingly important. This paper proposes that codesign can be approached from existing hardware development tools. The paper also reports on the development of a software tool which uses EDIF to generate parallel, real-time C code. The view taken is that the problematic issues of codesign are the same as those for optimising general parallel processing systems and that scheduling theory is the foundation of both codesign and parallel design environments.

1: Addressing the problem of codesign from a parallel processing perspective

Embedded real-time designs are often more challenging due to the fact that cost-sensitive high computing performance in small packages is usually required. The requirement for bringing products to market quickly is compounded by the mission-critical aspects, either due to the high costs of recalling products or to the catastrophic consequences which design flaws might suffer. Two relatively novel technologies which are presented to the real-time system designer are parallel processing and programmable hardware systems.

Parallel processor developments include the Inmos family of Transputers, the latest range of Texas Instrument DSP processors and low cost processors communicating via CAN networking. Programmable hardware has the FPGA and parallel processors communicating via dynamic crossbar switches as example. However, the design variables in such systems pose combinatorial optimisation problems which are difficult to address manually. One typical design decision is to decide on the portion of a particular block diagram which could beneficially be placed on a co-processor or dedicated hardware. With dozens of possible interrupts, hundreds of states and different function blocks, the number of decisions rises exponentially. The work done at the Department of Electrical Engineering at the University of Natal addressed the issue of optimally scheduling a block diagram representing control algorithms on a number of parallel processors (Transputers).

The SPLAT (Serial-Parallel Load Allocation and Timing) system accepts a block diagram in netlist format, optimises it and produces both parallel ANSI C and Gantt charts. Although a custom graphical interface was at first experimented with, it was found that schematic diagram drawing packages such as OrCad can be used by simply parsing the EDIF netlist. Whereas CASE tools are still relatively expensive, hardware schematic entering is a mature industry which offers proven products at reasonable prices due to a large and established user base. By using OrCAD, we were able to concentrate on the important scheduling aspects and thus produce a stable, high performance parallel software development environment in a relatively short time.

Of particular interest with parallel processing is coping with interprocessors communication delays (IPC) and heterogeneous networks (some transputers are faster than others). The difference in execution speed also depends on the nature of the function since a mix of processors with and without floating point coprocessors is possible.

We believe that the problem of designing systems with coprocessors such as custom ASICs can be viewed as an instance of the general parallel processing problem and could therefore also be addressed with our parallelising tool. The SPLAT system accommodates heterogeneous networks by using a weighting table of task durations on the different processors. Tasks which cannot be executed on a particular processor are given an infinite duration on that processor.

By viewing ASICs as separate processors which are efficient on certain tasks and inefficient (zero speed for tasks it cannot execute) on other, the problem of deciding for a particular cost or speed objective on how a block diagram of tasks is to be proportioned in hardware and software is made to be the same as the heterogeneous parallel processing scheduling problem. The most commonly employed scheduling heuristics at present are global optimisation methods which readily accommodate multiple-goal optimisation objectives. Factors such as IPC delays and process synchronisation which cannot be neglected in parallel processing systems can be just as important with codesigns.

Since the SPLAT system is used to generate real-time software from CAD software traditionally used for designing hardware, the overall design is therefore more "hardware friendly" and unlikely to contain software constructs and design styles which cannot be implemented in hardware.
between differently placed functions are left to be significant. The delays for memory communications are included in the total delay of the particular function.

3: Specific results: Scheduling and Neural Network FOC algorithms

The Gantt chart and C code for the block diagram of Fig.2 is illustrated in Fig.4. For the relatively simple block diagram of Fig.2 the SPLAT systems does not produce significant speedup and an program optimised manually proved to be slightly faster. In order to better evaluate the scheduler, more complex algorithms which exhibit more parallelism are needed. Our research group is currently investigating the use of artificial neural networks (ANN) for the control of AC motors and this requires sub-millisecond execution of algorithms which include more than one ANN. The ANNs are in the order of eight inputs, twelve middle layer neurons and two output neurons with back-propagation (8x12x2 ANN).

As a starting point, an attempt has been made to schedule a single 8x12x2 ANN with backpropagation (Fig.3). Granularity is at the neuron-level. The repetitive aspects of a neural net would typically be programmed in a high-level language by using "FOR" constructs. However, using "FOR" loops to program spatially parallel constructs destroys the parallelism. Programming complex algorithms in a Data Flow (non-recursive functional) language would be time-consuming and error-prone. The partial block diagram shown in Fig.3 illustrates how easily complex algorithms can be programmed in a Data Flow manner with the aid of CAD software. When more advanced CAD feature such as hierarchical design and busses are used for programming vectors, sparse matrix implementations are relatively easily obtained by peephole optimisation (removal of zero additions and unit multiplications). With the hierarchical editing feature the design can easily be changed from a granularity at neuron-level to a fine-grain level.

4: Conclusions

This paper reported on the development of the SPLAT software tool which produces optimised parallel C code from standard electronic CAD software. The tool employs graph theoretic preprocessing modules as well as heuristics such as simulated annealing with a static scheduling model. The SPLAT software operates from standard to standard because input is in the form of an EIA standard netlist format and it produces ANSI C. Apart from ensuring that the design does not generate code which is not implementable in hardware, the use of schematic design methods also permits the exploitation of parallelism and real-time optimisation which is difficult in traditional computer languages such as C.
Fig.2 Example of real-time software designed with standard CAD software.

References


Fig.3 Partial block diagram of a neural network controller.
read left to right -> time
1
0
Scale: 118 increments of 16 time units

<table>
<thead>
<tr>
<th>Proc</th>
<th>PO 7800</th>
<th>P1 7800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20/256</td>
<td>20/256</td>
</tr>
</tbody>
</table>

| Links | b?** | 0*** |

| Mem. | 2165 | 370 |
| Comp. | 350 | 350 |
| Total | 1387 | 747 |
| Ratio | 68% | 127% |
| Busy | 93% | 67% |

< except from boot link on L0P0, other links are free

LEGEND:

| XXX | func XXX starting at time x
| XXX | hard-placed activity
| dlPyrzy comm. on link y, to proc. y
| dlPyrzy comm. used to update buffers
| state same as previous

< end time (in 50 μsec cycles)
(2521 / 20 = 126 μsec)
< chosen link configuration
< memory used

/*
loop1slice(7) = loop1slice7;
while(1)do_loop1slice = ++do_loop1slice8;
0-404 A2DC(A2DC_17);
/A2DDRIVER*/
Fork..25..IN = BUFFER..30.OUT1;
A2DQ.6.6 = A2DC_17.CH1;
A2DQ.6.6 = A2DC_17.CH2;
A2DQ.6.THE = FORK..25.OUT1;
826 A2DQ(&A2DQ.6);
SUB.10.POS = BUFFER..34.OUT1;
SUB.10.NEG = A2DQ.6.6;
packet_0_arr((packet_0_dex=0++) = A2DQ.6.6;
ChanOut(chan.O1, packet_0_arr, packet_0.size);
SUB($SUB..10);
PI_2.IN1 = SUB.10.OUT;
PI(GPI_2);
packet_1_arr((packet_1_dex=0++) = PI_2.OUT;
packet_1_arr((packet_1_dex++) = PI_2.OUT;
BUFFER..30.OUT2 = BUFFER..31.IN2;
BUFFER..29.IN1 = A2DC_17.CH4;
BUFFER..28.OUT2 = BUFFER..29.IN2;
packet_2_arr((packet_2_dex=0++) = BUFFER..33.OUT2;
ChanOut(chan.O1, packet_2_arr, packet_2.size);
BUFFER..27.IN1 = A2DC_17.CH3;
BUFFER..26.OUT2 = BUFFER..27.IN2;
BUFFER..34.OUT2 = BUFFER..35.IN2; /*
loop1slice(7) = loop1slice7();
void loop1slice0(){SUB(&$SUB..8);)
void loop1slice1(){INTG(1INTG..5);}

Fig.4 Gantt chart and C code for Fig.2