Generating ASIPs with Reduced Number of Connections to the Register-file

Yosi Ben Asher | Irina Lipov | Vladislav Tartakovsky | Dror Tiv
CS. University of Haifa, Israel | irinalipov@gmail.com | vlad14176@gmail.com | dror.tiv@gmail.com

Abstract—We propose automatic synthesis of application specific instruction set processors (ASIPs). We use pipeline execution of multi-op machine-instructions, e.g., *(reg1*reg2) = *(reg3) + *(reg4) (C-syntax) an instruction with three memory pipeline stages and two arithmetic stages. The problem is, for a given set of loops, to find a pipeline configuration and a multi-op ISA that maximizes the IPC (instructions per cycle) while minimizing the cost of interconnections to the register-file of the resulting CPU. The algorithm is based on finding an efficient cover of a large graph by a small set of convex subgraphs g's that are consistent with a given set of pipeline units. Unlike previous works, g's are not synthesized to circuits that are executed in a co-processor mode but rather both g's and the rest of the program are executed by the same set of multiop pipeline units. In this way we eliminate the overhead associated with the co-processor mode of regular ASIPs but maintain high values of IPC of these ASIPs. Once the pipeline configuration and the cover \( g_1 \cup \ldots \cup g_n = G \) has been computed the Verilog RTL of the corresponding CPU (extended with branch instructions) is generated and synthesized to FPGA. The results show that, for a set of selected kernels, the resulting ASIP (called Ocpu) obtains higher IPC values compare to an equivalent compilation to an ARM cpu while obtaining similar clock frequencies.

I. INTRODUCTION

An application-specific instruction-set processor (ASIP) is a cpu-component used in system-on-a-chip design. The instruction set of an ASIP is tailored to benefit a specific application. This specialization of the core provides a tradeoff between the flexibility of a general purpose CPUs and direct implementation as circuits using a highlevel synthesis compiler or manual hardware design. Thus techniques for the automated ASIP generation have received much attention in recent years. In this work we target ASIPs that are used as parts of a system-on-chip (SOC) wherein the ASIP executes a single program acting as an accelerator of the complete SOC application. As such the ASIP executes a single program that interact with an on-chip memory and do not include features related to I/O and Operating system. The ASIP thus access a set of on-chip SRAM memory banks or multiport SRAM memories. Typically ASIPs use a compiler to obtain a set of data dependency graphs (DDGs) of code segments from a program and synthesize parts of them as circuits. These sub-graphs/circuits form new instructions (denoted by \( g_i \)) of the ASIP speeding up the execution. The main problem involved with automated ASIP generation is (see survey [12]) how to select the most profitable sub-graphs \( g_i \)'s from the Data Dependency Graph (DDG) of the program. The main considerations include: high number of occurrences in the DDG (accounting execution frequency via unrolling), available hardware resources, number of inputs and outputs, area, power and latency. Several techniques were proposed to reduce the high complexity of this instruction selection problem including: integer linear programming (ILP), selective clustering and branch-and-bound search techniques. Thus, the main algorithmic problem of ASIP generation is generating or enumerating all suitable sub-graphs of a given DAG \( G \) as is done in [27] describing a recursive technique to construct all possible \( g_i \) of \( G \).

However, this approach implies additional ports and connections needed to connect the \( g_i \)'s ops to the register-file increasing wire lengths and reducing the execution time. In order to overcome this we extend the approach of selecting profitable subgraphs \( (g_i) \) to that of finding a full cover of \( G \) by a set of subgraphs \( g_1 \cup \ldots \cup g_n = G \). The \( g_i \) are not executed by a set of circuits but rather by a set of multi-op pipeline units that form the core of the resulting ASIP. Thus by covering the whole of \( G \)'s nodes by \( g_i \) we can synthesize an ASIP that can harvest the potential of using multi-op instructions that are specially adapted to the structure of \( G \) without paying the extra cost associated with synthesizing sub-graphs of \( G \) into circuits. The problem of selecting profitable sub-graphs has thus been replaced by finding a configuration of \( p \) pipeline units of \( k \) stages each for which a graph cover \( g_1 \cup \ldots \cup g_n = G \) exits such that:

- Each \( g_i \) can be executed by a some pipeline unit.
- The product of the resulting execution time of this cover and \( p^2 \) is minimized. This is because the complexity of the interconnections between the pipeline units and the register-file is proportional to \( p^2 \) as each pipeline unit access the register file in its first stage and update it in its last stage.

Once the pipeline configuration and the cover \( g_1 \cup \ldots \cup g_n = G \) has been computed the Verilog RTL of the corresponding CPU (extended with branch instructions) is generated and synthesized to FPGA. The results show that, for a set of selected kernels, the resulting ASIP (called Ocpu) obtains higher IPC values compare to an equivalent compilation to an ARM cpu while obtaining similar clock frequencies.

The main contributions of this work are:

- Extending the current technique for ASIP generation to include true selection of cpu architectures by introducing the architectural space of possible pipeline configurations.
- Selecting ASIPs based not only on the execution time but also on the cost of interconnections.
- Formalizing the CPU generation as a graph cover problem (compare to selecting profitable sub-graphs used in previ-
ous works on ASIP generation).

- Simplifies the resulting ASIP architecture since the hardware side of the $g_i$’s circuits has been replaced by pipeline execution.

II. THE SELECTION PROBLEM

In this work we focus on adapted pipeline structure forming the following coverage problem:

- Let $G$ be a directed acyclic graph (DAG) with $n$ nodes where each node is either a binary arithmetic operation $’+’, ’/’, ’*’, ’<’, AND, . . .$, a single load/store operation $’L’$, a store or load operation $’S’$ with two inputs or a MUX with three inputs.

- We consider two types of pipeline stages: ’$E’$ capable of executing one arithmetic operation such as $max, ’+’, ’/’, ’*’, ’...$, and ’$M’$ for executing memory references (load/store operations).

- A pipeline unit $P_j$ is specified by a sequence of $k$ stages (forming a chain or a directed path), e.g. $P_2 = E \rightarrow M \rightarrow M$ is a pipeline with three stages that can execute an arithmetic operation followed by two memory references or any order subset of these operations.

- A DAG $g_i$ of operations ($’max, ’+’, ’/’, ’L’,...$) is a proper sub-graph of a pipeline unit $P_j$ (denoted as $g_i \subset P_j$) if each node of $g_i$ can be uniquely mapped to one of $P_j$’s stages such that the topological order between the nodes of $g_i$ is consistent with the order in $P_j$. Since each $g_i$ corresponds to an extended machine instruction then we also require that each $g_i$ has a single root corresponding to the register that will hold the value computed by $g_i$’s operations when executed by $P_j$.

- A cover of a DAG $G$ by a set of pipeline units $\mathcal{P} = \{P_1, ..., P_p\}$ is an ordered set of $T$ tuples:

\[
S^0 = (\langle g_1^1 | g_2^1 | ... | g_p^1 \rangle)
\]
\[
S^1 = (\langle g_1^2 | g_2^2 | ... | g_p^2 \rangle)
\]
\[
\ldots \ldots
\]
\[
S^T = (\langle g_1^T | g_2^T | ... | g_p^T \rangle)
\]

such that

- The union of all $g_j^i$ exactly covers the vertices and edges of $G$ (same as the graph cover problem defined in [24]).

- Each $g_j^i \subset P_j$. Intuitively, each $S_i$ is a parallel instruction wherein $g_1^i | g_2^i | ... | g_p^i$ are executed in parallel by the $p$ pipeline units at step $t$.

- Each leaf of $g_j^i$ is either: a leaf of $G$, or the root of some previous $g_j^{i'}$ where $i' < i$.

For example figure 1 illustrates a possible cover of a 12 node DAG by $g_1, . . . , g_6$ (for convenience we omit the step index $t$ and enumerate the $g_j$’s by the order they appear). The resulting program has two steps $S^0, S^1$ and each $g_i$ is consistent with the pipeline unit it is executed by. Thus $g_4 \subset P_3$ as its $M$ node is executed in the first stage of $P_3$ and its $E$ node is executed by the second stage of $P_3$.

Apart for the resulting cover $S^0: S^1$ figure 1 (top right) contains the equivalent $C$ code of $g_1, ..., g_6$ where the original operation of the $E$ nodes (+ and /) is marked up-left to the $E$ nodes and the original operation Load/Store is marked up-left to the $M$ nodes. Note that Load-nodes have one argument and Store-nodes have two arguments (address, value) and that Store-nodes return a value. We remark that the use of multiple memory operations in $g_i$’s implies the use of multi-port memory, a feature that can be easily supported when the Ocpu memory is synthesized on-chip with The Ocpu. Next remark is that the Ocpu compiler generates binary encodings of the $g_i$-instructions but also generates the $C$-code of the $g_i$’s for readability, thus we will use the $C$-code to describe the different $g_i$’s.

- As in regular CPUs, there can be stalls or need to insert NOP between two consecutive $S^t$ and $S^{t+1}$. Let $v$ be a leaf node in $g_j^i \in S^t$ executed in stage $m$ of $P_j$ whose input comes from the root of some $g_j^{i'} \in S^{<t}$. Then the delay of $v$ ($d(v)$) is $d(v) = \max(k - m + (t - t'), 0)$ where $k$ is the number of stages in each pipeline unit. If $v$ has several inputs then $d(v)$ is the maximum $d(v)$ computed for each of $v$’s inputs. For example the delay of the $M$ node of $g_4$ in the cover of figure 1 to the root node of $g_1$ is $\max(3 - 1 + (1 - 0), 0) = 3$ and to the root node of $g_3$ it is $\max(3 - 1 + (1 - 0), 0) = 3$. The amount of NOP instructions that the Ocpu compiler inserts between $S^t$ and $S^{t+1}$ is the maximum $d(v) - 1$ over all leaf nodes of $g_1^1, ..., g_6^T$. Thus we need to insert two NOPs between $S^0$ and $S^1$ the Ocpu program of figure 1.

- Let $T_{p,k}$ be the number of steps needed to execute an Ocpu-program $G$ by a set of $l$ pipelines with $s$ stages each. Let $IPC_{p,k} = \#op \over T_{p,k}$ where $\#op$ is the total number of operations/nodes in $G$. We select $\max_{p,k} IPC_{p,k}$ as the grading function that combine both the IPC and the cost of the pipeline-to-registers interconnection. This is because the total amount of stages $p \cdot k$ represent a measure of the amount of resources used and $IPC_{p,k}$ but more weight on increased values of the IPC. In addition, $p^2$ models the complexity of the hardware needed to allow parallel access between the $p$ pipeline units and any register in the register file (e.g., a crossbar network). For example,
As an example to the ASIP generation problem consider the following possible pipeline configurations and the resulting execution time given two CPUs one with \( p = 16, k = 1 \) (i.e., a 16 width VLIW) and the same execution time \( T_{2,8} = T_{16,1} \), we get that the grade that is \( \frac{16}{2} \cdot k \) for \( p = 2, k = 8 \) is eight times bigger than the grade for \( p = 16, k = 1 \). We remark that this grading function does not fully models the case of clustered VLIW wherein the register file is partitioned between the pipeline units and a separate communication network allow accessing to the nonlocal parts of the register-file.

- The ASIP generation problem is, given \( k \) the number of pipeline stages and \( p \) number of pipeline units, to find a set of \( p' \leq p \) pipelines \( P = \{ P_1, \ldots, P_{p'} \} \) that covers \( G \) with \( g_i \)'s such that the above grading function is maximized.

As an example to the ASIP generation problem consider the following possible pipeline configurations and the resulting covers for the same \( G \) depicted in figures 2 and 3. As can be seen by the grade of each cover, the best choice for minimized execution time is not to use the largest \( g_i \) of figure 3 but the medium three-op \( g_i \)'s of figure 2.

III. THE ALGORITHM FOR FINDING GRAPH COVERING THAT MINIMIZES THE EXECUTION TIME

The proposed algorithm for computing an optimized \( p \), \( k \)-pipelined cover of a DAG \( G \) is based on the following ideas (due to space limitations we give only a short description of this algorithm):

1) Obtain a planar embedding of \( G \) in a 2D-grid such that the number of distanced edges (edges \( u \rightarrow v \) with a large distance on the grid) is minimized. This embedding is done using the Barycenter [3] layout of a graph which is is a 1-D arrangement of the nodes in every level of \( G \) such that the maximal distance of all edges (between levels and inside every level) is minimized. [3] describes a fast approximation algorithm for computing the Barycenter layout of \( G \). Thus each node in \( v \in G \) has an \( x, y \) coordinate.

Fig. 2. Possible cover using two pipelines of three stages each.

Fig. 3. Possible cover using two pipelines of seven stages each.

Fig. 4. Barycenter layout of a DAG and its partition to \( 3 \times 2 \) blocks.
v_i \in \{E', M'\}:

remove(P, v): remove the smallest prefix of stages from P that ends with v's type;

DFS(G, v, P):
Initially: Q = empty_stack; push(Q, v);
mark(v); P = remove(P, v);
While (not_empty(Q) and not_empty(P)) {
  w = top(Q);
  if (is_leaf(w) or all_sons_marked(w)) {
    list(w); P = remove(P, w); pop(Q, w);
  } else {
    L = all_unmarked_sons(w);
    push(Q, L); mark(L);
  }
}
if(not_empty(Q)) return (fail);
else return (listed_nodes);

Note that the order of the returned listed node is exactly the order in which g_i = DFS(G, v, P) will be executed in P had this g_i been executed by the pipeline unit P.

5) For a given set of nodes U (all root nodes in G), and a pipeline configuration \( \mathcal{P} = \{P_1, \ldots, P_p\} \), we define \( \langle S, G' \rangle = \text{pealing}(G, U, \mathcal{P}) \) as follows:
   a) Iterating over all different ordered subset \( \alpha \subseteq U \) of up to \( p \) nodes from \( U \).
   b) For each \( \alpha = \langle v_1; v_2; \ldots; v_p \rangle \) we compute \( S = \langle g_1 | g_2 | \ldots | g_p \rangle \) where \( g_j = DFS(G, v_j, P_j) \).
   c) We remove each \( g_j \in S \) from \( G \) by 1) replacing the root node of \( g_j \) in \( G \) by a register that will hold the result of \( g_j \) once computed; 2) eliminating all nodes (and corresponding edges) of \( g_j \) in \( G \) whose all of exiting paths pass through the root of \( g_j \).
   d) \( G' \) is the resulting graph after eliminating \( S = \langle g_1 | g_2 | \ldots | g_p \rangle \) from \( G \).
   e) For all the subset \( \alpha \subseteq \mathcal{P} \) we select the one with the smallest \( G' \), i.e., the \( S \) that removed maximal number of nodes from \( G \).

6) The cover of a block \( \text{cover}(B) \) is done iteratively in pealing steps:
   a) Let \( U \) be the current set of root nodes of \( B \).
   b) Compute \( \langle G', S \rangle = \text{pealing}(G, U, \mathcal{P}) \) and make \( S \) the next stopping \( S_t \) from the end in the resulting code to cover/compute \( B \).
   c) Make \( G = G' \) and repeat until all the nodes of \( B \) in \( G \) are removed.

7) The cover of \( G \) is done by covering each block in the order \( \text{cover}(G) = \text{cover}(B_0) \cup \ldots \cup \text{cover}(B_z) \) iterating over all possible pipeline configurations selecting the one whose overall execution time is minimal. Overall there are about \( p \cdot 2^{k_p} \) pipeline configurations; \( z = \frac{n \cdot 2^{k_p} \cdot k_p}{2^{k_p} + k} \) blocks; upto \( z \) pealing steps in each block; upto \( \frac{2^{k_p} \cdot k_p}{2^{k_p} + k} \) ways to select \( \alpha \) in each pealing step; and each \( DFS(G, v_j, P_j) \) is at most \( k \) steps. Thus the overall amount of steps is:

\[
\left( \frac{2^{k_p} \cdot k_p}{2^{k_p} + k} \cdot p \cdot 2^{k_p} \cdot z \right) \approx n \cdot e \cdot k^{2p}
\]

Fig. 5. Covering process by matching pipeline stages to nodes.

IV. RELATED WORKS

The research in ASIPs (see [14], [12] for a detailed survey) mainly address the following five key problems: (1) application analysis, (2) architectural design space exploration, (3) instruction set generation, (4) code synthesis and (5) hardware synthesis. Thus basically, an ASIP system selects an optimized ISA and CPU architecture then synthesizes the CPU and the compiler for the selected ISA. The hardware implementing the specific instructions can be either runtime reconfigurable functional units [13], [7], or pre-synthesized circuits. We target pre-synthesized circuits written in Verilog and the adaptation of this generic circuit is done using Verilog macro mechanism.

The main problem considered by this work, namely the ASIP generation of multi-op instructions for variable pipeline units (to the best of our knowledge) has not been previously addressed. Formally this contain the following:

- Given a DAG \( G \), how to find a minimal cover of \( G \) by a subset of convex subgraphs \( g_1, g_2, \ldots \) such that each \( g_i \) is consistent with a given set of ordered pipeline units (an order of memory and execution stages).
- How to find a set of such pipeline units that minimize the above cover.

In this respect the bottom-up algorithm and the search over all possible pipeline configurations is a heuristic solution that seems to be effective. The graph covering problem we consider is NP-complete since it is at least as hard as graph-partitioning. As indicated, previous works on ASIPs addressed a different problem which is to find a sub-set of convex sub-graphs \( g_1, g_2, \ldots \) which is most profitable under various considerations. Common to both problems is the need to cover a graph by a set of convex subgraphs however they differ by their requirements: minimal cover by subgraphs that are consistent with a given set of pipelines compare to a cover that optimizes a given set of desired properties.

This issue is related to the problem of optimal code generation and instruction selection where conventional methods use heuristics that break up the DAG into a forest of trees, which are then covered optimally but independently [1].

Other techniques to cover \( G \) for ASIP construction are as follows, [15] proposed a template generation, matching, and covering algorithm. It also starts with a DAG \( G \) of operations which does not include load/store operations since it target a
different setting than the one we consider. [15] devise an ASIP system for the synthesis of a hybrid reconfigurable hardware. The generation of $g_1, g_2, \ldots$ (called templates) is done by a clustering technique where in edges are contacted to nodes until sufficient set of sub-graphs is formed. The edges are selected based on a usage profile histogram, e.g., assume that $\ast \rightarrow \ast$ is the next most frequent edge then all occurrences of this edge will be contracted to one node. This technique generates both the instructions and the cover however graph-isomorphism is needed to locate which contracted nodes hold the same $g_i$. The templates that are most frequently used in the cover of $G$ are selected as the new ISA. The method was mainly used to create add-mull (MAC) type of instructions. They also found that each benchmark needed a different subset of these instructions. The underline chip is reconfigured to connect different computational units to form the instructions.

[2] considers extending generic processors with units specialized for a given domain rather than designing completely custom processors. Unlike previous works [2] generates multi-output instructions. VLIW architectures like ST200 and TMS320 can commit 4 values per cycle. A $g_i$s is selected as a convex sub-graph (called convex sub-graphs) that a) whose edges covers all the paths between the nodes of this $g_i$. The $g_i$s (called cuts) are selected by a branch and bound search over all possible cuts such that the execution of them as instructions is beneficial (using some latency measure).

[10] considers a reconfigurable FPGA as the basic adaptable structure for generating different ASIPs. The $g_i$s are also convex sub-graphs whose grade include not only the execution time but also the circuit size when $g_i$ is implemented in custom logic. The $g_i$s are formed by enumerating all possible convex sub-graphs in $G$ with their cost and apply a Binate Covering [26] to find a subset of $g_i$s with a minimal cost that covers all the nodes in $G$. The new $g_i$s are added to Altera Nios processor whose ISA can be reconfigured to implement complex $g_i$s over the FPGA. A later work [11] extended Nios with arithmetic multi-op instructions using pattern-matching to optimize $G$’s cover. The multi-op instructions are executed by an external reconfigurable unit that is connected to the register-file of the Nios. A special mechanism of Shadow-registers was used to reduce the latencies caused by the double connection to the register file. Binate Covering was also used in [19] to optimize the selection problem. A greedy type of solution to the covering problem used in [9] by adding new operations to existing multi-op instructions based on profitability measurements.

[5] combines two instructions to to form two-op instructions. A graph whose edges correspond to possible merges of instructions is generated. Two data dependencies conflict if when using one dependency as a complex instruction invalidates the possibility of the other becoming a complex instruction. In this way conflicting combinations are resolved. An interesting and different way of forming optimized instructions for speeding up algebraic computations is described in [20]. This is achieved by simplifying a set of polynomials using symbolic algebra equations.

Many works considers reconfiguration of hardware units as a way to generate ASIP. [22] describe a multi-core ASIP that can reconfigure subsets of its cores to form an more powerful cores that can exploit more instruction-level parallelism (ILP). In this way the ASIP can be reconfigured to exploit different levels of thread-level parallelism (TLP) versus different levels of ILP found in a given application. [8] extends LISA an Architecture Description Languages (ADL) supporting highlevel abstractions for CPU designs. Basically, [8] adds reconfigurability APIs to LISA and thus can use the reconfigurable resources to support automatic custom instruction identification and selection. [16] discusses the code generation issues for a runtime reconfigurable VLIW architecture that can combine different functional units in one template. The $g_i$s are generated using contraction rules applied to the a tree representation of the code (obtained by using SUIF). Multi-op instructions with multiple load/store operations was proposed once in [6] proposing that a scratch-pad memory connected by a DMA to the main memory will be used by the application-specific functional units that are used to execute the multi-op instructions. This limits the general use of multi-op instructions compared to the Ocpu wherein memory stages are part of the main pipeline and can interact with the main memory directly.

As indicated most ASIPs use a fixed rigid pipeline structure which is not considered a part that is adapted to the input program. However some works considered limited forms of adapted pipeline structure. [23] describe dual pipeline instruction set generation. This is achieved, by combining consecutive instructions of a given program measuring the expected gain. [21] proposed a technique to reduce the amount of input/output ports of the selected multi-op instructions. Registers are inserted in a specific locations in the circuits of the selected $g_i$s such that the access to the input/output ports is serialized and the number of ports is reduced. The proposed technique uses a general instruction selection technique (e.g., [2]) to generate $g_i$s for all possible combinations of I/O ports and apply serialization to each such $g_i$. Finally the best $g_i$s are selected.

Some commercial ASIP systems include cores like TenSilica [18], Coware [25], Expression vanaken1981expression. These systems synthesize ISAs with adaptable vectorized SIMD instructions, adaptable register file, and additional data transfer interfaces for multiprocessor communication. For example, the expression processor contains multiple processing elements (PE’s), which can be configured either as an SIMD array or as an expression tree instruction. The expression tree instruction is formed by using the parse tree as constructed by a compiler (i.e., using the AST of arithmetic and logical expressions). A sequence of tree instructions can be executed in a pipeline mode using an “X-Pipe” which is a binary-tree networks of PEs. This work is the most related to ours since it allows pipeline evaluation of multi-op arithmetic computations (up to 32) however it does not form general ASIPs since it is restricted to trees and it also does not have the automatic synthesis of instructions from arbitrary code (e.g., memory references) but rather the use of parse trees.

[4] shortly indicates the basic idea of using multi-op instructions for ASIPs, however the architecture, algorithm and results are not included.

V. ISA AND THE OCPU STRUCTURE

The Ocpu architecture containing a flexible set of pipeline stages. This set is synthesized automatically in Verilog by
setting the value of several constants such as number of
registers, word size, number of pipeline units, number of
stages in each pipe and the type of each unit (with/without
increments). A pipeline stage holds new values computed at
this stage $i$ and also values from previous stages. Note that by
copying the registers we avoid the complexity of connecting
each stage to a common register file justifying the selected
grading function. If not updated at the current state the input
state is automatically copied to the next state and the state of
last stage forms the input state to the first stage. The
execution of a compute instruction connects the operations at
each stage to input/output registers and latches as described
by the different arrows in the figure. Increment operations can
be done only at the first stage affecting not only the suitable
register at the next stage but also the register at the input
stage. This is done in order that the next instruction that is
executed by the Ocpu will observe the incremented register
of a load/store operation done at the first stage by a previous
instruction. It follows that increment operations (in M-stages)
are visible to the next instruction only if they are executed
in the first stage or at the last stage. Thus, the Ocpu supports
visibility of increment operations at the first and the last stage,
however these increments should not be made to the same
register.

Except for increments, registers at the first stage are
updated only from values from the last stage. Thus nop-
instructions (stalls) must be inserted by the compiler in case
that an instruction $I_1$ needs a value computed by a previous
instruction that is not ready when $I_1$ enters the pipe. The Ocpu
is an exposed-pipe architecture similar to T1600... cpus. An
Ocpu uses 32-bits general registers that are used for computa-
tions and for passing parameters to functions. Each instruction
must update one register (even if it’s a store operation) that is
written back to the register file at the last stage of the pipe.
Branch instructions are executed in a pre-stage of the pipeline
(branch stage) along with the instruction-fetch operation. Thus
branch instructions always complete (change the PC) in one
clock cycles. Consequently there is no need to flush the pipe
when a branch instruction is miss predicted. Multiple branch
instructions in the same Ocpu-instruction [branch][branch] are
not allowed, and branch instructions are assumed to reside only
in the instruction memory of the first pipe.

The resulting ISA include (see figure 6): NOP, Multi-op
compute instructions, Move instructions, Function call/ret
executed at the first stage, and Hardware Loop instructions
that execute loops by loading a counter and repeating the body
until the counter decreases to zero. Conditional branch
are executed in the first stage of the pipe and therefore can affect
the next instruction address (PC) without any need to flush
instructions from the pipes. Branch instructions can contain a
complex logical and relational expression that is used to
evaluate the branch condition.

Finally figure 7 illustrates (left) an Ocpu that has three
registers ($R_1, R_2, R_3$), two pipeline units with three stages
each $E \to M \to E$ and $M \to E \to M$. For example, executing $R_4 = *(R_2 + R_3)$
$[R_5 = R_3 + R_2$ after $[R_3 = R_2 + (*R_1)]$]
$[R_2 = R_2 + 1]$ using the Ocpu of figure
7 (left) will require adding two nop-instructions between
the two instructions since $R_2 + (*R_1)$ is ready after the the
third stage but is needed at the first stage of $R_4 = *(R_2 * R_3)$.

Fig. 6. Instruction’s format.

Fig. 7. 2 $\times$ 3 Ocpu and its reconfiguration to a 1 $\times$ 6 Ocpu.

Note that interconnections to the register file is done only at
the first pipeline stage and that all other stages use values
that are copied from the previous stage justifying the grading
function we are using. By redirecting the wires of the last
stage, the Ocpu can easily allow dynamic partition/merging of
its pipeline units (see figure 7 right).

VI. EXPERIMENTAL RESULTS

The overall system we implemented contains: 1) an LLVM
[17] compiler that generates optimized LLVM RTL bytecode;
2) a special pass to convert the RTL+CFG+dependencies of the
LLVM to the graph of operations $G$; 3) unrolling is applied to
increase the number of operations in each $G$; and 4) a compiler
pass called the Other-compiler implements the ASIP selection
algorithm generating the ISA, pipeline configuration and the
assembly code for the selected Ocpu. The selected architecture
is automatically synthesized to Verilog using the generate-for-
loop mechanism of Verilog. The assembler generated by the
Other compiler is compiled to object-code and is added to
CPU as a Verilog module. Synthesis results obtained for a
32-bit Ocpu with two pipeline units of five stages each are
as follows: ARM ASIC CPU- Freq : 1200 Mhz Area: 3.68
mm$^2$; ARM CPU FPGA- Freq 70Mhz registers 57132 (we
use Altera Stratix III 340); Ocpu ASIC- Freq 1500Mhz Area
0.4mm$^2$; and Ocpu FPGA- Freq 150Mhz.

Next, we give results as to the effectiveness of the proposed
scheme. We evaluated several kernel benchmarks from the
HPEC Challenge that consist of ANSI C implementations of nine kernel benchmarks. The benchmarks are: time-domain finite impulse response filter bank (tdFir), singular value decomposition (svd), pattern matching (pm), graph optimization via genetic algorithm (ga) and real-time database operations (DB). We used the above system to compile one innermost loop from each benchmark using different pipeline configurations. For each pipeline configuration \( \text{pipe}^p_x \) we measured the IPC (instructions per cycle). The IPC reflects how many operations were executed in parallel in an average cycle. Ideally, when \( p \) pipeline units are used each with \( k \) stages then the IPC should be equal to \( p \cdot k \), e.g., for \( p = 2 \) and \( k = 3 \) we can hope for \( IPC = 6 \) implying full utilization of the pipeline-stages in every cycle. Dependencies between instructions and lack of resources can reduce the IPC.

The results table I and table II depict the \( \text{Cycles}/\text{IPC} \) for two pipeline units \( p = 2 \) with \( k = 2, 3, 4, 5 \) stages compared to four pipeline units \( p = 4 \) with \( k = 2, 3, 4, 5 \) stages. The size indicates the number of operations in the code. The number of Cycles \( C \) is the resulting execution time \( T \) including stalls caused by dependencies between the instructions of the unrolled loop (this include load/store dependencies). The goal was to estimate the effect of first increasing the number of pipeline units and then the effect of increasing the number of pipeline stages.

- In all entries using four pipeline units was better that using two pipeline units with an equivalent number of stages.
- In all cases increasing the number of pipeline stages improved the IPC. In some cases increasing the number of stages had almost no effect but this was mainly to the small size kernels.
- As can be seen in most cases IPC is close to \( p \cdot k/2 \). This implies linear scaling and that the Ocpu architecture can be used to speedup execution times linearly.
- The numbers in round brackets are for an equivalent VLIW configuration with \( p \cdot k \) pipeline units of one stage each. We only computed VLIW for up to eight execution units. As can be seen the equivalent VLIW obtains a bit better IPCs than the equivalent \( p \times k \) Ocpu. Clearly any scheduling of \( p \) pipeline units of \( k \) stages each can be imitated by a \( p \cdot k \) VLIW configuration, hence VLIW is likely to obtain better IPCs. However, it is the cost of the interconnections that matters, as a \( p \cdot k \times 1 \) VLIW configuration should obtain an IPC that is \( k \) times larger than that of a \( p \times k \) Ocpu in order to obtain the same grade \( IPC^2/(p^2 \cdot k) \). Since in all cases the difference between the IPCs was rather small then there is a clear motivation to use non VLIW configurations.

Next we view the results according to the combined measurement of \( IPC^2/(p^2 \cdot k) \). Table III depicts the \( IPC^2/(p^2 \cdot k) \) for different pipeline configurations ranging from 2 x 2 to 4 x 5. For each benchmark the \( p \times q \) configuration that achieved the maximum grade is marked in bold. The 2 x 5 and 2 x 3 configuration obtained the maximum grade for nine out of 13 benchmarks then 4 x 2 (3 times). In a similar experiments we considered a fixed budget of 12 operations (stages) and tested, for each benchmark, which \( p \times k \) values obtain the best grade. The results are Cfar 4x3, ct 3x4, eMul 3x4, Llist 2x6, ctV 6x2, db 6x2, dbV 6x2, pm 6x2, pmV 4x3, qr 4x3, qrV 3x4, ssv 6x2, and tdFir 2x6. Overall, the results show that:

- Most of the best Ocpu configurations were with \( k > 2 \) showing the effectiveness of pipeline execution of multi-op instructions, i.e., had the \( p = 6 \times k = 2 \) win it would show that the VLIW configuration is better than the proposed OCPU.
- Basically different best configurations were obtained for different benchmarks showing that there is a need for an ASIP mode of compilation rather than using the same configuration for all cases.
- Most of the benchmarks preferred more execution units over memory units in the ratio 1:2.5 (M:E). This is in accordance to the frequency of E operations versus M operations in the code.

It may be of interest to compare the Ocpu to an embedded CPU such as the ARM11. Preliminary experiments show that the gap between the two architectures is larger than X5, hence we only demonstrate it with a single example. Consider the following loop:

```
for (i=0; i<numrows; i++)
for (j=0; j<numcols; j++)
    *(out+j*numrows+i) = *(in+i*numcols+j);
```

Using the Ocpu Other compiler we got 16 vertices. In the GCC ARM compiler, we got 50 instructions for the nested loop and 20 instructions for the internal loop. The Ocpu completed this
TABLE III. \(IPC^2/(gp^2 \times k)\) FOR DIFFERENT PIPELINE CONFIGURATIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>2x2</th>
<th>2x3</th>
<th>2x4</th>
<th>2x5</th>
<th>4x2</th>
<th>4x3</th>
<th>4x4</th>
<th>4x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cfar</td>
<td>0.98</td>
<td>1.41</td>
<td>1.20</td>
<td>1.25</td>
<td>1.06</td>
<td>1.02</td>
<td>0.95</td>
<td>0.76</td>
</tr>
<tr>
<td>ct</td>
<td>0.50</td>
<td>0.93</td>
<td>0.69</td>
<td>0.56</td>
<td>0.52</td>
<td>0.52</td>
<td>0.39</td>
<td>0.31</td>
</tr>
<tr>
<td>eMul</td>
<td>0.78</td>
<td>0.93</td>
<td>0.69</td>
<td>0.56</td>
<td>0.52</td>
<td>0.52</td>
<td>0.39</td>
<td>0.31</td>
</tr>
<tr>
<td>Llist</td>
<td>2.00</td>
<td>1.33</td>
<td>1.00</td>
<td>0.80</td>
<td>2.01</td>
<td>1.33</td>
<td>1.00</td>
<td>0.20</td>
</tr>
<tr>
<td>ctV</td>
<td>1.02</td>
<td>1.41</td>
<td>1.39</td>
<td>1.45</td>
<td>1.23</td>
<td>1.36</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>db</td>
<td>1.72</td>
<td>1.83</td>
<td>1.53</td>
<td>1.55</td>
<td>2.05</td>
<td>1.65</td>
<td>1.53</td>
<td>1.55</td>
</tr>
<tr>
<td>dbV</td>
<td>0.84</td>
<td>1.13</td>
<td>0.10</td>
<td>1.13</td>
<td>1.25</td>
<td>0.87</td>
<td>1.06</td>
<td>0.92</td>
</tr>
<tr>
<td>pm</td>
<td>1.04</td>
<td>1.28</td>
<td>1.18</td>
<td>1.20</td>
<td>1.21</td>
<td>1.28</td>
<td>1.08</td>
<td>0.98</td>
</tr>
<tr>
<td>pmV</td>
<td>0.84</td>
<td>1.10</td>
<td>1.13</td>
<td>1.25</td>
<td>0.87</td>
<td>1.06</td>
<td>0.92</td>
<td>0.87</td>
</tr>
<tr>
<td>qr</td>
<td>1.02</td>
<td>1.42</td>
<td>1.32</td>
<td>1.23</td>
<td>1.26</td>
<td>1.33</td>
<td>1.14</td>
<td>1.05</td>
</tr>
<tr>
<td>qrV</td>
<td>1.25</td>
<td>1.48</td>
<td>1.34</td>
<td>1.55</td>
<td>1.67</td>
<td>1.39</td>
<td>1.34</td>
<td>1.07</td>
</tr>
<tr>
<td>sVd</td>
<td>1.45</td>
<td>2.11</td>
<td>2.19</td>
<td>2.31</td>
<td>1.60</td>
<td>1.97</td>
<td>1.71</td>
<td>1.61</td>
</tr>
<tr>
<td>tdFir</td>
<td>0.90</td>
<td>1.13</td>
<td>1.07</td>
<td>1.34</td>
<td>0.97</td>
<td>1.13</td>
<td>1.04</td>
<td>0.96</td>
</tr>
</tbody>
</table>

loop in 2 cycles for a 2x4 configuration and 3 cycles for the 4x2 configuration, yielding a gap of 7 cycles compared to the ARM-7 ISA. Since both CPUs have the same clock latency this gap is significant. Clearly such gaps will be obtained for compiling larger code segments since the Ocpu has more resources and its control structure (loops, conditional-statements and function-calls) use heavy instructions and thus take less cycles than the ARM-7 ISA. Table IV contains more compilation results between the ARM and Ocpu, it gives the total amount of instructions for both CPUs (including the NOPs for the Ocpu) showing similar gaps.

TABLE IV. \(\text{ARM} \#\text{INSTRUCTIONS VERSUS OCPU} \#\text{INSTRUCTIONS AND CYCLES.}\)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>ARM #inst.</th>
<th>#nodes</th>
<th>2x6 Ocpu #inst.</th>
<th>6x2 Ocpu #inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ct</td>
<td>50</td>
<td>16</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>initGA</td>
<td>210</td>
<td>352</td>
<td>52</td>
<td>35</td>
</tr>
<tr>
<td>tdFirSetup</td>
<td>450</td>
<td>306</td>
<td>48</td>
<td>25</td>
</tr>
<tr>
<td>mainpm</td>
<td>540</td>
<td>367</td>
<td>47</td>
<td>30</td>
</tr>
</tbody>
</table>

REFERENCES


