Hardware/Software Tradeoffs in ADAM

Jagannath Raghavendran and Alice Parker

CENG Technical Report 93-28

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4476

July 1993

1 Introduction

Application-specific systems are increasingly common. Many application-specific systems have microprocessors implementing some tasks. Such tasks are generally not in critical timing loops of real time systems, but provide support functions, exception handling, or control. In some systems, microprocessors, along with interface, memory and glue logic, provide most of the functionality.

A major design tradeoff occurs when a decision is made to use an off-the-shelf microprocessor instead of special-purpose chips or a custom-designed microprocessor. Much of the effort goes into the software design and implementation in these cases, and the advantages of such an implementation cannot always be couched in quantitative parameters like cost or performance.

Nonetheless, we are attempting to fashion a system which makes a restricted tradeoff between off-the-shelf microprocessors and custom IC's based on cost and performance predictions. We assume that the algorithm to be implemented is already specified to such a degree of detail in a hardware-descriptive language that code generation for a microprocessor implementation carries an insignificant cost. We also ignore, for the present, the costs of testing for both hardware and software, with future plans to integrate such costs into our model.

We approached the tradeoff problem by developing software cost and performance predictors for use with off-the-shelf microprocessors. We implemented prototype software to demonstrate our prediction capabilities and illustrated our tradeoff capability with a simple example. The remainder of this paper describes the prediction approach, and the illustrative example.

2 The Prediction Approach

This section describes an approach to the estimation of software cost and performance. Typically these are measured by the amount of memory required and the execution time of the program respectively. This program takes as input a dataflow graph and data about the microprocessor and computes the memory requirements and the execution time of the program.

The approach followed here is to divide the set of instructions into three types - arithmetic instructions, data movement instructions and branch instructions. Each of these instruction types is then estimated independently. First, it is assumed the fanout of every operation in the dataflow graph is one. Later this assumption is relaxed.

2.1 Estimating Arithmetic Instructions

In the software implementation of the dataflow graph, the number of arithmetic instructions would be equal to the number of arithmetic operations in the dataflow graph [1]. It is important to determine the number of arithmetic instructions that reference memory operands and those that use only register operands in order to determine the number of data movement instructions.

Definition 2.1 *An operation in the dataflow graph all of whose inputs are from the outputs of some other operations is called an* **Internal Operation** *and those operations that are not Internal Operations are called* **Peripheral Operations**.

It is assumed that the dataflow graph contains only operations with two inputs and one output. In most microprocessors, arithmetic instructions can refer to one memory operand at most. In our model of the microprocessor, we assume that each arithmetic instruction has two operands, the source operand and the destination operand. Also we assume each arithmetic instruction can reference one memory operand while the other operand is in the register. We also assume that the inputs to the dataflow graph are stored in memory when execution starts.

Now, let us consider the peripheral operations. For those peripheral operations with one input, the arithmetic instruction which performs the peripheral operation can address the memory operand. There are special cases where the arithmetic operation cannot reference the memory operand directly. We shall discuss these cases later. For the peripheral operations with two inputs, the arithmetic instruction can reference one of the operands from memory and the other operand from a register. Data movement instructions are required to load the first operand into the register. Here, the number of memory referencing arithmetic instructions would be equal to the number of peripheral operations.

We define the following notation:

d	Number of inputs to the dataflow graph

[1]Ignoring bit-width differences, which cannot be ignored in a production-quality predictor.

m	Number of operations in the dataflow graph
q	Number of internal operations

The number of peripheral operations is $m-q$, the number of memory referencing arithmetic instructions is $m-q$, and the number of register referencing arithmetic instructions is q.

An internal operation may also require a memory reference, and that must also be estimated.

3 Estimating Data Movement Instructions

Estimation of data movement instructions is a more complicated task. The number of data movement instructions is dependent on the topology of the graph and the sequence in which the operations are performed.

Data movement instructions are of two types: register to register and memory to register. For purposes of estimation, we consider register to memory and memory to register data movement instructions as equivalent. If the instruction times are different, then we take the average time of the two instructions.

First, we define the following terms:

Definition 3.1 *A* **Minimal Schedule** *of instructions is the sequence of operations executed such that the number of data movement instructions required is minimal.*

Definition 3.2 *A* **Complete Dataflow Graph** *is a dataflow graph where the operations and the edges of the dataflow graph form a complete binary tree. A complete dataflow graph is shown in Figure 1.*

We conjecture that, for a complete dataflow graph with 2^n-1 operations, at least n registers are required so that the dataflow graph can be implemented in a minimal schedule.

We also conjecture that, for a complete dataflow graph with 2^n-1 operations, implemented on a microprocessor with m registers, the number of memory references required would be max(n-m,0). These two conjectures allow us to predict memory references.

Any dataflow graph whose operations all have a fanout of one can be formed into a network of complete dataflow subgraphs. An example of such a decomposition is shown in Figure 2.

From the above result, the number of data movement instructions can be computed by computing the number of moves for each of the dataflow subgraphs. If m is the number of operations and h_i is the height of the complete dataflow subgraph i, then we have

$$m = \sum_{i=1}^{n}(2^{h_i} - 1)$$

3.1 Estimating Memory to Register Moves

To be able to estimate the number of move instructions, we should be able to determine heights of the complete dataflow subgraphs. Clearly, it is not possible to compute the heights of the dataflow subgraphs quickly.

To estimate the number of moves, we estimate the number of dataflow subgraphs that exist in the given dataflow graph which would require a move operation. For instance, let us consider a microprocessor with four registers. For this microprocessor, a complete dataflow graph with 31 operations would require one move operation. Now, in the given dataflow graph we would have to find the number of dataflow graphs with 31 operations. The number of such dataflow subgraphs would give the number of moves required. If R_{max} is the estimated number of moves required, then

$$R_{max} = \frac{m}{2^{n+1} - 1}$$

where m is the estimated number of operations in the dataflow graph and n is the number of registers in the microprocessor. $2^{n+1} - 1$ is the number of operations in a complete dataflow subgraph which requires a single memory move.

The above method works quite well for dataflow graphs that are not skewed. To take care of dataflow graphs that are skewed, we use the concept of **Internal Operations** that has been defined earlier. We follow the same method as described above, but instead of the total number of operations, we use the number of internal operations. If R_q is estimated the number of moves required due to internal

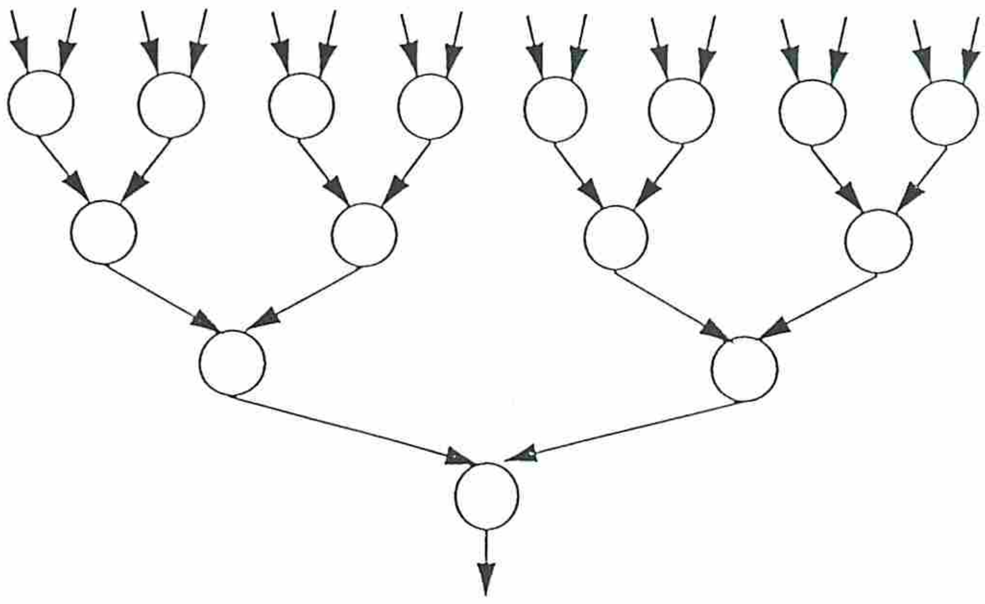
Figure 1: A Complete Dataflow Graph

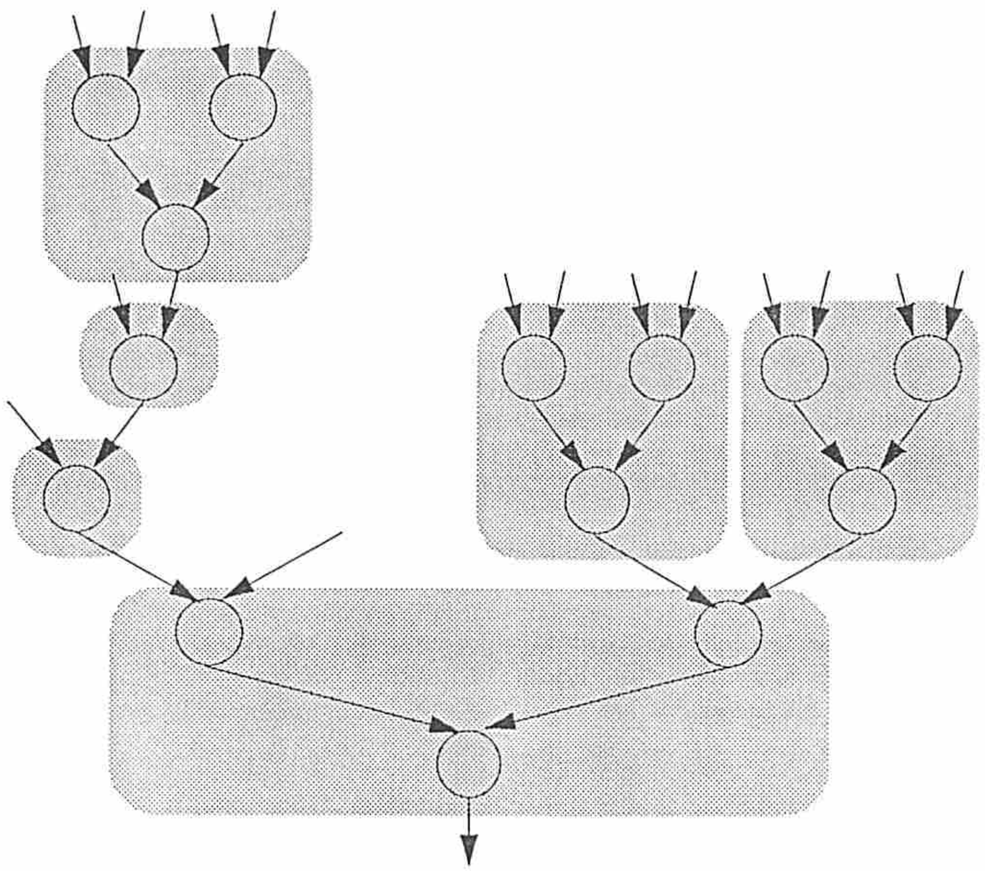
Figure 2: A dataflow graph formed of complete dataflow subgraphs

operations, then

$$R_q = \frac{q}{2^n - 1}$$

where, q is the number of internal operations in the dataflow graph.

The estimated number of moves would be the minimum of R_{max} and R_q. This is because, when the dataflow graph is skewed, R_{max} gives a value much higher than the actual value. In this case, R_q gives a more accurate value.

Extra moves would also be required depending upon how the complete dataflow subgraphs are connected. The number of moves required due to this is estimated by

$$min\{\frac{R}{2}, m \bmod (2^{n+1} - 1)\}$$

Move instructions are also required for operations both of whose inputs are primary inputs of the dataflow graph. The number of operations both of whose inputs are primary inputs, C, is given by

$$C = d - (m - q)$$

3.2 Estimating Register to Register Moves

The code generated by a compiler contains move instructions between registers. The execution times of these instructions are considerably less than the register to memory move instructions. The compiler usually generates these instructions in the following cases:

1. when the operation is commutative and both the inputs are from operations that have multiple fanouts,
2. when the operation is non-commutative and one of the input values cannot be destroyed, or
3. when the operation has as its input a primary input to the dataflow graph.

We define the following notation:

P_c	The probability of a commutative operation.

P_{nc}	The probability of a non-commutative operation.
$TotalFanout$	The sum of the fanouts of the multiple fanout operations.
m	The number of operations in the dataflow graph.
T_f	The number of multiple fanout operations.
q	The number of internal operations.
d	The number of inputs to the dataflow graph.

The probability that an operation in the dataflow graph is a multiple fanout operation is $\frac{T_f}{m}$, and the probability that an input to an operation is from a multiple fanout operation is given by $\frac{T_f}{m}$.

The number of register to register moves due to the first case, described above, is estimated by

$$P_c \times \frac{T_f}{m} \times \frac{T_f}{m} \times TotalFanout$$

The number of register to register moves due to case 2 is estimated by

$$\frac{P_{nc}}{2} \times m$$

The number of register to register moves due to the third case is estimated by

$$P_c \times \frac{T_f}{m} \times 2 \times ((m - q) - d)$$

The number of register to register moves is estimated by the sum of all the above cases.

4 Register Usage Estimation

It is necessary to estimate the usage of registers to be able to estimate the number of data movement instructions due to multiple fanout operations. This estimate of average register usage is used to allocate values to either registers or memory.

We define the following notation:

k	The number of MFOs in the dataflow graph.
T_f	The total fanout of the multiple fanout operations.
d	The number of primary inputs to the dataflow graph.
m	The number of operations in the dataflow graph.

The given dataflow graph is modified in the following way: For every operation with multiple fanout, one output edge is retained and all the other edges are cut off from this operation. These edges that are cut off are now considered as input edges to the modified dataflow graph. So, there are $d + T_f$ input edges to the dataflow graph, effectively.

To be able to estimate the register usage, we should be able to compute the size of the complete dataflow subgraphs in the dataflow graph. This is clearly not possible since the number of possible combinations of sizes is far too large. So, to estimate the most likely register usage, we assume that the dataflow graph is made of complete dataflow subgraphs which are all of the same size.

Assume there are x_1 complete dataflow subgraphs in the modified dataflow graph, all of the same size.

If the height of the dataflow subgraphs is h, then we have

$$x_1 \times (2^h - 1) = m$$

Equating the estimated number of inputs to the modified dataflow graph, we get

$$x_1 \times 2^h = d + T_f$$

From the above two equations, we get

$$x_1 = d + T_f - m$$

So, the height of these dataflow subgraphs is given by

$$\log_2(x_1) = \log_2(d + T_f - m)$$

The height of the trees is used to estimate the number of registers that are used for the execution of this dataflow graph. This value of register utilization is used to estimate the number of values that are in registers and in memory.

5 Multiple-Fanout Operations

Until now, we have discussed how to handle dataflow graphs in which all the operations have a fanout of one. In this section we shall deal with operations in the dataflow graph that have a fanout of more than one. In the case of Multiple Fanout Operations (MFO), the problem is complicated since the value generated by the MFO is alive until the completion of all the destination operations. This makes the estimation of the lifetime of the value difficult. We use a probabilistic model, similar to Kurdahi's wiring estimation model, to estimate the lifetime of the values generated by the MFOs.

We make the following assumptions:

- A value is born at the end of an arithmetic instruction and dies at the beginning of the last instruction that uses the value.
- Once a value dies, the storage location used by the value is free and can be used by another value.
- All the arithmetic instructions are considered to be points. Consequently, a value is alive from point i to point j. There are as many points as there are arithmetic instructions.

We define the following notation:

k	The number of MFOs in the dataflow graph.
v_i	The value generated by the MFO i.
f_i	The fanout of the MFO i.
m	The number of operations in the dataflow graph.
T_f	The sum of the fanouts of the MFOs.
$p_b(i,j)$	Probability of the value v_i from MFO i, being born at point j.
$p_d(i,j)$	Probability of the value v_i from MFO i, dying at point j.
$P_b(i)$	Probability of a value being born at point i.
$P_d(i)$	Probability of a value dying at point i.
V_i	Estimated number of values at the point i

Every value v_i with a fanout f_i has to be born between point 1 and point

$m - f_i$. So, assuming a uniform probability distribution we get

$$p_b(i,j) = \begin{cases} \frac{1}{m-f_i} & \forall\, j = 1,\ldots,m-f_i \\ 0 & \text{otherwise} \end{cases}$$

Similarly, every value v_i can only die beyond the point f_i. So, we get

$$p_d(i,j) = \begin{cases} 0 & \forall\, j = 1,\ldots,f_i \\ \frac{1}{m-f_i} & \forall\, j = f_i+1,\ldots,m \end{cases}$$

For a value to be active at a point i, it has to be born between point 1 and $i-1$, and it should die between the points i and k. Using the above criterion, the probability that a value j is active at a point i is given by

$$P(j,i) = \sum_{t=1}^{i-1} p_b(j,t) \times \sum_{t=i}^{m} p_d(j,t)$$

The expected number of values active at a point i is given by

$$E(i) = \sum_{t=1}^{k} P(j,i)$$

To get an estimate of the number of values active at any point i, we take the integer that is nearest to $E(i)$. So,

$$V_i = \lfloor E(i) + 0.5 \rfloor$$

Now, we make the assumption that the lifetime of a value generated by an MFO is proportional to the fanout of the MFO. The total lifetime of all the values generated by the MFOs is given by

$$L_{sum} = \sum_{i=1}^{m} V_i$$

So, the lifetime of the values are given by

$$L_i = \frac{f_i}{\sum_{j=1}^{k} f_j} \times L_{sum}$$

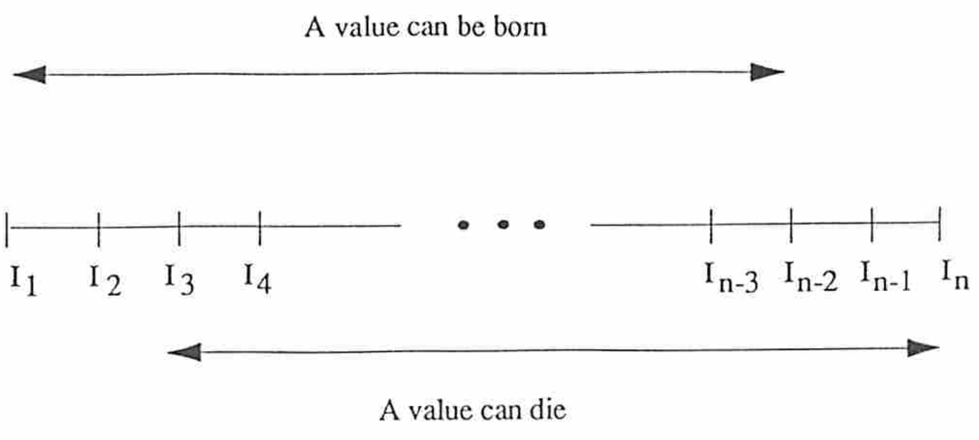

Figure 3: Birth and Death of a Value from a MFO with a fanout of two

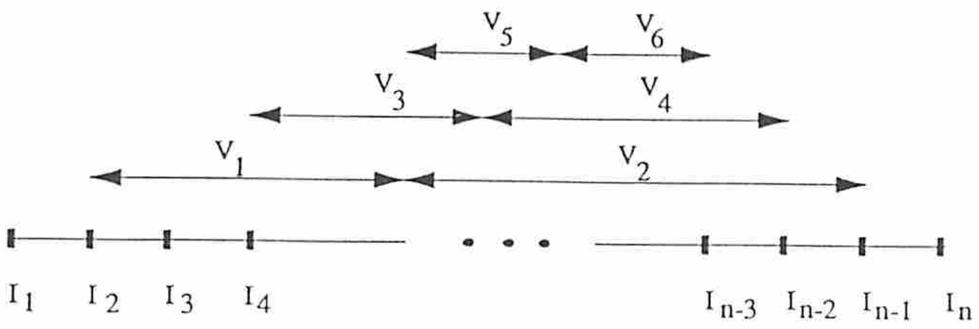

Figure 4: Allocation of Values to Storage Location

These values are then allocated to storage locations such that the number of storage locations with active values does not exceed V_i at any point i. The allocation of values is shown in Figure 4.

Now, we make use of the estimated number of registers used. From the maximum number of registers used at any point of time G_{max}, we can find the minimum number of registers that are free at any time. These unused registers are used to store the values generated by the MFOs. If registers are not available to store all the values, then they are allocated to locations in memory. This scheme allows us to estimate the number of memory references and register references.

5.1 Conditional Blocks and Loops

Conditional blocks and loops are evaluated as follows:

1. First the code size and execution time are estimated for each of the conditional blocks and loops.
2. Then the memory sizes for each of the blocks is added up to get the total memory size.
3. Conditional blocks are assigned a probability of execution for the block.
4. For loops, the number of iterations is given by the user. This is used to estimate the effective execution time for the block.
5. Then, it is necessary to estimate the number of moves due to the values being transferred between blocks. This is done as described below.

5.2 Some Other Prediction Issues

Some of the other issues that affect the cost and performance of software are

- the quality of the code generated by the compiler,
- the quality of the software written in the high level language,
- execution overlap of assembly instructions, and

- varvi

- varying addressing modes available for different instructions.

Most compilers cannot produce optimal code. Also compilers do not perform all types of optimization possible. Certain compilers are tuned to optimizing certain constructs in the source code, while other compilers are tuned for some other constructs. In the above estimation, this is taken care of by having a fudge factor called the Compiler Quality Factor.

The code generated by compilers depends to a large extent on the quality of the source code. If the source program is written such that any value that is generated by an arithmetic instruction is used as soon as possible, then the object code produced by compilers is close to optimum. Otherwise, values are unnecessarily stored in registers leading to non-optimal register usage. Consequently, this leads to non-optimal code. A fudge factor is used to account for this called the Source Code Quality Factor.

5.3 Execution Overlap

In the above estimation, it is difficult to take care of execution overlap of different instructions. This is because the amount of execution overlap depends on the ordering of the instructions. Also the addressing modes used by the instructions determine the amount of overlap. Branch instructions can change the sequence of instructions executed dynamically. Some other factors like the bus cycles used by co-processors can affect the execution overlap. All these factors make determining the amount of execution overlap almost impossible. Also, in our estimation, we are not concerned about the sequence of instructions, but the number of each instruction type. Again, we use a fudge factor called the Execution Overlap Factor to account for this.

6 Results

The results of the estimation program are shown in the tables below. Table 1 shows the results of the memory estimation. The estimated results are compared with the exact results. The exact results were obtained by translating the VHDL description to a C program by changing the syntax. This C program was compiled and the assembly code generated. From this assembly program, the exact values were obtained. This procedure was followed for obtaining exact values for memory and execution time

Design	Estimate (in bytes)	Exact (in bytes)	Difference (%)
AFT	218.66	216.00	1.23
AR Filter	135.00	132.00	2.27
FIR Filter	118.00	112.00	5.36
EW Filter	131.67	142.00	7.28
RAC	203.00	214.00	5.14

Table 1: Memory Estimation Results

Design	Estimate (clk cycles)	Exact (clk cycles)	Difference (%)
AFT	349.33	353.00	1.04
AR Filter	605.00	602.00	0.49
FIR Filter	382.00	381.00	0.26
EW Filter	398.67	409.00	2.53
RAC	672.67	683.00	1.51

Table 2: Execution Time Estimation Results

shown on Table 2. The instruction times are given in Table 3. A 5 MHz clock rate was assumed for the microprocessor.

7 Running the Program

This section describes the inputs, the outputs and invocation of the estimation program.

7.1 Inputs

The estimation program requires as inputs a description of the operations in the dataflow graph and data about the microprocessor. The two inputs are contained in two separate files. The operations description file, called the operation file, contains the following information for each operation: the operation name, the operation type,

Instruction	Type	Average Time (in cycles)	Best Time (in cycles)
mov	rr	3.0	2.0
mov	mr	8.0	6.0
cmp	rr	3.0	2.0
cmp	mr	9.0	7.0
add	rr	3.0	2.0
add	mr	9.0	7.0
sub	rr	3.0	2.0
sub	mr	9.0	7.0
mul	rr	28.0	27.0
mul	mr	34.0	32.0
div	rr	34.0	36.0
div	mr	39.0	42.00
jmp	su	7.0	9.0
jmp	un	7.0	9.0
bcc	su	9.0	6.0
bcc	un	7.0	6.0

Table 3: Instruction Execution Times for various instruction on 68020 microprocessor

- "rr" denotes register register operands.
- "mr" denotes memory register operands.
- "su" denotes successful jump.
- "un" denotes unsuccessful jump.

the number of inputs, the number of outputs, the fanout of each output, the number of values being transferred to another block and the destination block ID. Although not all of this information is being used in the estimation, it is anticipated that the extra information will be used by programs that invoke the estimation program.

The operation description file consists of different blocks, each block corresponding to a block in the VHDL description. Each block begins with one of the following keywords - "begin", "for", "if" or "else". The keyword "begin" marks the beginning of a non-conditional block, "for" marks the beginning of a repititive block, "if" and "else" are used for conditional blocks. Following this is the block id which is a unique number for each block. Then there is a field which gives the number of operations in the block. There is a list of operations, with description about each operation. After the list of operations are four more fields followed by the keyword "end". The first field gives the number of primary inputs of the dataflow graph that are also inputs to the block. The second field gives the number of inputs to the block from other blocks. The third and fourth fields are the same as the first and second, but they are outputs.

The operation description file can be automatically created from the DDS database by a program which extracts this information from the database.

The file with data about the microprocessor has information about the instruction times for the arithmetic instructions, data movement instructions and jump instructions. The file for the Motorola 68020 microprocessor is given below:

```
mov  rr  3.0  2.0
mov  mr  8.0  6.0
cmp  rr  3.0  2.0
cmp  mr  9.0  7.0
add  rr  3.0  2.0
add  mr  9.0  7.0
sub  rr  3.0  2.0
sub  mr  9.0  7.0
mul  rr  28.0  27.0
mul  mr  34.0  32.0
div  rr  34.0  36.0
div  mr  39.0  42.00
jmp  su  7.0  9.0
jmp  un  7.0  9.0
bcc  su  9.0  6.0
bcc  un  7.0  6.0
```

7.2 Outputs

The estimation program prints out the number of clock cycles taken by each block and also the total time for the whole program. The program also outputs the estimated number of bytes for the program.

7.3 Invoking the Program

The estimation program can be invoked from the C shell command line by typing "est" followed by the name of the operation description file.

8 Some Experiments with the Estimator

The software estimation program along with the behavioral partitioner CHOP were used in some tradeoff experiments which are described in this section. The example used was that of a Robot Arm Controller shown in Figure 5. We assume that a robot arm controller is being designed and the design is to be implemented partially with hardware and partially with software. The software portion is executed on a microprocessor and a co-processor is designed to obtain the required performance. We use the estimation programs to carve out portions of the dataflow graph that can be executed on a co-processor. The co-processor is implemented as an ASIC.

The tradeoff results are presented in Table 4. Each row in the table represents a tradeoff point. Each tradeoff point represents a different mix of hardware and software. The dataflow graph of the co-processor for each of the tradeoff points is shown in Figure 6.

9 Conclusions

This report describes a program for estimation of the software cost and performance. The memory estimation results and the execution time results are within 8% of the

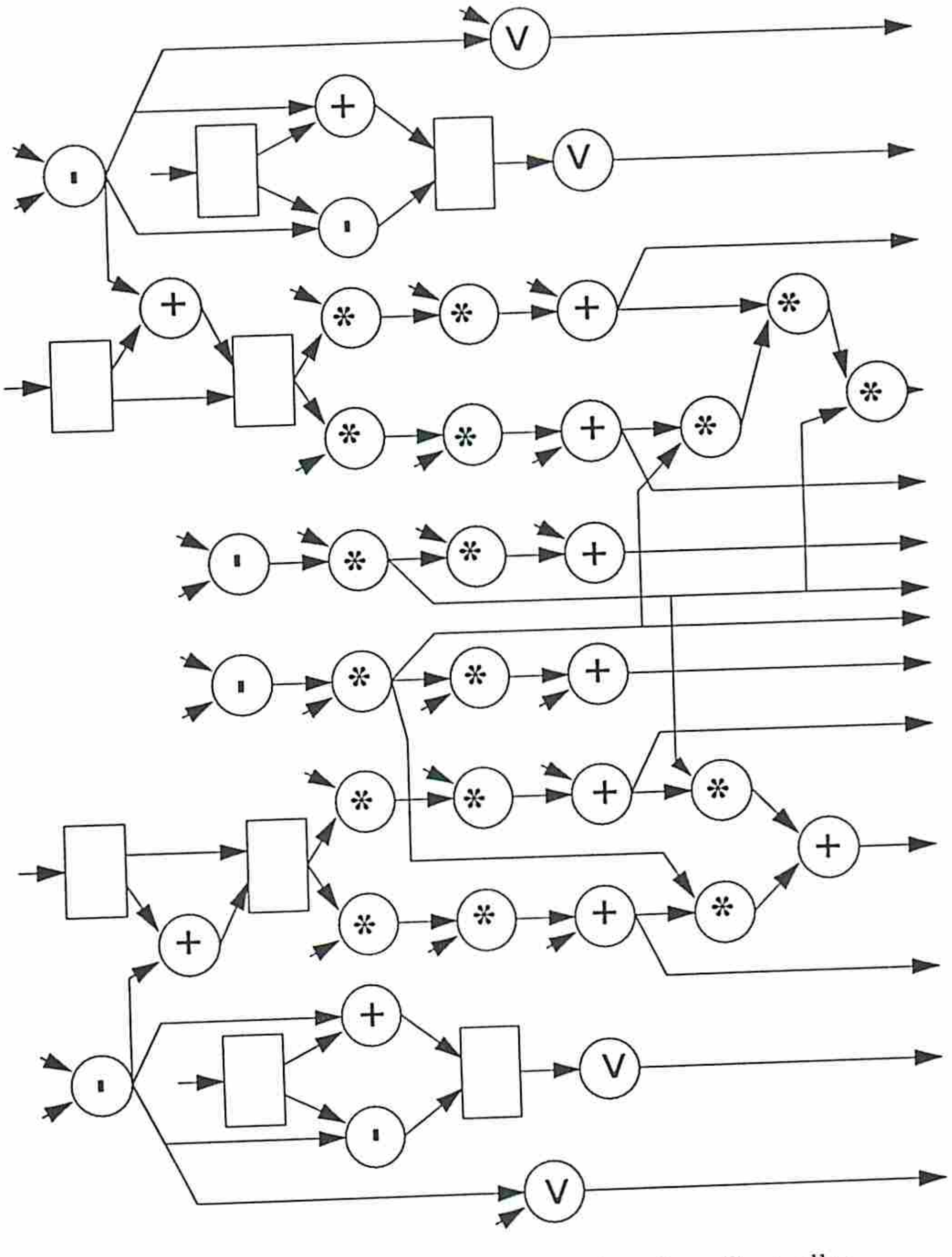

Figure 5: Dataflow Graph of the Robot Arm Controller

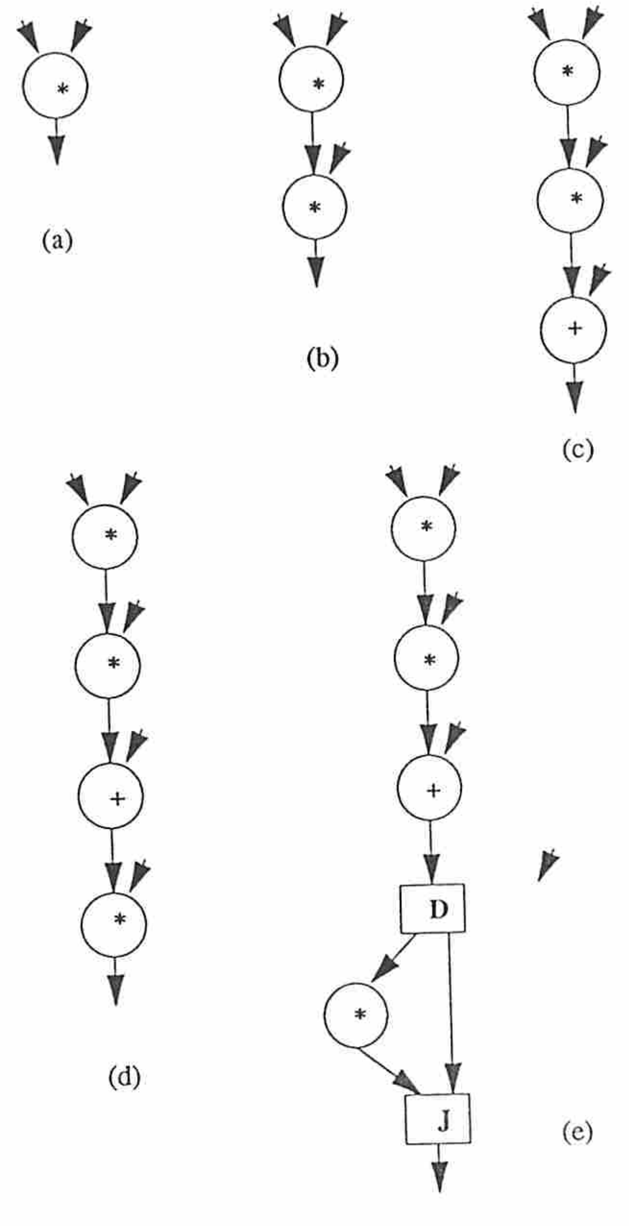

Figure 6: Different possible functionalities for the co-processor

Design	Software Time (in μsecs)	Hardware Time (in μsecs)	Total Time (in μsecs)
Software	134.3	-	134.4
1 mul	29.3	102.0	131.3
2 muls	56.5	36.0	92.5
2 mul, 1 add	47.7	39.0	86.7
3 mul, 1 add	54.7	26.0	80.7
(2,3) muls 1 add	25.8	39.0	64.8
Hardware	-	16.0	16.0

Table 4: Tradeoff Results on the Robot Arm Controller

exact values. The estimation program along with the behavioral partitioner CHOP have been used in performing experiments on hardware-software tradeoffs.

Appendix A: VHDL description for AR Filter

```
--
-- This is VHDL description of the AR Filter.
--

package TYPES is
  type SixteenBitVector is array(15 downto 0) of Bit;
end TYPES;

use work.TYPES.all;

package OPR16 is
  function "+"(OPN1,OPN2:SixteenBitVector) return SixteenBitVector;
  function "-"(OPN1,OPN2:SixteenBitVector) return SixteenBitVector;
  function "*"(OPN1,OPN2:SixteenBitVector) return SixteenBitVector;
end OPR16;

use work.TYPES.all;
use work.OPR16.all;

entity ARF is
  port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,
       a11,a12,a13,a14,a15,a16,a17,a18,a19,a20,
       a21,a22,a23,a24,a25,a26:in SixteenBitVector;
       out1,out2:out SixteenBitVector);
end ARF;


architecture Behaviour of ARF is
begin
  process
  variable e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,
           e11,e12,e13,e14,e15,e16,e17,e18,e19,e20,
           e21,e22,e23,e24,e25,e26,e27,e28,e29,e30:SixteenBitVector;
  begin

    e7:=a13*a14;
    e8:=a15*a16;
    e12:=e7+e8;
    e15:=a18+e12;
    e20:=e15*a22;
    e5:=a9*a10;
    e6:=a11*a12;
    e11:=e5+e6;
    e13:=e11+a17;
    e19:=e13*a21;
    e23:=e19+e20;
    e28:=a26*e23;
    e17:=a19*e15;
    e18:=e13*a20;
    e21:=e17+e18;
    e27:=a25*e21;
    e30:=e27+e28;
    e3:=a5*a6;
    e4:=a7*a8;
    e10:=e3+e4;
    out2<=e10+e30;
    e25:=a23*e23;
    e26:=e21*a24;
    e29:=e25+e26;
    e1:=a1*a2;
    e2:=a3*a4;
    e9:=e1+e2;
    out1<=e9+e29;
  end process;
end Behaviour;
```

Appendix B: VHDL description for Arithmetic Fourier Transform

```
--********************************************************************
--*                                                                  *
--*      VHDL DESCRIPTION FOR AN ARITHMETIC FOURIER TRANSFORM.       *
--*                                                                  *
--********************************************************************

--
--  This is the VHDL description of the Arithmetic Fourier Transform. This
--  algorithm was developed by Irving S. Reed of USC.
--
--


package TYPES is
  type SixteenBitVector is array(15 downto 0) of Bit;
end TYPES;

use work.TYPES.all;

package OPR16 is
  function "+"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function "-"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function "*"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function less(opn1,opn2:SixteenBitVector) return Boolean;
end OPR16;

use work.TYPES.all;
use work.OPR16.all;

package SHIFT is
  function rs1(opn1:SixteenBitVector) return SixteenBitVector;
  function rs2(opn1:SixteenBitVector) return SixteenBitVector;
  function rs3(opn1:SixteenBitVector) return SixteenBitVector;
  function rs4(opn1:SixteenBitVector) return SixteenBitVector;
end SHIFT;

use work.TYPES.all;
use work.OPR16.all;
use work.SHIFT.all;
```

```
entity AFT is
  port(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9:in SixteenBitVector;
       out1,out2,out3,out4,out5,out6,out7,
       out8,out9,out10:out SixteenBitVector);
end AFT;


architecture Behaviour of AFT is

begin  process

  variable e1, e2, e3, e4, e5, e6, e7, e8, e9, e10,
           e11,e12,e13,e14,e15,e16,e17,e18,e19,e20,
           e21,e22,e23,e24,e25,e26,e27,e28,e29,e30,
           e31,e32,e33,e34,e35,e36,e37,e38,e39,e40,
           e41,e42,e43,e44,e45,e46,e47,e48,e49,e50,
           e51,e52,e53,e54:SixteenBitVector;
  constant OneByTwo   :SixteenBitVector:=X"0008";
  constant OneByFour  :SixteenBitVector:=X"0004";
  constant OneBySix   :SixteenBitVector:=X"0003";
  constant OneByEight:SixteenBitVector:=X"0002";
  constant OneByTen   :SixteenBitVector:=X"0001";
  constant Zero       :SixteenBitVector:=X"0000";

  begin
    e3:=a0-a1;
    e4:=a3-a4;
    e27:=e3+e4;
    e5:=a6-a8;
    e6:=a5-a9;
    e28:=e5-e6;
    e39:=e27+e28;
    out1<=e39*9;    /*    out1:=e39*9;  */
    e1:=a0-a5;
    e2:=a3-a2;
    e26:=e1+e2;
    e38:=e26+e5;
    e49:=e38*6;     /*    e49:=e38*6;   */
    out8<=e49;
    e37:=e1*3;       /*    e37:=e1*3;     */
    e12:=a5-a8;
    e14:=a0-a3;
    e34:=e12-e14;
    e44:=e34*5;      /*    e44:=e34*5;    */
    out7<=e44;
    e50:=e44-e37;
    out9<=e49-e50;
    e32:=e2-e3;
    e8:=a4+a6;
    e43:=e8+e32;
    e13:=a7+a9;
    e33:=e12+e13;
    e48:=e43-e33;
    e53:=e48*10;     /*    e53:=e48*10;   */
    out5:=e53;
    out6<=Zero-e53;
    e52:=a1-a6;
    e11:=e4-e52;
    e10:=a7-a9;
    e31:=e10+e11;
    e41:=e31*6;      /*    e41:=e31*6;    */
    out3<=e41;
    e15:=a3-a8;
    e35:=e15*3;      /*    e35:=e15*3;    */
    e47:=e41+e35;
    out4<=e47+e53;
    e29:=e6-e8;
    e9:=a2+a7;
    e30:=e3+e9;
    e40:=e29-e30;
    out2<=e40*9;     /*     out2:=e40*9; */
    e16:=a1-a4;
    e17:=a6-a9;
    e36:=e16+e17;
    e46:=e36*5;      /*    e46:=e36*5;    */
    out10<=e46;
  end process;
end Behaviour;
```

Appendix C: VHDL description for Elliptic Wave Filter

```
package TYPES is
    type SixteenBitVector is array(15 downto 0) of Bit;
end TYPES;

use work.TYPES.all;
package OPR16 is
    function "+"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
    function "-"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
    function "*"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
    function "/"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
end OPR16;

use work.TYPES.all;
use work.OPR16.all;

package CMP16 is
    function lt(opn1,opn2:SixteenBitVector) return Boolean;
    function le(opn1,opn2:SixteenBitVector) return Boolean;
    function gt(opn1,opn2:SixteenBitVector) return Boolean;
    function ge(opn1,opn2:SixteenBitVector) return Boolean;
    function eq(opn1,opn2:SixteenBitVector) return Boolean;
    function ne(opn1,opn2:SixteenBitVector) return Boolean;
end CMP16;

use work.TYPES.all;
use work.OPR16.all;
use work.CMP16.all;

entity ELLIPTIC is
    port(inpi,sv2i,sv13i,sv18i,sv26i,sv33i,sv38i,sv39i,
         reset,over,rega,regb,regc,regd,rege,regf,
         regg,regh:in SixteenBitVector;
         sv26o,sv39o,sv2o,sv18o,sv38o,sv13o,sv33o:out SixteenBitVector);
end ELLIPTIC;


architecture Behaviour of ELLIPTIC is
begin process
```

```
constant Zero  :SixteenBitVector:=X"0000";
constant One   :SixteenBitVector:=X"0001";
constant Two   :SixteenBitVector:=X"0002";
constant Three :SixteenBitVector:=X"0003";
constant Four  :SixteenBitVector:=X"0004";

variable outpi,inp,sv2,sv13,sv18,sv26,sv33,sv38,sv39:SixteenBitVector;
variable op3,op32,op12,op20,op25,op21,op24,op19,op27,op11,op22,
         op29,op9,op30,op8,op31,op7,op10,op28,op41,op6,op15,op35,
         op40,op4,op16,op36:SixteenBitVector;
begin
op3 := inpi + sv2i;
op12 := op3 + sv13i;
op20 := op12 + sv26i;
op32 := sv33i + sv39i;
op25 := op20 + op32;
op21 := op25 * rega;
op19 := op12 + op21;
op22 := op19 + op25;
op24 := op25 * regb;
op27 := op24 + op32;
sv26 := op22 + op27;
sv26o <= sv26;
op29 := op27 + op32;
op30 := op29 * regd;
op31 := op30 + sv39i;
op11 := op12 + op19;;
op9 := op11 * regc;
op8 := op3 + op9;
op7 := op3 + op8;
op6 := op7 * rege;
op4 := inpi + op6;
sv2 := op4 + op8;
sv2o <= sv2;
op10 := op8 + op19;
op15 := op10 + sv18i;
op16 := op15 * regg;
sv18 := op16 + sv18i;
sv18o <= sv18;
op28 := op27 + op31;
```

```
    op35 := sv38i + op28;
    op36 := op35 * regh;
    sv38 := sv38i + op36;
    sv38o <= sv38;
    op41 := op31 + sv39i;
    outpi := op41 * regf;
    sv39 := op31 + outpi;
    sv39o <= sv39;
    sv13 := op15 + sv18i;
    sv13o <= sv13;
    sv33 := sv38i + op35;
    sv33o <= sv33;
    end process;
end Behaviour;
```

Appendix D: VHDL description for FIR Filter

```
--
-- This is VHDL description of the Finite Impulse Response Filter.
--

package TYPES is
  type SixteenBitVector is array(15 downto 0) of Bit;
end TYPES;

use work.TYPES.all;

package OPR16 is
  function "+"(OPN1,OPN2:SixteenBitVector) return SixteenBitVector;
  function "-"(OPN1,OPN2:SixteenBitVector) return SixteenBitVector;
  function "*"(OPN1,OPN2:SixteenBitVector) return SixteenBitVector;
end OPR16;

use work.TYPES.all;
use work.OPR16.all;

entity FIR is
  port(a1, a2, a3, a4, a5, a6, a7, a8, a9, a10,a11,a12,
       a13,a14,a15,a16,a17,a18,a19,a20,a21,a22,a23,a24:in SixteenBitVector;
       out1 :out SixteenBitVector);
end FIR;

architecture Behaviour of FIR is
begin
  process
  variable e1,e2,e3,e4,e5,e6,e7:SixteenBitVector;
  begin
e1:=(a1+a2)*a17;
e2:=(a3+a4)*a18+e1;
e3:=(a5+a6)*a19+e2;
e4:=(a7+a8)*a20+e3;
e5:=(a9+a10)*a21+e4;
e6:=(a11+a12)*a22+e5;
e7:=(a13+a14)*a23+e6;
out1<=(a15+a16)*a24+e7;
```

```
  end process;
end Behaviour;
```

Appendix E: VHDL description for Robot Arm Controller

```
--
-- This is the portion within the outer loop of a robot arm controller.
-- This was obtained from the C code for the robot arm controller.
--

package TYPES is

  type SixteenBitVector is array(15 downto 0) of Bit;

end TYPES;

use work.TYPES.all;
package OPR16 is

  function "+"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function "-"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function "*"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function less(opn1,opn2:SixteenBitVector) return Boolean;

end OPR16;

use work.TYPES.all;
use work.OPR16.all;
entity ROBOT is
    port(xv1,xv2,uv1,uv2,kv1,kv2,xvh1i,xvh2i,evthresh,T1,km11,km12,km21,
         km22, u11i,u21i,mh11i,mh12i,mh21i,mh22i:in SixteenBitVector;
         xvh1o,xvh2o,mh11o,mh12o,mh21o,mh22o,q1,q2,u11o,
         u21o: out SixteenBitVector);
end ROBOT;


architecture Behaviour of ROBOT is

begin process
    variable tmp1,em1,em2,u10,u20,mh110,mh120,mh210,mh220,
             ev1,ev2: SixteenBitVector;
    constant  TwoFiftySix: SixteenBitVector :=X"0100";
    constant  Zero,Zero1,Zero2,Zero3,Zero4: SixteenBitVector :=X"0000";
```

```
begin

    em1:=xvh1i-xv1;
    if less(em1,Zero) then
        tmp1:=Zero4-em1;
    else
        tmp1:=Zero4+em1;
    end if;

    if less(tmp1,evthresh) then
        em1:=Zero3;
    else
        em1:=Zero3+em1;
    end if;

    em2:=xvh2i-xv2;
    if less(em2,Zero) then
        tmp1:=Zero2-em2;
    else
        tmp1:=Zero2+em2;
    end if;

    if less(tmp1,evthresh) then
      em2:=Zero1;
    else
      em2:=Zero1+em2;
    end if;

    mh110:=km11*em1*u11i+mh11i;
    mh120:=km12*em1*u21i+mh12i;
    mh210:=km21*em2*u11i+mh21i;
    mh220:=km22*em2*u21i+mh22i;

    ev1:=uv1-xv1;
    u10:=kv1*ev1;
    ev2:=uv2-xv2;
    u20:=kv2*ev2;

    q1<=mh110*u10+mh120*u20;
```

```
        q2<=mh210*u10*mh220*u20;

        xvh1o<=u10*T1+xv1;
        xvh2o<=u20*T1+xv2;

        mh11o<=mh110;
        mh12o<=mh120;
        mh21o<=mh210;
        mh22o<=mh220;
        u11o<=u10;
        u21o<=u20;

    end process;
end Behaviour;
```