Resource Sharing of Pipelined Custom Hardware Extension for Energy-efficient Application-specific Instruction Set Processor Design

Hai Lin and Yunsi Fei
Dept. of Electrical & Computer Engineering
University of Connecticut. Storrs, CT 06269
E-mail: {hal06002,yfei}@engr.uconn.edu

***Abstract*— Application-Specific Instruction set Processor (ASIP) has become an increasingly popular platform for embedded systems because of its high performance and flexibility. Energy efficiency is critical for portable and embedded devices, and should be addressed separately from performance consideration. The hardware extension in ASIPs can speed-up program execution, but also incurs area overhead and static energy consumption of the processors. Traditional data path merging techniques reduce circuit overhead by reusing hardware resources for executing multiple custom instructions. However, they introduce structural hazard for custom instructions on extended processors, and hence reduce the performance improvement. In this paper, we introduce a pipelined configurable hardware structure for the hardware extension in ASIPs, so that structural hazards can be remedied. With multiple subgraphs of operations selected for custom hardware realization, we devise a novel operation-to-hardware mapping algorithm based on Integer Linear Programming (ILP) to automatically construct a resource-efficient pipelined configurable hardware extension. We demonstrate that different resource sharing schemes would affect both the hardware overhead and datapath delay of the custom instructions. We analyze the design trade-offs between resource efficiency and performance improvement, and present the design space exploration results.**

I. Introduction

Application-Specific Instruction set Processors (ASIPs) have become a promising design platform for modern embedded systems, satisfying their demanding requirements on performance, cost, power consumption, and turn-around time. With customized instruction set architecture (ISA) and custom hardware extensions tailored for a specific application domain, the performance of an ASIP is improved greatly over general-purpose processors for the specific applications.

While the traditional ASIP design focuses on performance improvement [1], [2], improving resource efficiency of the custom hardware extensions has been an effective method to reduce the die area overhead and chip cost. With the proliferation of battery powered electronic devices, energy efficiency has become an imperative design metric for modern embedded processors. There have been a lot of researches in ASIP design that tackle the energy efficiency issue, either by optimizing individual processor components, e.g., cache, register file and instruction fetch stage [3], [4], [5], or by power-managing the whole processor [6], [7]. However, most of them target dynamic energy consumption and do not consider the effect of static energy specifically.

Leakage energy consumption has become an important issue in modern electronic circuits. When the process technology shrinks below 65 nm, the leakage power increases to be comparable to dynamic power [8]. Although some recent technologies, like the"high-k dielectric technology" [9] adopted in top-of-the-line processes, can effectively suppress the leakage current, static energy remains an important component of the total energy consumption. Custom hardware extensions in ASIP design are able to reduce the execution time and dynamic energy. However, the static energy consumption caused by these extensions may greatly offset the total energy reduction rate. Improving the resource efficiency, i.e., lowering the hardware overhead, of the custom hardware extensions will reduce the static energy consumption in energy-efficient ASIP design.

When implementing resource-efficient custom logic for ASIPs, traditional resource sharing techniques, such as data path merging and reconfigurable data-path synthesis [10], [11], [12], [13], [14], [15], are widely used to generate a configurable hardware extension that can share the functional units among multiple custom instructions. However, such resource sharing will introduce potential structural hazard for executing custom instructions, because at any time, only one custom instruction can be executed on the shared functional units. It will result in less performance improvement.

In this paper, we propose a pipelined configurable custom hardware structure to remedy the structural hazards and meanwhile achieve promising energy efficiency, reducing the static energy overhead caused by custom hardware. With the proposed configurable custom functional units (CFUs), we formulate our hardware overhead minimization problem into an ILP form, and develop a novel algorithm for optimal resource sharing among multiple candidate custom instructions. We analyze the trade-off between resource efficiency and performance improvement.

The rest of the paper is organized as follows. In Section II, we discuss the motivation for a pipelined configurable CFU, and briefly describe the proposed CFU structure. In Section III, we formulate the resource sharing problem into an ILP problem and present our algorithm. We then demonstrate the experimental setup and results in Section IV. Trade-offs between resource sharing and performance improvement in ASIPs are also shown, and the design space is analyzed. Following that, we conclude our work in Section V.

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

II. Exploring Resource Sharing of Hardware Extensions in ASIP Design

In this section, we discuss the unique features of resource sharing of hardware extensions in ASIP design, and propose our pipelined configurable CFU structure. We assume that the widely used technique for instruction set extension, operation fusion [1], is adopted. Selected groups of operations in the dataflow graphs (DFGs) of a program's basic blocks are fused into single complex operations. Both the execution time and dynamic energy consumption of the application can be reduced.

A. Resource Sharing of Hardware Extensions for Multiple Custom Instructions

Fig. 1 illustrates the traditional resource sharing process among multiple custom instructions. Each custom instruction, selected by a custom instruction synthesis flow, performs a sub-graph of operations. To reduce the area overhead, the custom hardware is shared by the two instructions, and includes a superset of the functional units for different operation sub-graphs. For example, sub-graph 1 needs 2 adders and 1 multiplier, and sub-graph 2 needs 3 adders, and the configurable custom functional unit contains 3 adders and 1 multiplier. Configurable interconnections, i.e., MUXes, are inserted so that different sets of functional modules are used for executing different custom instructions. For this kind of custom hardware synthesis, traditional resource sharing techniques such as multiple graphs and paths merging [11], [12], [13] are applied. The work in [11] formulated such datapath merging problem into a maximum compatible clique searching problem, and evaluated the performance and complexity of several typical heuristic solutions. In [12], the problem is converted into a substring matching problem and a heuristic is developed to find reasonable solutions. The authors also analyzed the effect of resource sharing on critical path delay of each data path. The trade-off between area overhead and critical path delay is demonstrated by experimental results. However, these techniques have a common assumption that custom instructions are executed on the configurable custom hardware at different time, i.e., the functional units are used in a time-multiplexing fashion. Our resource sharing technique differs from the previous work in that it considers the specific features of executing custom instructions on ASIPs. Fig. 2 illustrates the partial pipeline of an out-of-order extended processor architecture. In an out-of-order processor, instructions are issued and executed whenever the resources for executing them are ready. Without resource sharing in the extended hardware, a custom functional unit will be generated for each custom instruction and exists in parallel to the baseline functional units. For example, two independent custom instructions, e.g., *ci*1 and *ci*2 as shown in Fig. 2, can be issued from the instruction queue and executed simultaneously in different custom functional unit (CFU1 and CFU2), with the control flow marked in blue arrowed lines in the figure. The instruction level parallelism of the program is well maintained. However, with resource sharing, the two custom functional unit are merged into one to save area, shown as the shaded module (CFU) in the dotted component in Fig. 2. Any time only one custom instruction can be issued to the merged CFU, as shown in the red arrowed line. For example, when instruction *ci*1 is in execution, instruction *ci*2 has to wait for the availability of hardware resource, i.e., the traditional resource sharing method causes structural hazard for custom instructions and the performance improvement will be reduced. When these custom instructions are multi-cycle instructions, such performance degradation is more drastic, even in single-issue processors.

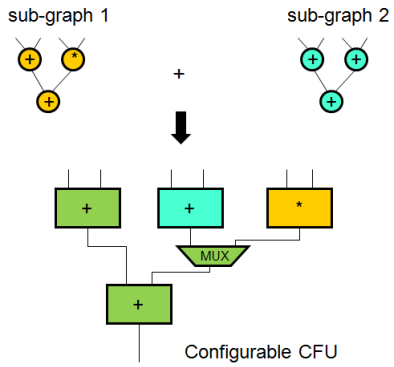

Fig. 1. A resource sharing example of configurable custom functional unit

To remedy such structural hazard so as to improve the performance for an ASIP, we introduce a novel pipelined configurable hardware structure and develop our resource sharing algorithm for the CFUs.

B. Structure of Configurable CFU

Fig. 3 (a) illustrates the architecture of an example configurable CFU used in our flow. Functional modules are arranged in rows and connected in a feed-forwarding manner, i.e., signals transfer from the outputs of components in upper-level rows to the inputs of components in the subsequent rows. MUXes are inserted to allow configuration of interconnections, and control signals for the MUXes are generated by the custom instruction decoding logic. Thus, different custom instructions can change the connection of functional modules in the configurable CFU. For a selected sub-graph, as shown in Fig. 3 (b), its operations are mapped onto the functional components with color in the configurable CFU, shown in Fig. 3 (a). If the critical path delay of the datapath exceeds one clock cycle of the processor, registers are inserted into the configurable data path, and hence custom instructions can be fed into the shared hardware extension in a pipelined sequence. With this pipelined structure, at one execution cycle, multiple multi-cycle custom instructions can possibly share the resource without conflict, because different custom instructions can use functional units at different pipeline stages. Considerable area reduction can still be achieved through our novel resource sharing algorithm. Fig. 4 shows the cycle-accurate execution under the three different resource sharing scenarios in a single-issue out-of-order processor. We assume *ci*1 takes four cycles for

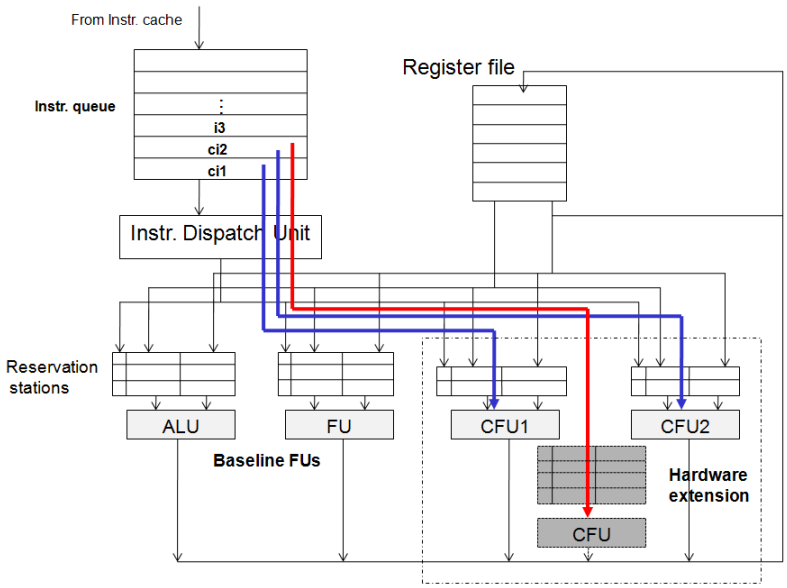

Fig. 2. Structural hazard caused by resource sharing in an out-of-order extended processor

execution and *ci*2 takes two cycles. When no resource sharing, different instructions can execute in the same cycle without conflict. In the traditional resource sharing without pipelining, the processor pipeline has to be stalled for the second instruction due to structural hazard. With our resource sharing with pipelining, the performance is as good as the first case, and the hardware overhead is reduced.

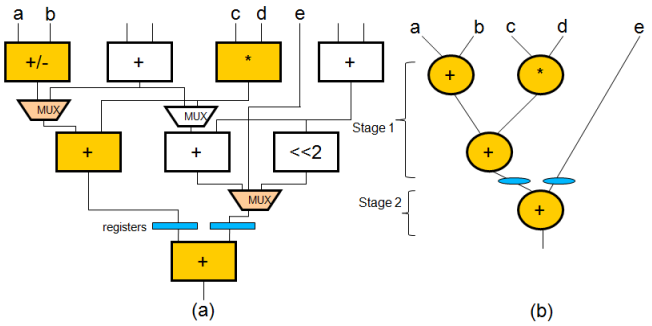

Fig. 3. Mapping a sub-graph onto a configurable custom functional unit

The configurable CFU architecture proposed is similar to the *Configurable Compute Accelerator (CCA)* presented in [16]. However, different to their design, we introduce pipelining within the shared hardware. Moreover, unlike the previous work which pre-defines a CCA structure and develops an algorithm to identify and map appropriate DFGs onto it, our approach automatically generates the optimal configurable custom hardware for the target application.

In the following sections, we assume that the sub-graphs of operations for custom instructions have been selected by certain synthesis tools, like the one in [17], and focus on our resource sharing algorithm for the proposed hardware architecture.

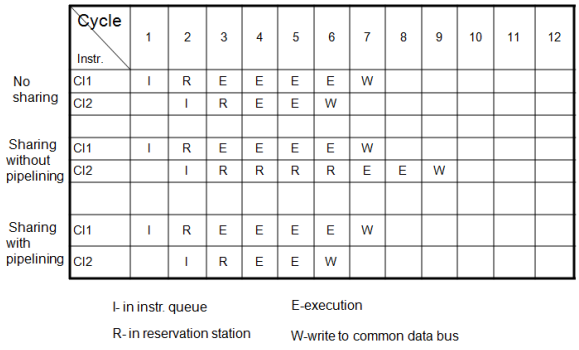

Fig. 4. Cycle-accurate execution of the custom instructions under three resource sharing scenarios

III. Solution to the Resource Sharing Problem

A new resource sharing problem is raised in our custom hardware generation process for better performance. It is based on two assumptions: a set of sub-graphs of operations have been selected for custom instruction implementation; and dataflow sub-graphs can be pipelined if they need multiple cycles to finish execution. The problem then becomes how to schedule the operations into different execution cycles in the pipelined data path and merge the operators into configurable hardware. The functional modules in a pipeline stage can be shared among different data paths for custom instructions. The goal is to maximize the resource sharing so as to reduce the area overhead and static energy consumption of the configurable custom hardware.

Although the resource sharing idea has been widely used in both high-level synthesis (HLS) and previous custom hardware synthesis in ASIP design, our resource sharing problem differs distinctly from both of them. HLS targets

resource sharing within one data path and it shares resource between different cycles. In our problem, we do not allow operations in the same data path but at different cycles to share resource. Previous research for resource-efficient custom hardware synthesis in ASIPs, like [11], [13], have not considered the potential structural hazard of multiple data path merging, and the shared resources are not pipelined. Although this can provide large flexibility of sharing, the performance improvement can be degraded. Fig. 5 depicts the three different types of resource sharing in three rows, i.e., in HLS, traditional resource sharing in custom hardware synthesis, and our proposed resource sharing in pipelined structure. For the figures on the left, each dotted arrow connects two operations that share the same resource, each solid bar indicates the places where registers are inserted. The figures on the right are the abstract structure diagrams of dataflow.

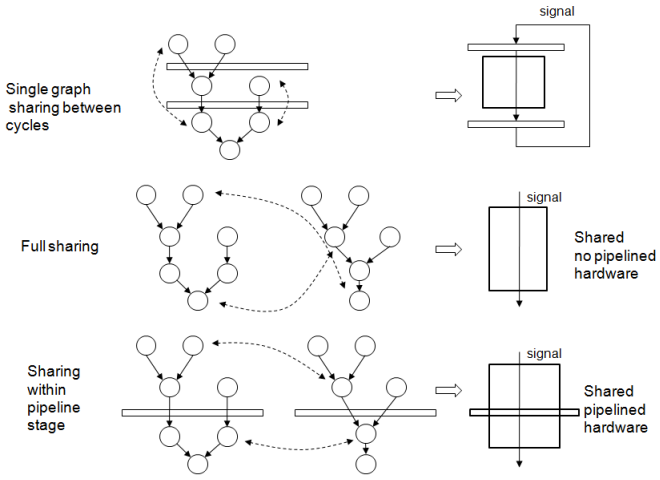

Fig. 5. Different types of resource sharing

A. Problem Definition

With further characterization, we next show that our problem is transformed into a problem of operation scheduling and mapping onto the custom hardware, with the hardware co-generated. Fig. 6 shows such a process, where two sub-graphs of operations, g_1 and g_2, are presented in directed-acyclic-graphs (DAGs) with data dependency between operations shown on the connecting edges. We assume that operations of the gray nodes have the same operation type, i.e., candidates sharing the same operator in the custom hardware. Operations in these two graphs can be scheduled to different virtual stages while maintaining their data dependency, for example, in an as-soon-as-possible (ASAP) or as-late-as-possible (ALAP) manner. When mapping the operations onto the custom hardware, these virtual stages are hence mapped to the CFU's hardware pipeline stages. A hardware stage can cover several consecutive virtual stages, depending on the latency in each virtual stage. The edges crossing the lines of hardware pipeline stages represent positions for pipeline registers. For a scheduling scheme of multiple DAGs, we estimate the data-path delay and group virtual stages to pipeline stages. Only operations of different data paths that are of the same type and at the same virtual stage are allowed to share functional modules, e.g., the two nodes in gray at virtual stage 4 in Fig. 6. Finally, the needed functional components at each pipeline stages are sum up for hardware overhead estimation, and the interconnection components like the MUXes and pipeline registers are also estimated. Different scheduling of DAGs will affect both the number of pipeline stages for each data path and the overall hardware overhead, i.e., resource efficiency. As illustrated in Fig. 7, for the first scheduling plan for sub-graph 2, custom instruction 2 takes 1 cycle to execute, while custom instruction 1 takes 2 cycles, and one extra adder is needed for pipeline stage 1 in addition to what have been provided for sub-graph 1. For the second scheduling plan, both the two custom instructions take 2 cycles instead, but no extra adder is needed for sub-graph 2, because it can share the adder for sub-graph 1 at virtual stage 4 (pipeline stage 2). These two scheduling schemes demonstrate possible trade-off between execution delay and custom hardware area overhead. Among all the scheduling plans for the DAGs, we select the best one that can reduce the hardware overhead most and meanwhile satisfy the performance requirement.

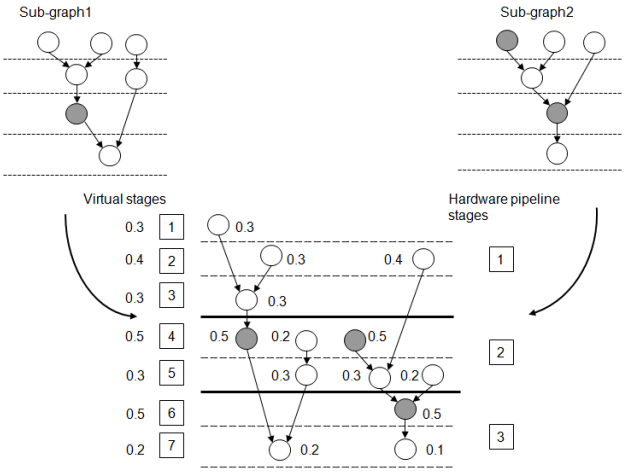

Fig. 6. One possible operation scheduling and resource sharing plan

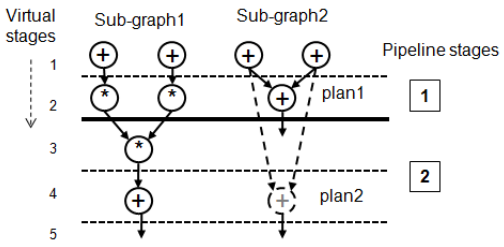

Fig. 7. Trade-off between custom hardware overhead and execution cycles

B. Integer Linear Programming Solution

To solve the resource efficiency problem, we develop a novel algorithm for operation scheduling and multiple DAG resource sharing based on Integer Linear Programming (ILP). In contrast to the previous iterative algorithms [10], [12], we take multiple DAGs in at the same time and find an optimal CFU template to implement all of them based on one-time ILP solving process. Both the area overhead and delay should be estimated in the objective function to guide the exploration for optimal solutions. We present the entire ILP problem formulation as follows.

Primary Variable definition: For each operation in the DAGs, we define a set of binary variable $\{s_{i,l}\}$, where i is the index of the operation (unique for each operation in all the DAGs), and l is the index of virtual stages. If operation i is scheduled in virtual stage l, $s_{i,l}$ is assigned 1, otherwise 0. Clearly, for an operation i, only one of these variables is assigned 1.

Parameter definition: For each operation i, a set of $\{type_{i,k}\}$ is defined, where k is the index of operation types. $type_{i,k} = 1$ indicates the operation belongs to type k. Similarly, for each operation i, a set of $\{group_{i,j}\}$ is defined, where j is the index of DAGs. $group_{i,j} = 1$ indicates the operation belongs to DAG j. The value of these parameters are determined by the given DAGs.

Assistant Variable definition: To evaluate the delay and execution cycle of each DAG, we add a set of assistant variables. C_l represents the total delay for virtual stage l. C_l equals to the largest delay of the functional units that are in the same stage l. AC_l is the accumulated delay from the primary input to stage l, i.e., AC_l is the summation of C_t, $t = 1,...,l$. If AC_l exceeds $n * T_{cycle}$ and AC_{l-1} is within $n * T_{cycle}$, virtual stage l will be put to pipeline stage $n+1$. T_{cycle} is the clock cycle for the processor.

Constraints: There are several rules for the resource sharing that should be reflected by the constraints in ILP. First of all, when scheduling operations to different virtual stages and implementing them by functional components, the logic dependency within each DAG should be maintained. In other words, if there is an edge connecting two operation nodes in a DAG, the source node of the edge (parent node) should be assigned to a virtual stage earlier than the virtual stage where the destination node (child node) is assigned to. With the variables definition described above, this constraint is presented in Equation (1), where node i is the parent node of node j:

$$\sum_l l * s_{i,l} - \sum_l l * s_{j,l} < 0 \tag{1}$$

$$\sum_l s_{i,l} = 1 \tag{2}$$

Obviously, one operation can only be assigned to one virtual stage, as represented in Equation (2).

Objective Function: For this problem, we want to find the optimal operation scheduling for each DAG, so that: 1. when implementing these operations on hardware stages in the CFU, as many operations as possible can share the functional components in the CFU and hence the hardware cost for the CFU is minimized; 2. the execution cycles of each DAG (representing different custom instructions) will not be increased greatly with resource sharing. This reflects the trade-off between resource efficiency and performance for each custom instruction. We put both the area and delay estimation into an unified objective function.

For each virtual stage, the number of functional component instances needed for each operation type should be equal to the largest number of this type of operations assigned to this stage among all the DAGs. This is a $MAX()$ function. The total hardware overhead is a summation of these $MAX()$ functions for each stage and each type, multiplied by the unit area of each type. Since $MAX()$ function is not a linear function, we add additional variables and constraints to convert this objective function into a linear form.

$$M_{j,k,l} = \sum_i group_{i,j} * type_{i,k} * s_{i,l} \tag{3}$$

$$X_{k,l} - M_{j,k,l} \geq 0 \ (for\ any\ j) \tag{4}$$

Where i, j, k, l are the operation index, DAG index, operation type index and virtual stage index respectively. $M_{j,k,l}$ in Equation (3) represents the number of operations of type k in DAG j that are assigned to virtual stage l. $X_{k,l}$ denotes the number of operator of type k at virtual stage l. The $MAX()$ function is hence converted to the set of constraints presented in Equation (4).

To estimate the execution cycle delay information for each DAG, a set of integer variable K_j represents the cycles needed for the jth DAG. Equation (5) presents the estimation of K_j. K_j should be the ceiling of AC_l, where virtual stage l is the last stage for DAG j. AC_l is described before in "Assistant Variable definition".

$$\begin{gathered} K_j - AC_l * s_{i,l} \geq 0 \\ (for\ any\ i\ that\ satisfies\ group_{i,j} = 1) \end{gathered} \tag{5}$$

Note that $AC_l * s_{i,l}$ is not a linear representation and should be linearized for ILP. We take the approach in [18], for $C = B * A$, where A is a binary variable and M is an upper bound of B, it can be linearized as follows:

$$\begin{aligned} 0 \leq C \leq B \\ C \leq M * A \\ C \geq B - M(1-A) \end{aligned} \tag{6}$$

The final objective function is presented linearly in Equation (7), where A_k is the unit hardware area of operator type k. We use coefficient α to adjust the consideration of resource sharing vs. delay. Both resource overhead and delay are normalized. $AREA$ is the summation of the total area cost for each DAG if implemented separately. T_j is the number

of cycles needed for DAG j if implemented separately with the operations in an ASAP scheduling. The resource sharing problem is to find the best set of $s_{i,l}$ values so as to minimize the object function.

$$min : \alpha(\sum_{k}\sum_{l} A_k * X_{k,l})/AREA + (1-\alpha)(\sum_{j}(K_j/T_j)) \quad (7)$$

For all the above equations, the ranges of i, j, k are determined by the actual number of operations in all the DAGs, operations types, and number of DAGs. The range of l, i.e., the number of possible virtual stages in the CFU is determined by the total number of operations in the DAGs, considering the extreme case when each operation is assigned to a separate virtual stage, as shown in Fig. 8.

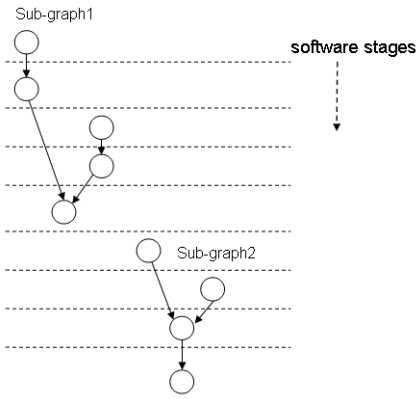

Fig. 8. Possible number of virtual stages

With all these equations and constraints presented, the resource sharing problem is now formulated into an ILP problem. We revisit Fig. 6 which shows a possible scheduling plan. The delay of each operation is marked by the node, normalized to the processor's clock cycle. The delay of each virtual stage is estimated and annotated on the leftmost. Based on this, we assign pipeline hardware stages to virtual stages, and the pipeline stages are marked on the right. We explore all the possible scheduling plans and find the optimal solution with the objective function minimized. With a scheduling plan, an arbitrary functional component sharing scheme between operations of the same type, in the same virtual stages and from different graphs can be determined, and the corresponding interconnections, MUXes and control signals are generated for the configurable CFU. Note that an optimal functional component sharing scheme can further reduce the number of MUXes, however here we omit the difference considering the area overhead of MUXes is much smaller than functional units. With the scheduling plan and pipeline information, the actual number of registers for pipelining the CFU is also determined. We add the area cost of MUXes and pipeline registers into the final area estimation as well.

The ILP problem scale increases as the total number of operation nodes in DAGs increases. With the above ILP modeling, the total number of variables is O(N^2), where N is the number of operations.

IV. Experimental Results

In this section, we present the experimental setup and show the resource sharing results using our proposed pipelined CFU structure and ILP algorithm. First we implement an ASIP custom instruction synthesis flow. We extract a set of sub-graphs from each testbench for custom instruction implementation. We take the SUIF and MachSUIF framework [19], [20], and develop our own passes for front-end compiling, program profiling, and DFG extraction. Based on the profiling information, we implement a DFG exploration and custom instruction selection algorithm similar to the one proposed in [17]. The sub-graphs for custom instructions are selected to best speed up the application with the constraints on register file I/O ports. The timing and area information of different types of functional units in the CFU is estimated based on the logic synthesis using Design_Compiler from Synopsys Inc. [21] with the 0.13 um process library, and we set the processor clock frequency to be 500MHz in the experiment. After the candidate sub-graphs are selected, they are taken in as the DAGs in our ILP model generation program. Our program utilizes LPsolver APIs [22] to solve the ILP model and generate the configurable CFU.

We randomly selected and tested a set of benchmark applications from Mibench [23], which is a commercially representative embedded benchmark suite, to evaluate the effect of resource sharing using the proposed algorithm. Table I presents the statistic information of the selected sub-graphs from each testbench, where the input/output ports constraint for the selected sub-graphs is set to 4/2 as a typical example. Column 2 gives the number of sub-graphs chosen for each testbench, and Column 3 the total number of operations for these sub-graphs, which give an idea about the input size to the ILP model.

TABLE I
Sub-graph Statistics

Testbenches	# of Sub-graphs	Total # of operations
adpcm_c	6	41
adpcm_d	6	33
blowfish_d	6	39
blowfish_e	5	32
basicmath	5	36
bitcount	4	27
dijkstra	6	32
partricia	6	31
stringsearch	4	30
sha	4	35
qsort	4	24
CRC	6	33
jpeg	3	24

In the experiments, we focus on reducing the area overhead first and hence set the coefficient α in the object function of Equation (7) to be 1. Table II gives the statistics for the configurable CFU generated by our ILP models for different testbenches. Column 2 gives the total number of pipeline stages. Column 3 - 7 show the number of functional components of each type. Column 8 gives the total area

overhead including MUXes added in the configuration logic. Column 9 shows the original circuit overhead if implementing all the custom instructions' data paths separately. Column 10 gives the area reduction rate. The average area reduction rate achieved in our experiment for all the testbenches is 43.9%, with the maximum reduction rate reaching 60.8%. This is similar to the reduction rate achieved in the previous technique [12].

When solving such ILP problems, we use the LPsolver on a Intel(R) Xeon(TM) CPU 2.80GHz, 6G Memory computer. Table III presents the statistics of solving the ILP model for each testbench. Column 2 and Column 3 show the number of variables and constraints within each ILP model. Column 4 shows the time consumed to solve the ILP problems. As we can see, the solving time varies a lot between different testbenches with similar scale, which demonstrates that the solving time is dependent on the specific DAGs, not only the number of variables and constraints. We set the maximum solving time to be 1800 seconds to reduce the time cost, and for most of the testbenches within such a solving time, the difference between the optimal results (estimated by the LPsolver with relaxed constraints) and the results found so far is within 5% range. The results can be improved further through extending the solving time. To further reduce the solving time of this problem, other heuristic approaches can also be considered.

TABLE III
ILP MODEL STATISTICS

Testbenches	# of variables	# of constraints	Time elapsed(s)
adpcm_c	3656	15282	1800
adpcm_d	2679	10484	1800
blowfish_d	3399	12628	1800
blowfish_e	2345	8033	1246
basicmath	3029	9857	1800
bitcount	2029	5885	143
dijkstra	2566	10134	1058
partricia	2455	9786	1106
stringsearch	2344	6623	1800
sha	2909	7904	1800
qsort	1732	5153	183
CRC	2679	10484	1800
jpeg	1731	4028	1800

As described in Section III, there is a possible trade-off between scheduling for achieving more resource sharing and for reducing the execution cycles for each custom instruction in a pipelined architecture. Next we change the coefficient α to explore the design space. We use the same DAGs from Table I. The coefficient α is set to 1, 0.5, 0 to represent three different cases, i.e., considering the area reduction only, area and delay trade-off, and minimizing the cycle delay for each custom instruction. We name the three cases as "Area", "Area-Timing" and "Timing" respectively. Fig. 9 and 10 show the design trade-off. In Fig. 9, the area of the custom hardware generated is normalized to the total area overhead when implementing the custom instructions on separate hardware. In Fig. 10, we estimate the execution cycles for each custom instruction executed on the generated hardware, and compare them to the minimum cycle number for each custom instruction when they are executed on separate hardware and all the operations are scheduled in ASAP manner. Comparing the data for the same testbenches in the two figures, we see that for most testbenches the trade-off of area reduction and cycle delay is explicit, i.e., the resource sharing plan with the largest area reduction normally has the longest average cycle delay. With more pipeline stages, the functional resources can be shared more efficiently between different DFGs. We also observe that when balancing the two design objectives by setting α between 0 and 1, the cycle number can be reduced greatly without increasing the area overhead much, e.g., in *qsort*, *dijkstra* and *adpcm_d*. As an example, we show the three different CFUs obtained for the same testbench - *adpcm_d* with different design objectives in Fig. 11. We omit the interconnections between functional units for simplicity. Blocks of different shape represent different type of functional units and the solid horizontal lines divide the functional units into different pipeline stages. We can see that the CFU generated in "Area" case contains fewest functional units however more pipeline stages than the CFUs generated in the other two cases. The CFU in "Timing" case has the least pipeline stages however the most number of functional units. The CFU generated in "Area-Timing" case achieves balance between the two design objectives. It has less area overhead than the CFU in "Timing" case, and also less pipeline stages than the CFU in "Area" case.

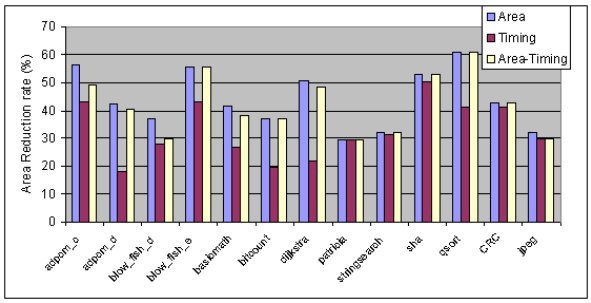

Fig. 9. Area reduction by resource sharing for different testbenches

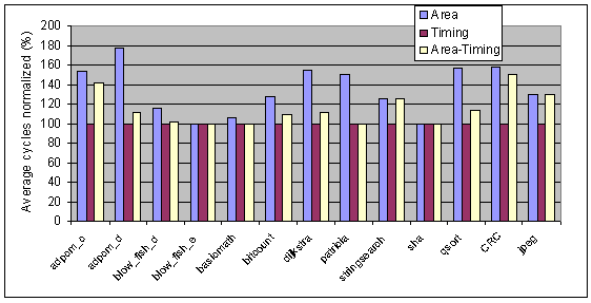

Fig. 10. Cycle number increase by resource sharing for different testbenches

V. CONCLUSIONS

In this paper, we consider the static energy in ASIP design, and address the static energy overhead caused by custom hardware added in ASIPs. To reduce the custom hardware extension for both static energy consumption and die area reduction, this paper analyzes a new resource sharing problem in the hardware extension generated for the

TABLE II

GENERATED CONFIGURABLE CFU

Testbenches	Pipeline stages	Adder/Sub.	Shifter	Logic	Compare	Multiplier/Divider	Total area ($10^3 gates$)	Orig. area ($10^3 gates$)	Area reduction (%)
adpcm_c	5	11	2	1	2	1	71.0	163.1	56.4
adpcm_d	4	12	0	3	2	0	49.2	85.2	42.2
blowfish_d	4	16	0	1	2	0	78.6	124.6	36.9
blowfish_e	3	11	0	1	2	0	37.8	85.4	55.7
basicmath	7	16	0	0	0	4	73.7	125.8	41.5
bitcount	5	14	1	1	0	0	73.9	117.3	37.0
dijkstra	3	10	0	1	2	0	37.3	75.4	50.5
partricia	3	12	1	1	0	0	60.4	85.6	29.4
stringsearch	4	14	0	1	0	0	50.3	74.3	32.3
sha	4	13	2	3	0	0	41.2	86.9	52.6
qsort	3	10	0	0	1	0	45.7	116.6	60.8
CRC	6	12	0	3	0	1	58.6	102.0	42.6
jpeg	4	13	1	1	0	0	48.3	70.2	32.0
average									43.9

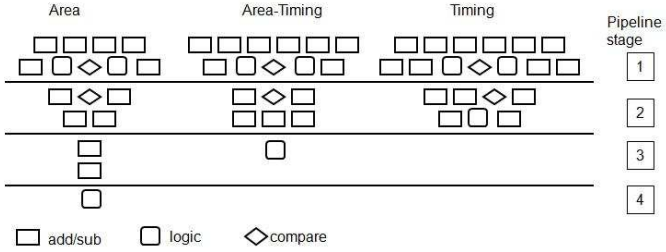

Fig. 11. Different CFUs generated for testbench - *adpcm_d*

custom instructions in ASIPs. We show that the resource sharing problem in ASIPs is different from previous HLS and data path merging problem. We propose a pipelined custom hardware structure and develop an ILP algorithm to explore the design space for an optimal resource sharing solution. The trade-off between execution cycle of custom instructions and area of the custom hardware is discussed, and demonstrated with experimental results.

REFERENCES

[1] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2003, pp. 137–147.

[2] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Custom-instruction synthesis for extensible processor platform," *IEEE Trans. Computer-Aided Design of Integrated Circuits*, vol. 23, no. 2, pp. 216–228, Feb. 2004.

[3] R. A. Ravindran, "Hardware/software techniques for memory power optimizations in embedded processors," *Doctoral Dissertation, Computer Science and Engineering, University of Michigan*, 2007.

[4] R. Ravindran, M. Chu, and S. Mahlke, "Compiler-managed partitioned data caches for low power," in *Proc. of LCTES Conf.*, 2007, pp. 237–247.

[5] H. Lin and Y. Fei, "Harnessing horizontal and vertical parallelism of programs to improve system overall efficiency," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2008, pp. 748–763.

[6] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha, "A hybrid energy estimation technique for extensible processors," *IEEE Trans. Computer-Aided Design of Integrated Circuits*, vol. 23, no. 5, pp. 652–664, May. 2004.

[7] J.-E. Lee, K. Y. Choi, and N. D. Dutt, "Energy-efficient instruction set synthesis for application-specific processors," in *Proc. Int. Symp. Low Power Electronics & Design*, 2003, pp. 330–333.

[8] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *IEEE Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003.

[9] "Intel 45nm high-k metal gate silicon technology," [http://www.intel.com/technology/-architecture-silicon/45nm-core2/-index.htm].

[10] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction sets synthesis for reconfigurable system-on-chip designs," in *Proc. Design Automation Conf.*, 2004, pp. 395–400.

[11] C. C. de Souza, A. M. Lima, G. Araujo, and N. Moreano, "The datapath merging problem in reconfigurable systems: complexity, dual bounds, and heuristic evaluation," *ACM Journal of Experimental Algorithms*, 2005.

[12] M. Zuluaga and N. Topham, "Resource sharing in custom instruction set extensions," in *Proc. IEEE Symp. on Application Specific Processors*, June 2008, pp. 7–13.

[13] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Trans. Computer-Aided Design of Integrated Circuits*, vol. 24, no. 7, pp. 969–980, Jul. 2005.

[14] A. van der Werf, M. J. H. Peek, E. H. L. Aarts, J. L. van Meerbergen, P. E. R. Lippens, and W. F. J. Verhaegh, "Area optimization of multi-functional processing units," in *Proc. Int. Conf. Computer-Aided Design*, 1992, pp. 292–299.

[15] Z. Huang and S. Malik, "Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks," in *Proc. Design Automation & Test Europe Conf.*, 2001, pp. 735–740.

[16] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," in *Proc. Int. Symp. Computer Architecture*, 2005, pp. 272–283.

[17] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. Design Automation Conf.*, June 2003, pp. 256–261.

[18] Y. Wang, H. Lin, H. Yang, R. Luo, and H. Wang, "Simultaneous fine-grain sleep transistor placement and sizing for leakage optimization," in *Proc. IEEE Int. Symp. on Quality Electronic Design*, March 2006, pp. 723–728.

[19] "SUIF Compiler System," [http://suif.stanford.edu/suif/suif2/].

[20] "MACHINE SUIF," [http://www.eecs.harvard.edu/hube/software/software.html].

[21] "Synopsys Design Compiler, Synopsys Inc." [http://www.synopsys.com].

[22] "LPsolver, Linear Programming Solver," [http://lpsolve.sourceforge.net/5.5/].

[23] "MiBench," [http://www.eecs.umich.edu/-mibench/].