A State Transformation based Partitioning Technique using Dataflow Extraction for Software Binaries

M. Sangeetha[1], Dr. RajaPaul Perinbam[2] , M.Kumaran[3]

[1]M. Sangeetha , Research Scholar, Department of Electronics and Communication Engineering, Anna University, Chennai-600025, INDIA.
[2]Dr.Raja Paul Perinbam is Professor at Department of Electronics and Communication Engineering., Anna University, Chennai-600025, INDIA.
[3]M.Kumaran , Assistant Professor, Department of Computer Science & Engineering, Jaya Engineering College, Chennai, INDIA.

Abstract
Hardware-Software Partitioning and decompilation is a key issue in the Codesign of embedded systems. Partitioning in binary level helps in independent usage of software languages for the compilers. In this paper, the critical kernel of the software binary is relocated to the hardware and this is identified using instruction level profiling. The partitioned software binary is represented with initial and final state by a set of register value pairs. In the software binary the initial state to final state transformation is derived by equating the final state in terms of algebraic place holders, and then synthesized into hardware. A generalized decompiler is also designed to generate equivalent HDL for software binary block. The proposed method is applied to standard benchmarks and show significant speedup with lesser hardware resources. A source level partitioning is carried out for the scheduled elliptic wave filter and buffer size is estimated in binary and source level approach.

Key words:
Dataflow, Hardware/Software Codesign, Embedded systems, Partitioning, Decompilation

1. Introduction

Traditional design technologies and flows required that hardware and software be specified and designed separately for an embedded system. Once the behavior of a system is fixed, the specification needs to be coded in different languages: For example HDLs (Hardware Description Languages) for the hardware and typically C/C++ for the software. The software or hardware partition can be done by profiling the specification or legacy software code. However, the partitioning is often pre-determined. Defining a system partition apriori could

i. Lead to sub-optimal design
ii. Create lack of a unified hardware-software representation
iii. Lead to complexities in the verification of the entire system performance
iv. Result in incompatibilities across the hardware/software boundary
v. Time-consuming due to rewriting an entire code in HDL

To overcome the above limitations, Hardware/Software codesign has emerged as a successful approach.

Hardware/Software codesign [12][13] attempts to integrate the hardware and software paths by envisioning a common platform, and increases the possibility of interaction between the hardware and software development. The hardware-software codesign gives an optimized design in terms of performance. The investigation in this work is directed towards optimal use of binary level partitioning and source level partitioning to achieve maximal performance. A technique that leverages a systematic transformation of the basic blocks of software binaries into dataflow descriptions for implementation of the partitioned software in hardware, by equating the final state attained due to execution of each basic block in the partition in terms of algebraic placeholders for the initial state in the system, is described. Control nodes are used for representing branching and loops that lead to different basic blocks based on conditional expressions. Initially the part of the software binary to be transformed into hardware is identified using instruction level profiling. The tool flow adopted is as shown in figure1.

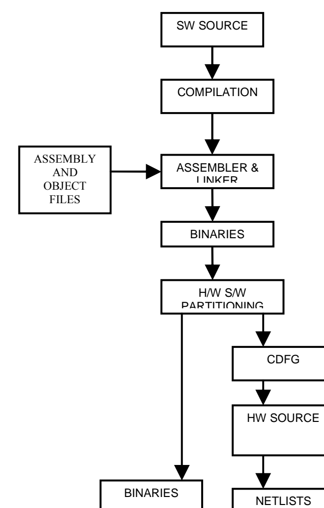


Figure 1 HW/SW Tool Flow

Manuscript received February 5, 2009
Manuscript revised February 20, 2009

2. Previous work

An optimization method that selectively breaks up some of the specification operations, mitigating the RAW dependences among them, in order to speed up the execution of data-intensive behavioral specifications [15]. Register allocation is an optimization technique which is implemented using Linear-Scan (left-edge) algorithm and is performed after scheduling to reduce the number of registers [16] used in the proposed decompiler. Reducing registers in circuit designs generally leads to smaller design size [17] and provide creation of register transfer list for software binaries that is partitioned to hardware. However, they are unable to schedule operations in different blocks in the same clock cycle.

Merging the consecutive sets of blocks into one large block, allows the compiler to possibly schedule multiple instructions in parallel [18]. In the proposed work merging is carried out for forward branching and set of basic blocks. These transformations and optimizations has the advantage, that they reduce the design size and also increase parallelism in the design.

Whenever the constant values for multiplication operation were powers of two (AddersSR = 0), both sets of hardware (for multiplier and multiplicand) had identical performance. This is because, strength promotion converted the strength-reduced code back into multiplication operations. This is again strength-reduced by the synthesis tool and there would be no benefit from using multipliers [19].

An elliptic wave filter is implemented for different scheduling algorithms like ASAP, ALAP, FDS, LS, FDLS and suggested MOGS [20]. The MOGS algorithm has less cost function when compared to other scheduling algorithms. The proposed scheduling and allocation algorithm proves to have one control step less which in turn reduces the execution time and cost function. Since the other algorithms are optimized for scheduling only (they minimize the number of functional units only) but proposed work includes more functional units but total number of functional units does not influence the cost function.

The elliptic filter is scheduled for minimum of 17 control step by [21], deadline ranges from 17-34 in [22], 18 control step for ALAP, ASAP, FDS, LS and MOGS except FDLS which takes 19 control step. An effort is not made to reduce the critical path length which is suggested in scheduling and allocation algorithm to shorten the control step to 16 without modifying the functionality. Buffer size for hardware/software partitioning is also calculated to obtain communication cost.

3. Hardware/Software Codesign

The Codesign starts with a behavioral description specifying the functionality of the system using a high-level language (C language). The high level language is transformed into binary level description using Small Device C Compiler (SDCC [8]). The target architecture chosen in this work is PIC 18F452. The executable code or binary compiled for the processor is profiled by passing test vectors using GPSIM [6]. The most frequently executed code from the profiled list is partitioned for execution by hardware units. This provides a basis for extremely accurate hardware/software partitioning at a very fine grain level. An optimized decompiler is designed using MATLAB to transform the partitioned binary into Hardware Description Language (HDL) as register transfers for each instruction [17]. The speedup, hardware utilization and runtime of the instructions partitioned are obtained from the benchmarks.

The Scheduling and Allocation Algorithm is proposed with an objective to reduce the critical path length. The obtained results show better optimization for resource constrained and time constrained system as compared to the results tabulated in [20]. The scheduled graph is partitioned into hardware and software modules and buffer size is calculated.

4. Binary level partitioning

The behavioral modeling is transformed into software binaries and need to be partitioned for hardware and software. The partitioned software binary[1] for hardware should be decompiled to hardware description language. So a partitioner and decompiler are needed to be designed in binary level approach. The decompiler design should be an optimized one, to obtain good result in binary level approach. The buffer size needs to be estimated in binary and source level approach to obtain the communication cost between hardware and software.

4.1 Decompilation

The hardware partition along with the software glue code has to precisely realize the effect of the software instructions that are going to be replaced by them, in a partitioned system. A decompilation technique for software binaries partitioned for hardware is discussed in [2][14][23]. This hardware partition must maximize performance/ minimize delay and buffer requirements as the cost function. Translation of binaries to FPGA is discussed in [7][24]. In this work, for each instruction processed in the partitioning algorithm, a jump is checked and lists of all such jump instructions are created. The addition in terms of time complexity is $O(j*p)$ where j represents the number of jumps encountered and p the number of partitions gathered. Similarly, space complexity also is of order $O(j)$.

4.1.1. Dataflow Extraction

The partitioned instruction is transformed in control data flow graph which is then converted into HDL source. We can separate the initial state and the final state needed for this realization by decoding the instruction sequence. The algebraic variables are assigned to all the registers in instruction sequence. For a non-control-flow instruction, we would have one or more source operands, one or more destination operands and an operation that maps source operands to destination operands. The destination operands have been changed by the operation in terms of the source operands resulting in algebraic expression. The original value of the destination operands in the initial-state set varies with the final-state set values according to the algebraic expression. Then, the next instruction's initial-state set is the final-state set of the previous instruction.

Initial-state set can suffice to contain only the register-value pairs whose values are used as source operands in the instruction. Final-state set can have only those register value pairs whose values are modified from the original-state set. The registers that are neither used as source operand or destination operand in register value pairs from the initial-state and final-state sets are omitted. Given an already populated initial-state and final-state set, the processing of an instruction requires scanning the final-state set for the value of the source operand. If the lookup is successful, the value is propagated to the expression for the destination operand. If the lookup fails, the same is scanned in the initial-state set and propagated. If this lookup also fails, it means that this register is added either a source or destination. So we have to add the register to the initial-state set and assign a new algebraic value. This value is propagated to the destination register's value expression. At the start of processing of a partition block (no instruction would have yet been processed) .The initial-state and final-state sets would be empty, and get populated as and when more instructions are processed. In effect, the initial-state and final-state sets are incrementally updated to reflect the effect of execution upto the instruction in question from the beginning of the block. This kind of mapping from initial state to final state is straightforward for a basic block. Suppose if we have the code snippet, shown in figure 2, given in pseudocode form, the control flow can be visualized as in figure 3. The application of the dataflow extraction algorithm results in the sets of basic blocks and control nodes as shown in figure 4. However, when loops are involved, this mapping method quickly degenerates lead to complex and computationally expensive expressions.

```
add reg-b to reg-a                  //basic block 1
//some more processing

if(reg-a <> 0) goto block(3)        //control node 1

add reg-c to reg-b                  //basic block 2
//some more processing
goto block(4)

Add reg-c to reg-a                  //basic block 3
//some more processing

Add reg-d to reg-b                  //basic block 4
//some more processing
```

Figure 2 Example code snippet

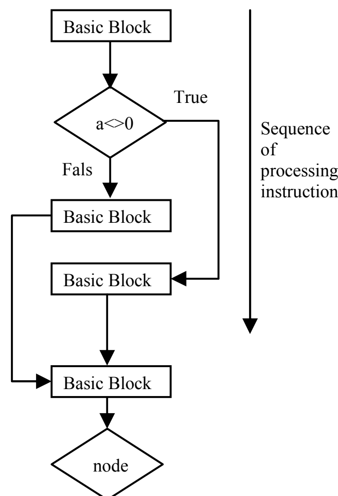


Figure 3 Control flow and block Visualization in Memory

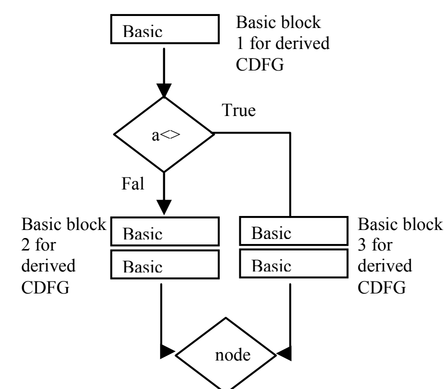


Figure 4 The CDFG generated for the snippet

4.1.2 Optimization of CDFG

Optimizations steps need to be applied for the CDFG [10] once it is ready, and these are listed as follows:

1. For a basic block, if the registers in the initial-state set are also listed in the final-state set and the value is the same in both (can happen if registers are shadowed and used as scratchpads temporarily), the entry can be removed from the final-state list.

2. After processing by step 1, if the values in the initial-state set are not used in the final-state set, these can be removed from the initial-state set as well.

3. For the section of the graph that has only forward jumps and no jumps from the remaining part of the graph into the middle of the section in question, the initial-state sets and final-state sets of all the basic blocks in that section can be clubbed together, and the intermediate control nodes can be removed.

4.1.3 Merge operation

The destination operand element in the clubbed final-state set can have multiple values, each corresponding to a basic block, that are tagged with a conditional expression corresponding to the control nodes that lead to the execution of the basic block. The conditional expressions for these values of an operand are mutually exclusive, meaning that only one of the values is selected at any instant of execution in the CDFG, depending on the evaluation of the expressions. As an example of demonstrating, figure 5 shows the result of merging a basic block (B1), a control node (C1) following the basic block, and the two basic blocks (B2,B3) that represent the two paths that a control node can lead to. This is the basic structure for a merge operation. All constructs having forward jumps only would be contained by recursively applying this model structure. The detailed steps for executing the Merge operation of the Basic Blocks are given in Appendix 1.

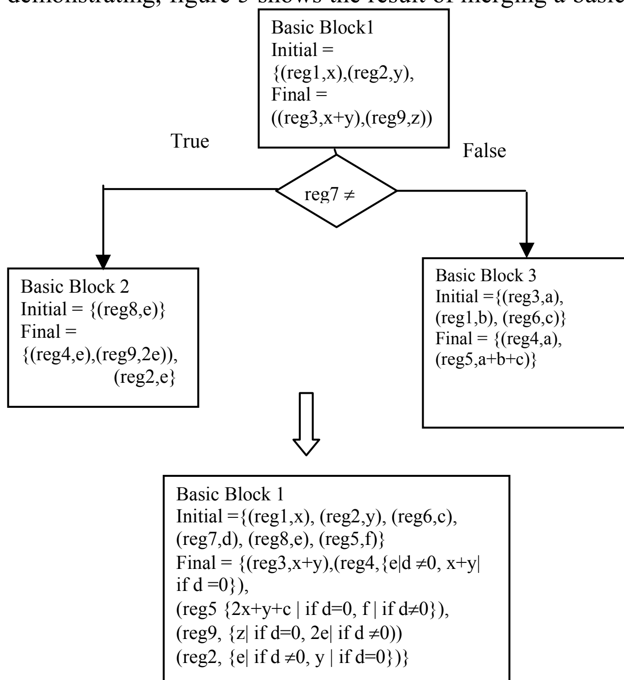


Figure 5 Merge operation for Basic Blocks

4.2 Complexity Analysis for Dataflow Extraction

4.2.1 Time complexity

The time complexity of the algorithm can be computed assuming each instruction has a single source operand and a single destination operand. If the number of instructions in a partition is 'n', and the number of instructions determining control-flow (equivalent to number of control nodes that would be created) is 'c', the average length of a block is n/c. The number of blocks that could be created (accounting for possible duplication of blocks) is 2c. If each operand in each instruction is assumed distinct as a worst case, the number of comparisons to create initial and final state sets for a block of length n/c would be O((n/c)2). The total number of comparisons for all 2c blocks would be $O(n^2/c)$. Comparisons to determine the uniqueness of control nodes is of $O(c^2)$.

4.2.2 Space Complexity

Similarly the determination of the uniqueness of basic block nodes is $O((2c)^2) = O(c^2)$. Hence, in summary, the time complexity of the algorithm is $O(n^2/c) + O(c^2) + O(c^2)$ = $O(n2/c) + O(c^2)$. Similarly, the space complexity can be calculated. For a basic block, the initial and final state sets can have elements of O(2n/c), and for 2c basic blocks, the space complexity is O(2n/c*c) = O(n). Similarly space complexity for control nodes is O(c). Similarly for the stack, the complexity is determined by the number of basic blocks 2c, hence O(c). Hence the total space complexity is O(n) + O(c).

5. Source level partitioning

The behavioral description should be transformed into intermediate format which has to be partitioned into hardware and software. The control steps and cost function need to be reduced to increase the system performance. To obtain the communication cost, the buffer size needs to be calculated for the software and partitioned hardware of the intermediate format. An effort is made to prove binary level partitioning as same as source level partitioning with respect to buffer size.

5.1 Scheduling and Allocation Algorithm

The data flow graph obtained from the input description is scheduled using As Soon As Possible(ASAP) scheduling and As Late As Possible(ALAP) scheduling. In ASAP scheduling the earliest time at which an operation can be scheduled is computed and ALAP can also be computed by adapting the longest path algorithm to work from the outputs backwards. Combining the information obtained in both ways of scheduling[4][9][11] algorithm gives rise to more powerful heuristics called mobility based scheduling according to the available functional units as shown in figure 6. ASAP scheduling time of node v_i is denoted by $\sigma_S(v_i)$ and the ALAP time by $\sigma_L(v_i)$, the interval $[\sigma_S(v_i), \sigma_L(v_i)]$ contains all possible time instants at which vi can be scheduled. This interval is called the time frame

```
determine by computing σS and σl
k ←0;
While ("there are unscheduled operations")
{
v  "one of the nodes with lowest mobility";
"Schedule at some time that optimizes the current resource utilization";
"Determine by updating the scheduling ranges of the unscheduled nodes";
k ← k+1;
}
```

Figure 6 Mobility based scheduling algorithm

or the scheduling range of operation. The length of the interval, i.e. $\sigma_L(v_i)$ - $\sigma_S(v_i)$, is called the operation's mobility. The scheduling algorithm proposed takes care of resource-constrained synthesis. With the given allocation of hardware, find a scheduling and assignment such that the total computation is completed in minimal time that is called resource constrained synthesis. The proposed scheduling sets a cut in the critical path to reduce the latency of the data flow graph.

The root nodes are calculated from the graphical description and the critical path is determined. The algorithm merges the nodes that has data dependency, which is of same type and has minimum two external inputs that is decomposed into parallel form as shown in Figure 7, the condition that is considered is the last node should have single output edge, if predecessors have one output edge than the both the nodes are merged and formed into single node or if the node has more than one output edge the node should not be disturbed and a cut in the path is set and the current node is moved to previous cycle where it meets the hardware constraint problem. If the problem satisfies the condition, a node is inserted in the previous cycle else it takes up the critical path. If the critical path is cut, the latency of the system is reduced which leads to the reduction in clock cycle of the entire system without any change in the hardware constraint. A cut in the critical path i.e. between node 3 and node 4 is made, the most serial is converted into most parallel form which leads to the reduction in single control step without affecting the hardware constraint 3

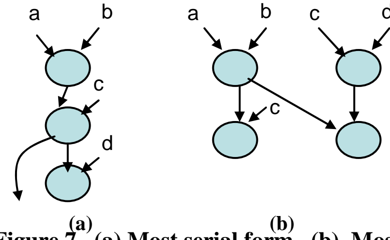


Figure 7 (a) Most serial form (b) Most parallel Form

adder and 2 multiplier. Hardware is allocated according to data dependency of the nodes. Schedule and allocation algorithm is detailed in Figure 8. In the elliptic wave filter benchmark (EWF), the clock period was constrained such that add operation takes one cycle and multiply operation takes two clock cycle.

The aim of allocation/binding is typically to minimize factors such as the number of resources used and the amount of wiring and steering logic (e.g. multiplexers) required to connect resources. A simplest case of minimizing is minimizing only the number of resources used (i.e. ignoring wiring and steering logic). In this case the standard technique involves building a compatibility graph from the input expression. The compatibility graph has nodes for each operation in the expression and an undirected edge (n_1, n_2) iff n_1 and n_2 can be computed on the same resource (i.e. if they do not occur in the same time-step and there is a single resource type capable of performing the operations corresponding to both n_1 and n_2. Each clique in the compatibility graph corresponds to operations which can share a single resource. The minimum number of functional units required could be calculated using maximal clique algorithm for the scheduled graph.

5.2 Buffer Size Estimation

The scheduled Control Data Flow Graph (CDFG) is partitioned into hardware and software module, buffer size and system delay is estimated by analyzing the data flow patterns of the CDFG. According to the data dependency, the nodes are assigned to the available hardware resources. The scheduled CDFG is partitioned using four different methods and its buffer size and system delay for all scheduling algorithms are estimated using edges. The life time of each edge is calculated by labeling the edges which is used to detect the variables with non-overlapping lifetimes. The variables with non-overlapping lifetimes can share the same register in a buffer, which leads to buffer size reduction.

Four models have been compared in the table for an elliptic filter benchmark which is scheduled using

different scheduling algorithms. The first method proves to have less buffer size than all the other methods.

```
Int i, j, k, l, m, lat;
N1    "set the hardware constraints";
N2    "number of hardware resources in each level";
N3    "number of nodes used in critical path(CP)";
N4    "number of nodes in a cluster removing the root node";
O     "number of output edges of the node";
;
S  i U j;
Cs  cs+3;
Lat  ;
While (
"calculate all the critical paths which has latency > Lat and N3"
while ( N3  0){
"find (vi, vj) ε, C1  node with single output edge vi with two or more same
type predecessors(vi-1..) having external inputs in CP "
Merge vi and (vi-1,vi-2..) and form a cluster
*** If  (O(i-1) < 2)
Remove and form root node with external inputs
i=i-1;
N4 = N4-1;
if (N4  0)
go to ***
else
end;
Else
If N2(L) < N1
Update the list
Else
N2(L)  0;
End;
Cut O(j)
"check for the availability of hardware resources from L-1 to 0"
"insert a root node rm with external inputs of vi,vi-1 using ASAP scheduling
algorithm "
"node vi takes the predecessors rm and pred(vi-1)"
}
}
```

Figure 8 Scheduling and Allocation Algorithm

Method I: Identifies the functional unit which requires many clock cycles and partitions that section into hardware which proves to be better than all other cases. The other methods uses maximal clique partitioning which is used to identify critical paths.

Method II: The functional units in that critical path are implemented in hardware and other functional units in software.

Method III: Functional units in the critical paths are implemented in software and other functional units are implemented in hardware.

Method IV: All the above three methods are based on data dependency. But results in the fourth method show when exactly data dependency is avoided to the maximum, when partitioning the functional units.

6. EXPERIMENTAL RESULTS

The proposed algorithms were applied to a handful of simple benchmarks and the results were obtained as tabulated in table 1.

TABLE 1 RUNTIME COMPARISON

	Loop	Basic block 1 (mul int)	Basic block 2 (mult long)	S/W runtime in cycles (100 ns) approx	Partition runtime in cycles (100ns) approx	Pure H/W(in 100 ns)	Speed up by partitioning
Dct	√	-	-	26667	23222	-	1.14
Diffeq	-	√	-	645	357	4.5	1.8
Ellip	-	√	-	952	538	0.375	1.76
Fir	-	√	-	769	402	0.192	1.9
Iir	-	√	-	714	366	0.35	1.95
Lattice	√	-	√	13333	8855	-	1.5
Nc	√	-	-	16000	15161	-	1.05
Volterra	√	-	-	16000	15588	-	1.02
Wavelet	√	-	-	13333	12922	-	1.03
Wdf7	-	-	√	20000	13479	-	1.5

TABLE 2 PERCENTAGE OF INSTRUCTIONS PARTITIONED AND CORRESPONDING SPEEDUP

Benchmark	Total Of number instructions	Instructions partitioned For SW	Instruction partitioned for HW	%of Instructions partioned	% runtime of the instructions partitioned	Speedup in % by partitioning
Dct	3504	3481	23	0.7	12	114
Diffeq	410	330	80	17	59.6	180
Ellip	574	494	80	12.45	58.2	176
Fir	404	324	80	17.75	63.9	190
Iir	382	302	80	18.8	65.3	195
Lattice	2613	2589	24	9.2	38.1	150
Nc	3886	3862	24	0.6	5.3	105
Volterra	2664	2640	24	0.9	2.6	102
Wavelet	3617	3593	24	0.7	3.1	103
Wdf7	3958	3692	266	5.5	37.4	150

There were three sections of code that were frequented in most of the benchmarks. Two were procedure calls representing integer multiplication and long integer multiplication. The third was a conditional loop. These were selected manually for hardware synthesis out of the candidate partitions selected by applying the fuzzy partitioning algorithm with a cutoff ratio of 1.5 and percentage of partitioned instruction is tabulated in table 2 and circuits were synthesized for the target device xc2v8000-5-ff1152. Many of the benchmarks showed significant speedup on partitioning while using very less resources compared to pure hardware implementation. Particularly notable are the rows for the ellip and iir benchmarks. They show a speedup of 1.76 and 1.95 respectively, while showing high savings in hardware resources compared to pure hardware implementation (slices of 0%,0% in partitioned approach compared to 55%,18% in pure hardware, and Multipliers of 25%,25% in partitioned approach compared to 116%,116% respectively in pure hardware which exceeds the multiplier resources available, as also seen in figure 10).

A comparative bar chart of the speedup shown in figure 9 shows to what extent each program's size in terms of number of cycles consumed per run gets reduced. The results with certain benchmarks (dct, nc, volterra, wavelet) may not be impressive at first look, but from figure 11, it is seen that the temporal size, (temporal size is the percentage of runtime of the partition as defined in [5]), for the partition selected manually out of the candidate partitions, with small size as a desired criteria to make easy the processing of dataflow extraction, is very low compared to other benchmarks. Figure 12 shows the profiling results of the diffeq benchmark. First three columns show instruction word location, the binary opcode and opcode in

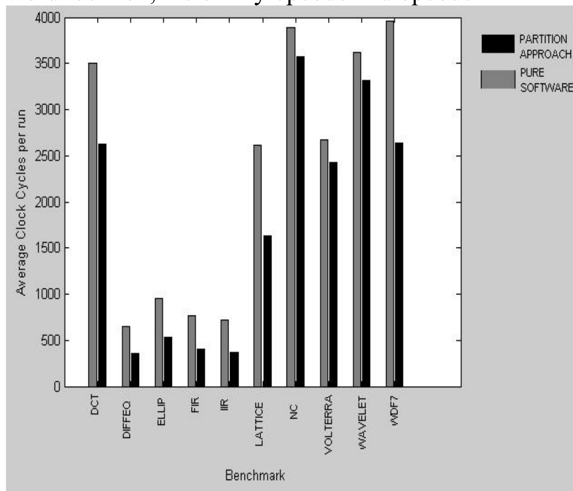


Figure 9 – Comparative Bar chart of Speedup gained

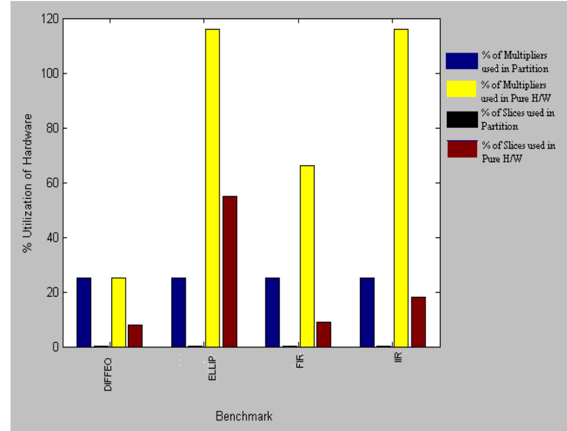


Fig.10.Comparative Bar chart of Percentage of Hardware Utilization

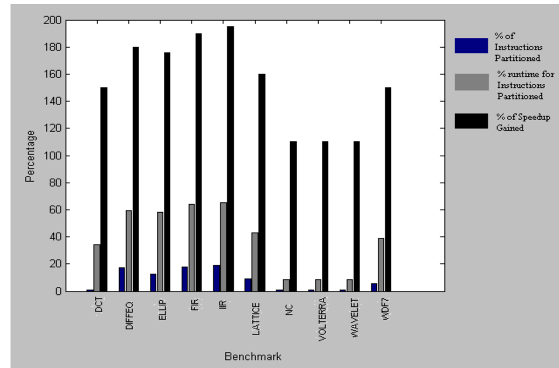


Figure 11 –Bar chart of Percentage of Instructions partitioned and corresponding speedup

```
0x01F0 0xCFD9 movff    _FSR2L,_POSTDEC1      82
0x01F1 0xFFE5 nop                       0
0x01F2 0xCFE1 movff    _FSR1L,_FSR2L         82
0x01F3 0xFFD9 nop                       0
0x01F4 0xC000 movff    0x00,_POSTDEC1        82
0x01F5 0xFFE5 nop                       0
0x01F6 0xC001 movff    0x01,_POSTDEC1        82
0x01F7 0xFFE5 nop                       0
0x01F8 0xC002 movff    0x02,_POSTDEC1        82
0x01F9 0xFFE5 nop                       0
0x01FA 0xC003 movff    0x03,_POSTDEC1        82
0x01FB 0xFFE5 nop                       0
0x01FC 0x0E02 movlw    0x02                  82
0x01FD 0xCFDB movff    _PLUSW2,0x00          82
0x01FE 0xF000 nop                       0
0x01FF 0x0E03 movlw    0x03                  82
0x0200 0xCFDB movff    _PLUSW2,0x01          82
0x0201 0xF001 nop                       0
0x0202 0x0E04 movlw    0x04                  82
0x0203 0xCFDB movff    _PLUSW2,0x02          82
0x0204 0xF002 nop                       0
0x0205 0x0E05 movlw    0x05                  82
0x0206 0xCFDB movff    _PLUSW2,0x03          82
0x0207 0xF003 nop                       0
0x0208 0x5000 movf     0x00,w,0               82
0x0209 0x0100 movlb    0x00                  82
0x020A 0x6F90 movwf    0x90                  82
0x020B 0x5001 movf     0x01,w,0               82
0x020C 0x6F91 movwf    0x91                  82
0x020D 0x5002 movf     0x02,w,0               82
0x020E 0x0100 movlb    0x00                  82
0x020F 0x6F92 movwf    0x92                  82
0x0210 0x5003 movf     0x03,w,0               82
0x0211 0x6F93 movwf    0x93                  82
0x0212 0x0100 movlb    0x00                  82
0x0213 0x5190 movf     0x90,w,1               82
```

Figure 12 – Profiling results for Diffeq

average 645 clock cycles per run . Using the technique to extract dataflow description for this block, the following

set of input and assembly format respectively. Last column shows the number of times the instruction location was traversed. The part shown is the procedure for integer multiplication. The profiling result showed that this procedure call with 71 instruction cycles had each instruction in it consuming 82 cycles when the program was run repeatedly to accumulate data for a 10,000 cycle period which output parameters for the coprocessor– shown in figure 13 - was obtained. A savings of 53 cycles was seen with an 18-cycle procedure call using the coprocessor - A speedup of 3.94 times the original call, and an overall speedup of 1.8 for the whole program. Similarly, figure 14 shows profiling results for the dct benchmark. The instruction rlcf, which means rotate left with carry, takes as input the carry bit of status register, and generates again as output the carry bit. This means we need to hold the carry bit in the initial-state as well as the final-state. This emphasizes that the bits of status registers also need to be held in the final-state if they are going to be affected by the present instruction and in the initial-state if they are going to be used in the instruction execution. The corresponding CDFG is shown in figure 15. The status bits have been omitted from consideration in our algorithm for the sake of brevity, since they are also a part of a register. The CDFG derived after merge operation is shown in figure 16. The hardware realization of the CDFG represents the body of the loop, with registers to hold the result of computation for each iteration of the loop. The coprocessor communicates with the processor using a handshake protocol. The processor issues a go signal after placing all the required input parameters for computation, on the data lines. The coprocessor starts computation at the edge of the clock pulse after receiving the go signal. Each clock pulse signifies the start of a new iteration. After computation for all iterations are complete, the coprocessor issues the done signal, placing the output parameters on the data lines. Hence there is no need for clock synchronization. The coprocessor can run independently from the processor, and both can run tasks simultaneously. Communication buffers can be used instead of the data lines, in which case, the maximum of the number of input parameters and the number of output parameters is the buffer size needed for communication. The estimates of buffer size requirements for each partition block are shown in table 3. The buffer size requirement for source level partitioning is shown in table 5. The table 4 shows the number of adders/subtractors and registers required for different practical implementations with and without forward branching. The present work has performed merging with forward branching and hence, there is a good improvement in the utilization of resources. For ex: the implementation of DCT algorithm reported using merging[25] alone requires two adders/subtractors and thirty nine registers. However, in this work by the novel use of merging with forward branching the number of registers reduces to 22. Similar results obtained for other real time implementations are reported in table 4.

Input parameters

Parameter	Comment
G	1st byte of Operand1
H	2nd byte of Operand1
I	1st byte of Operand2
J	2nd byte of Operand2

Note: The compiler used a little endian representation. Hence 2nd byte represents most significant byte.

Output parameters

Parameter	Value	Comment
X	Lower byte of (G*I)	1st byte of result
Y	Higher byte of (H*I)	Sideeffect of s/w code on register ProdH
Z	Lower byte of (G*J) + Lower byte of (H*I) + Higher byte of (G*I)	2nd byte of result

Figure 13 – Dataflow extraction for Diffeq

```
0x0B55 0x5000 movf     0x00,w,0                 144
0x0B56 0x2600 addwf    0x00,f,0                 144
0x0B57 0x3601 rlcf     0x01,f,0                 144
0x0B58 0x3602 rlcf     0x02,f,0                 144
0x0B59 0x3603 rlcf     0x03,f,0                 144
0x0B5A 0x0EFF movlw    0xff                     144
0x0B5B 0x2604 addwf    0x04,f,0                 144
0x0B5C 0x2205 addwfc   0x05,f,0                 144
0x0B5D 0xC004 movff    0x04,0x0a                144
0x0B5E 0xF00A nop                               0
0x0B5F 0xC005 movff    0x05,0x0b                144
0x0B60 0xF00B nop                               0
0x0B61 0x0E00 movlw    0x00                     144
0x0B62 0x5C03 subwf    0x03,w,0                 144
0x0B63 0xE108 bnz      $+0x12   ;(0x16d8)       144
0x0B64 0x0E80 movlw    0x80                     144
0x0B65 0x5C02 subwf    0x02,w,0                 144
0x0B66 0xE105 bnz      $+0xc    ;(0x16d8)       144
0x0B67 0x0E00 movlw    0x00                     0
0x0B68 0x5C01 subwf    0x01,w,0                 0
0x0B69 0xE102 bnz      $+0x6    ;(0x16d8)       0
0x0B6A 0x0E00 movlw    0x00                     0
0x0B6B 0x5C00 subwf    0x00,w,0                 0
0x0B6C 0xE3E8 bnc      $-0x2e   ;(0x16aa)       144
```

Figure 14 – Profiling Results for Dct

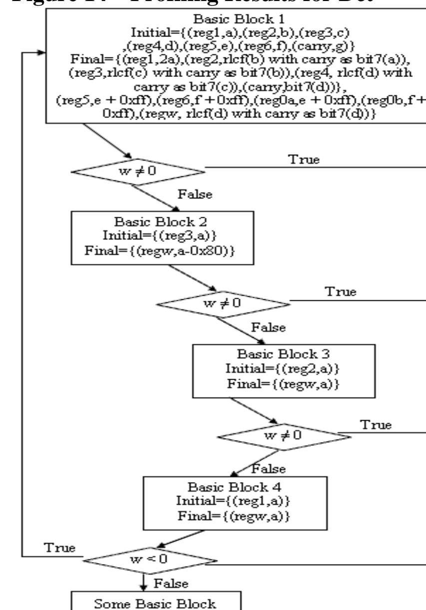
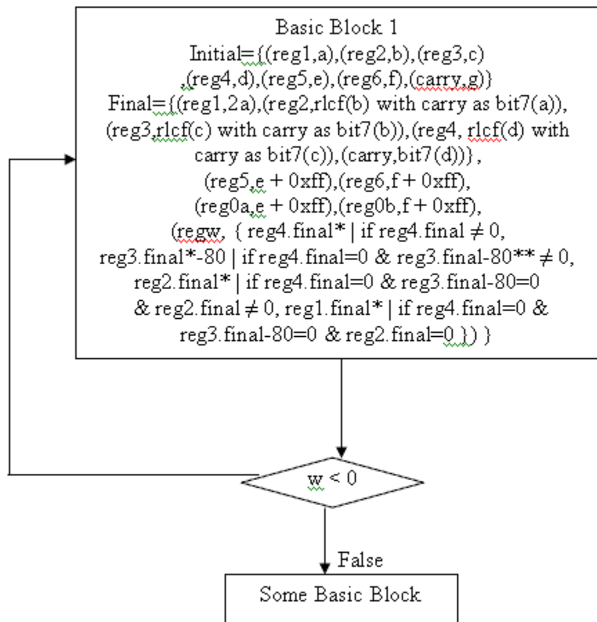


Figure 15 – CDFG for Dct without Merging operation

Table 3 – Buffer size/ Data bus width Estimates

Partition Block	Buffer size/ Data bus size in bytes
Loop1	7
Basic block 1 (mul-int)	4
Basic block 2 (mul-long)	8



*value expansion omitted for the sake of brevity
**subexpressions can be deduced to avoid

Figure 16 – CDFG for Dct after Merging operation

7. CONCLUSION

The experimental results of the dataflow extraction based approach to partitioning of software binaries shows significant speedup compared to pure software implementation, using significantly less hardware resources. The approach can be used for dynamic partitioning [3] of software binaries and extended to cover arbitrary control flow. The buffer size estimation can be applied for software binaries using different scheduling and partitioning algorithms.

References

[1] G. Stitt, and F. Vahid, "Hardware software partitioning of software binaries", In *International Conference on Computer Aided Design (ICCAD '02),* Nov.10-14,2002, pp. 164-170.

[2] G. Stitt, and F. Vahid, "A Decompilation approach to partitioning software for Microprocessor/FPGA platforms", In *Proceedings of Design, Automation and Test in Europe (DATE '05),* Vol. 1., pp. 396-397.

[3] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: A first approach.", In *Proceedings of Design AutomationConference (DAC '03),* June 2-6,2003, pp. 250-255.

[4] R. Camposano, "Path-based scheduling for synthesis", In *IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 10, Issue 1, Jan. 1991, pp. 85-93.

[5] A. Jantsch, P. Ellervee, J. Oberg, and A. Hemani, "A Case Study on Hardware/Software Partitioning", In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[6] http://gpsim.sourceforge.net

[7] http://www.xilinx.com

[8] http://www.microchip.com

[9] K. O'Brien, M. Rahmouni, and A. Jerraya, "DLS: A scheduling algorithm for high-level synthesis in VHDL", In *Proceedings of 4th European Conference on Design Automation with the European Event in ASIC Design*,Feb.22-25, 1993, pp. 393–397.

[10] R. Namballa, N. Ranganathan, and A. Ejnioui, "Control and data flow graph extraction for high-level synthesis", In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI - Emerging trends in VLSI systems design (ISVLSI 2004),*Feb.19-20, 2004, pp. 187–192.

[11] M. Rahmouni, and A. Jerraya, "PPS : A pipeline path-based scheduler", In *Proceedings of European Design and Test Conference,* March 6-9,1995, pp. 557–561.

[12] R. Gupta, and G. De Micheli, "Hardware-Software Cosynthesis for Digital Systems", In *IEEE Design & Test of Computers,* September, 1993, pp. 29-41.

[13] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers", In *IEEE Design & Test of Computers,* December, 1993, pp. 64-75.

[14] Greg Stitt, Frank Vahid, "New Decompilation Techniques for Binary-level Co-processor Generation", In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*", 2005, pp. 547 – 554.

[15] Rafael Ruiz-Sautua, María C. Molina, and José M. Mendías (2007), Exploiting Bit-Level Delay Calculations to Soften Read-After-Write Dependences in Behavioral Synthesis, IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems, vol. 26, no. 9, 1589-1601.

[16] David Zaretsky, Gaurav Mittal, Xiaoyong Tang, Prith Banerjee (2004), Evaluation Of Scheduling And Allocation Algorithms While Mapping Assembly Code Onto FPGAs, Proceedings of the 14th ACM Great Lakes symposium on VLSI, pp. 397-400.

[17] Greg Stitt and Frank Vahid , Binary-Level Hardware/Software Partitioning of MediaBench, NetBench, and EEMBC Benchmarks, Technical Report UCR-CSE-03-01. January 2003.

[18] Gaurav Mittal, David Zaretsky, Xiaoyong Tang and Prith Banerjee, An Overview of a Compiler for Mapping Software Binaries to Hardware(2007), IEEE Transactions On Very Large Scale Integration Systems, vol. 15, no. 11, pp. 1177-1190.

[19] Greg Stitt and Frank Vahid, A Decompilation Approach to Partitioning Software for Microprocessor/FPGA Platforms, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05), 2005, pp. 396-397.

[20] Papa G. and Silc J. (2000), 'Multi-objective genetic scheduling algorithm with respect to allocation in high-level

synthesis', Proceedings of the 26th Euromicro Conference, Vol. 1, pp. 339-346.

[21]Heejin Yoo and Dosoon Park (1999), 'A scheduling algorithm for pipelined data path synthesis with gradual mobility reduction', Proceedings of first IEEE Asia Pacific Conference, pp. 51-54.

[22] Gang Wang, Wenrui Gong, Brian DeRenzi, and Ryan Kastner (2007), 'Ant Colony Optimizations for Resource- and Timing-Constrained Operation Scheduling', IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems, Vol. 26, No. 6, pp. 1010- 1029.

[23] G.Stitt, and F. Vahid (2005) , New Decompilation Techniques for Binary-level Co-processor Generation, IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2005, pp. 547-554

[24] Gaurav Mittal David C., Zaretsky Xiaoyong Tang and P.Banerjee (2004), Automatic Translation of Software Binaries onto FPGAs, Proceedings of the 41st annual conference on Design automation. Pp. 389 - 394

[25]Gaurav Mittal, David Zaretsky, Xiaoyong Tang and Prith Banerjee (2007), An Overview of a Compiler for Mapping Software Binaries to Hardware, IEEE Transactions On Very Large Scale Integration Systems, vol. 15, no. 11, pp. 1177-1190.

Table 4 Merging for binary level partitioning

	Without Merging		Merging		Merging forward branching	
	Adders/ subtractors	Registers	Adders/ subtractors	Registers	Adders/ subtractors	Registers
DCT	14	125	2	39	2	22
Diffeq	24	123	1	8	-	-
Ellip	24	123	1	8	-	-
Fir	24	123	1	8	-	-
Iir	24	123	1	8	-	-
Lattice	14	124	2	39	2	22
NC	14	124	2	39	2	22
Volterra	14	124	2	39	2	22
Wavelet	14	124	2	39	2	22
Wdf7	36	125	2	47	-	-

Table 5 Buffer size estimation for source level partitioning

Algorithm	Number of partitions	Method I			Method II			Method III			Method IV			
		Edge cut	Buffer size	delay	Edge cut	Buffer size	delay	Edge cut	Buffer size	delay	latency	Edge cut	Buffer size	delay
FDLS	2	16	4	234	9	3	490	9	3	312	23	33	10	410
MOGS	2	16	4	234	9	3	490	9	3	312	23	33	10	410
LS	2	16	4	234	9	3	490	9	3	312	23	33	10	410
ALAP	2	16	4	234	8	5	490	8	5	312	23	33	10	410
ASAP	2	16	4	234	9	3	490	9	3	312	23	33	10	410
SAA	2	16	3	243	5	2	508	5	2	303	24	33	10	419