Introducing Control-Flow Inclusion to Support Pipelining in Custom Instruction Set Extensions

Marcela Zuluaga*, Theo Kluter†, Philip Brisk†, Nigel Topham*, Paolo Ienne†

* University of Edinburgh
School of Informatics, Institute for Computing Systems Architecture, Edinburgh, UK
E-mails: g.m.zuluaga@sms.ed.ac.uk, npt@inf.ed.ac.uk
† Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland
E-mails: {Theo.Kluter, Philip.Brisk, Paolo.Ienne}@epfl.ch

***Abstract*—Multi-cycle Instruction set extensions (ISE) can be pipelined in order to increase their throughput; however, typical program traces seldom contain consecutive calls to the same ISE that would allow this temporal parallelism. Often, there are intermittent calls to branch instructions, at a minimum, that prevent the pipelined execution of subsequent calls to the same ISE within a loop. What is needed is ISEs that cover an entire loop body, which can create a stream of repeated calls to the same ISE during program execution; this, in turn, permits the use of hardware pipelining. To address this concern, we introduce a new type of ISE that borrows ideas from zero-overhead loop instructions to permit pipelined execution of loops. To further expose instruction-level parallelism, the ISE supports loops whose bodies form hyperblocks, which are regions of program control flow that have multiple exits (including loop iterations and break points within loops). These ISEs broaden the scope of instruction-level parallelism and obtain higher speed ups compared to traditional ISEs, primarily through pipelining, the exploitation of spatial parallelism, and reducing the overhead of control flow statements and branches.**

I. Introduction

The market for embedded processors is driven by stringent requirements, namely cost, time-to-market, power/energy consumption, and performance. Processor architects must begin with knowledge of the application domain in order to meet such demands; however, design automation can further help to meet time and cost constraints.

Numerous design methodologies for application-specific instruction set processors (ASIPs) have been proposed. There are two general approaches: loosely coupled external coprocessors, such as loop accelerators, and customizations to the instruction set architecture (ISA), which permits application-specific tuning of the instruction set, cache size, register file size, branch predictors, and other microarchitectural features. Commercial processors that feature application-specific enhancements include the Xtensa by Tensilica [10] and ARC-tangent by ARC [1].

Much research in recent years has focused on ASIP design automation, e.g., to find the best partition of an application into hardware, either as co-processors or custom instruction set extensions (ISEs). A co-processor can provide substantial acceleration, but requires significant area and often has a large communication overhead. At present, the designer is most often responsible for partitioning the application; however, once the partition is defined, advanced behavioral synthesis methods can automatically generate the accelerators.

ISEs, on the other hand, offer speedups at a finer granularity. Unlike coprocessors, which are loosely coupled, ISEs are tightly coupled with the processor pipeline, and can exchange scalar data with the processor's register file every cycle. ISEs have evolved to the point where they have their own local memories [3], and, as a consequence, can achieve speedups comparable to co-processors.

This paper presents an innovative method to create multi-cycle ISEs that are executed as hardware pipelines that comprise complete loops, including those with irregular control flow. This approach is justifiable because most programs spend the bulk of their runtime in a few deeply nested loops, and this is particularly true in embedded applications. Our ISEs for loop acceleration are based on zero-overhead loop instructions; to increase instruction-level parallelism, our method supports hyperblocks, which have multiple exits. We support this feature to facilitate loops that have infrequently executed regions that contain operations, such as function calls, that present ISEs cannot support in hardware.

To evaluate the effectiveness of our techniques, we extend the OpenRISC processor with our ISEs, and execute the resulting system on an FPGA as a soft processor. This permits us to measure, rather than estimate, the performance gain of our approach in comparison to existing techniques that employ smaller ISEs that cannot affect control flow. Existing ISEs cannot modify the program counter, therefore they must have a single entry and a single exit; they can execute the computations within a single loop iteration, but they cannot execute several iterations at once (let alone the entire loop); the most important drawback here is that existing ISEs cannot pipeline the loop.

A detailed case study of the JPEG application shows that our method achieves a speedup of $3.1\times$ over pure software execution; in contrast, the most sophisticated ISEs that exist prior to our work, which can access local memories but without pipelining or support for hyperblocks, achieve a speedup

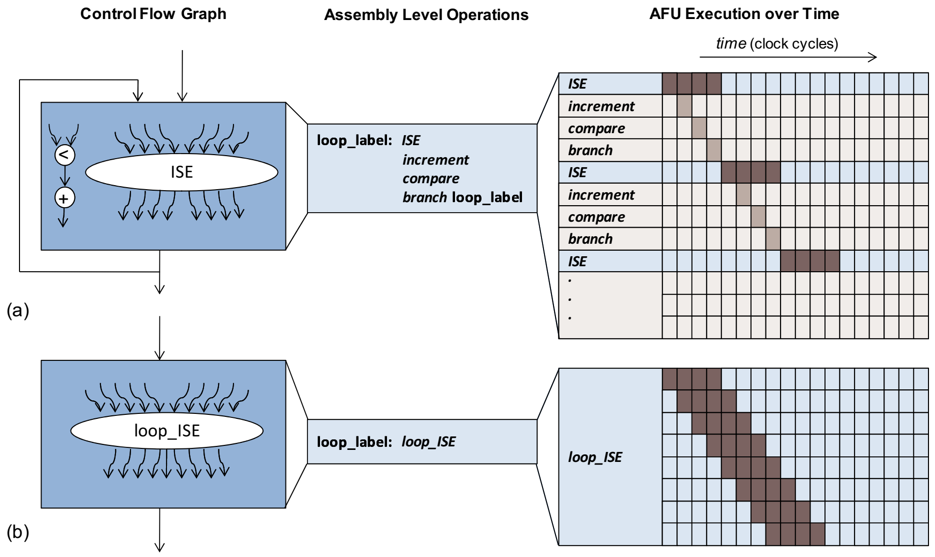


Fig. 1. (a) A basic block extracted from the control flow graph of the DCT kernel in the JPEG application. In this basic block, a multi-cycle ISE is identified using traditional techniques. This instruction takes 4 clock-cycles to complete in the AFU. Assuming that the ISE is a non-blocking instruction, i.e. the operations following it can be issued before its completion, the computations in the AFU cannot be overlapped in order to take advantage of a pipelined AFU. (b) All operations in the basic block are included in one ISE, and the AFU issues every iteration. Thus, operations in the AFU can overlap in time, yielding a speedup of at least 2.9× on the loop execution by taking advantage of a pipelined AFU.

of 2.2× over software.

II. Pipelining ISEs

Early ISEs were designed to interface only with a processor's register file; in a single cycle, all inputs were read, the computation was performed, and all outputs were written. These restrictions were stringent, because the typical register file in a RISC processor has two read ports and one write port. Limited I/O has been a major problem to overcome, and several solutions have been presented to address this concern [3], [7], [13], [16], [24]. Increasing the data bandwidth to and from ISEs facilitates the creation of larger and larger ISEs, which increases parallelism and application performance. State-of-the-art ISEs are multi-cycle, may read their data from the memory subsystem in addition to the register file and processor pipeline, and execute synchronously with the main processor clock.

These multi-cycle ISEs can be pipelined in order to increase their throughput. Pipelining divides the circuit into several execution stages, allowing it to operate concurrently on different inputs. This permits the simultaneous exploitation of both spatial and temporal parallelism in order to increase throughput. To exploit a hardware pipeline, several ISEs must be emitted in consecutive cycles; however, typical program streams rarely contain consecutive invocations of the same ISE that would permit this type of execution. This requires the use of ISEs that cover entire loop bodies; prior ISE identification methods, e.g., [23], are insufficient, as they cannot include memory accesses (loads and stores), branch instructions that modify the program counter of the base processor, or function calls. Historically, only the base processor can execute these types of operations.

Biswas *et al.* [3] and Kluter *et al.* [17] introduced methods to facilitate architecturally visible storage. Under this scheme, ISE data can be communicated through a coherent scratchpad memory in addition to the register file. *Direct Memory Access (DMA)* operations are required to move data between the scratchpad memory and off-chip RAM. Load and store operations that access the scratchpad can be included in the ISE. This significantly increases I/O bandwidth and the speedups achievable through ISEs.

The hardware realization of a specific ISE is called an *Application-specific Functional Unit (AFU)*. As an example, consider the DCT kernel from JPEG encoding, as shown in Fig. 1. The body of the main loop of the kernel is identified as an ISE. When 8 inputs are available, the AFU requires 4 cycles. As there are no loop-carried dependencies, this instruction can be pipelined, yielding a speedup of 2.9× for this kernel.

Unfortunately, there is one limiting factor: several operations relating to the loop itself must be done in software. In particular, the loop counter must be incremented, compared with the maximum loop count, and a conditional branch that determines whether the loop continues, must all execute in software. This will require that the processor issues at least three instructions per iteration to facilitate the loop; this, in effect, cancels any benefits from the pipelined AFU.

As shown in the time table in Fig. 1(a), the computations in the AFU cannot be overlapped in order to take advantage of a pipelined AFU. The shaded squares account for the number of cycles taken by the execution stage of each instruction. Following the ISE, the processor issues the three consecutive instructions. When the branch is taken, the processor issues another ISE. At this point, the AFU has already finished the computations of the previous instruction.

This example assumes that the ISE is a non-blocking instruction, i.e. the operations following it can be issued before its completion. This type of overlap is feasible for long-latency ISEs, but not for single-cycle ones. If the ISE is a blocking instruction, then the processor must wait for the AFU to finish before it can issue another instruction, which prohibits concurrent execution of software instructions with the ISE.

Zero Overhead Loops

To address this concern, hardware pipelined ISEs can be implemented in a style that is similar in principle to zero-overhead loop instructions. The three software operations described above are now integrated into a single complex instruction that controls the iteration of the loop. The ISE implementing the pipelined loop body executes continuously until the breaking condition is met. By executing these operations as a zero overhead loop, the impediments to pipelined ISE execution, as stated above, are mostly eliminated. This permits ISEs to issue automatically every clock cycle, which, in turn, permits the use of every stage of the pipeline during loop iterations. This situation is shown in Fig. 1(b), where operations in the AFU can overlap in time, reducing the execution time of the loop by taking advantage of a pipelined AFU.

III. Allowing Loop Bodies with Multiple Exits

Significant performance improvements can be obtained by pipelining critical loops. However, loops often contain structures that cannot be included in a single ISE without introducing control dependencies. These structures include multiple control flow paths, multiple exits, inner loops and calls to funtions that cannot be inlined. In these cases, unimportant paths with high resource usage can prohibit the optimization of the execution of more important paths. To mitigate this problem and further expose instruction-level parallelism, we propose ISEs that support loops whose bodies form hyperblocks [21].

A hyperblock is a single-entry, multiple-exit region of the control flow of a program, with no internal join points and no loops. Hyperblocks support predicated execution, which has already been considered in the field of custom instructions [4]. The use of predication increases the size of regions that correspond to a single path of control flow, which, in turn, increases the likelihood of finding instruction-level parallelism. Unfortunately, predication does not always remove all control flow disruptions, and many blocks that can be predicated are poor candidates for custom intructions. Additionally, unbalanced branches may be costly to predicate, as if-conversion always executes both sides of a branch. Moreover, one side of the branch may contain forbidden operations, such as a function call, that cannot become part of an ISE.

To construct hyperblocks, applications are profiled to identify hot loops and determine which branches are taken most frequently; heuristics are then applied to select which basic blocks are consolidated into a hyperblock. Furthermore, techniques such as loop unrolling, tail duplication, and loop peeling can assist hyperblock formation without altering the correctness of program execution.

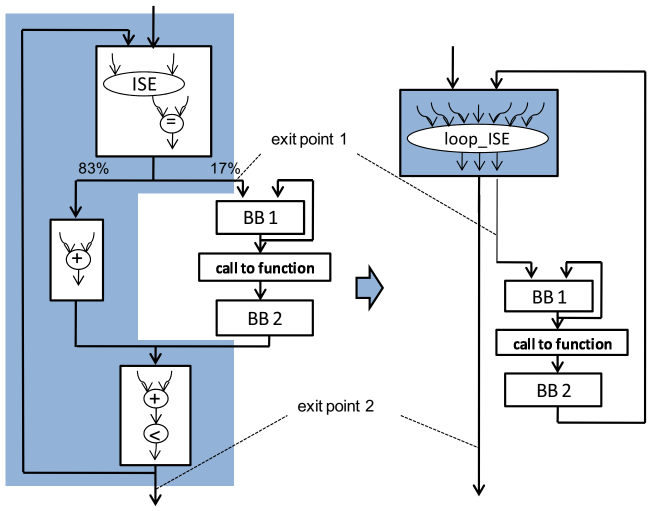


Fig. 2. The detection of a hyperblock. As one of the branches is only executed 17% of the time during profiling, it can be excluded. This allows the formation of a loop ISE with multiple exit points in order to optimize the execution of the most taken trace in the loop.

As an example, Fig. 2 shows the entropy encoding kernel from JPEG. Traditional techniques can find an ISE in the first basic block of the control flow segment. Unfortunately, there is a control flow disruption that prevents the complete loop from being converted into an ISE. Profiling indicates that the branch leading to a function call is rarely executed. Fig. 2 shows that a multi-exit hyperblock can be formed so that the entire loop body, except for the rarely executed branch, can be implemented in custom hardware. For the common case, when the disruption does not occur, the entire loop body will benefit from the speedup achieved through the ISE.

The new loop instruction will contain the operations corresponding to two conditions, one to exit the loop, and another one to execute the branch that has been excluded.

IV. Identification of Loop Instructions

The process for identifying loop ISEs is described as follows:

1) The application is profiled and hyperblocks are formed. Hyperblocks that contain operations that can always be included in an ISE trivially become loop ISEs.
2) Hyperblocks that contain at least one operation that is forbidden from inclusion of an ISE are deformed, i.e., the hyperblock is discarded and replaced with the original control flow constructs.
3) Traditional ISE identification [23] is carried out in the remaining basic blocks and hyperblocks in the application. Methods that can exploit architecturally visible storage [3], [17] are used:
    a) Data structures (arrays) accessed in the basic block or hyperblock are identified using existing disambiguation techniques. All load/store instructions to

data structures that cannot be disambiguated are forbidden from inclusion in an ISE.

b) DMA transfers are placed in the most profitable positions.

4) Instances of ISEs are inserted in the program at the appropriate locations.

V. AFU Implementation

Fig. 3 shows the interface between the AFU and the base processor. The processor provides control signals to initiate the ISE, along with read/write interfaces to its register file. In typical RISC processors, the register file can provide two inputs and read one output from the AFU every cycle. An AFU controller, which receives the initialization signals, enables the pipelined datapath execution and activates the AFU's local storage units.

The AFU datapath has its own architecturally visible local memory. This memory can provide inputs to the ISE and store its results after the loop executes. DMA transfers data from the main memory to the local memory for ISE execution, and copies the data back after the loop terminates, without stalling the processor. As noted by Kluter *et al.* [17], this type of memory access creates coherence problems, as data that is modified in the AFU local memory and written back to the memory may be different from the values of the same data that reside in the data cache. A hardware or software coherence mechanism needs to be used to prevent the cache to AFU local memory coherence problem.

Additionally, a local register file is used to store loop-carried variables. Specialized load and store instructions are included in the set of ISEs to permit the reading and writing to this local register file.

The AFU outputs that correspond to loop break conditions are passed to its control unit, which transfers control back to the processor. To facilitate correct execution in the presence of multiple exit points, the AFU also returns to the processor the address of the correct exit point. A loop ISE has itself as the default destination, which is achieved by implicitly issuing the same instruction within the AFU. The remaining destinations are the other hyperblock exits, e.g., break instructions in the original program's control flow, including the eventual completion of the loop.

The local register file also stores the possible destinations of the ISE loops. The ISE store instruction is used to load these values at the beginning of the execution of each loop ISE. Although this data is all compile-time constant, the register file would be quite large if it must store every exit address for every loop ISE. In actuality, the maximum number of local registers will be the sum of the number of exit point plus the number of loop-carried variables for the ISE that uses most.

Due to the use of hyperblocks, loop ISEs can be viewed as a complex multi-target branch instruction. Implicitly, this suggests that the AFU would need to modify the Program Counter (PC) in the fetch stage of the base processor pipeline. Sidestepping this issue is beneficial, as branching out of the

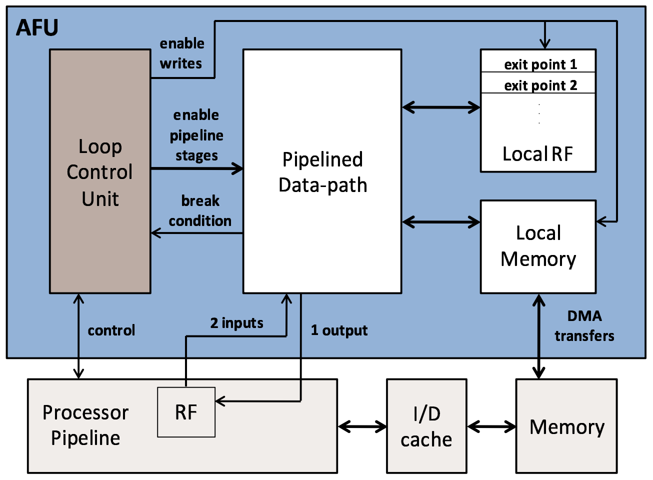


Fig. 3. System overview of an accelerated processor. The Loop Control Unit enables the pipelined execution of the AFU. Furthermore, it detects the break condition and sends back the exit addresses (see Fig. 2) stored in the local register file.

hyperblocks directly would issue the result of the branch to the branch predictor.

In a sense, the formation of a hyperblock can be viewed as a type of profile-guided static branch prediction. Consequently, branches that have been absorbed into the hyperblock, including hyperblock exits, are removed from the program and are no longer issued to the branch predictor; conflicts involving these branches are eliminated as a consequence.

As hyperblock execution is not driven by the PC, there is no need to predict branch target addresses for exiting the hyperblock. Consequently, updating the PC directly on a hyperblock exit would overwrite meaningful entries in the predictor itself. In contrast, our approach returns the branch target address to the processor pipeline as data, permitting the use of a direct, rather than a conditional branch instruction, which will not update the predictor.

Initiation Constraints

The *Initiation Interval (II)*, in the context of pipelining, is the number of cycles the AFU must wait before issuing the next iteration of the loop. To initiate a new ISE every clock cycle, the aggregate I/O bandwidth required for its computations cannot exceed the number of available memory and register file read and write ports. If these conditions are not met, then the input and output operations must be scheduled accordingly. This can be taken as the resource constraints of the initiation interval of the loop, and can be formalized as follows:

$$ResII = \max\left(\left\lceil\frac{\text{inputs}}{\text{in ports}}\right\rceil, \left\lceil\frac{\text{outputs}}{\text{out ports}}\right\rceil\right) \quad (1)$$

In other words, *ResII* is the initiation interval, as determined solely by resource constraints.

The greatest dependency distance, in clock cycles, found between iterations is another constraining factor of the initiation interval. This is known as recurrence constraint *RecII*.

The maximum instruction throughput is the inverse of stage delay or initiation interval of the AFU. It specifies the number of clock cycles between the initiations of sequential instructions into the AFU. It is calculated as follows:

$$II = \max\left(ResII, RecII\right) \tag{2}$$

Pipeline scheduling

As-Soon-As-Possible (ASAP) scheduling [8] is applied to the ISE so that the loop break conditions are evaluated at the earliest possible point; however, the pipelined execution of several loop iterations at once, technically, is speculative. Therefore, write operations must not be scheduled before the break conditions of the previous iterations are resolved. The alternative is to implement the interface by which the AFU writes data back to the processor using some form of gated write buffer, which is beyond the scope of this work. Inputs and outputs are then scheduled along different pipeline stages making sure that the number of read and write accesses does not exceed the number of ports at any time.

VI. Code Example

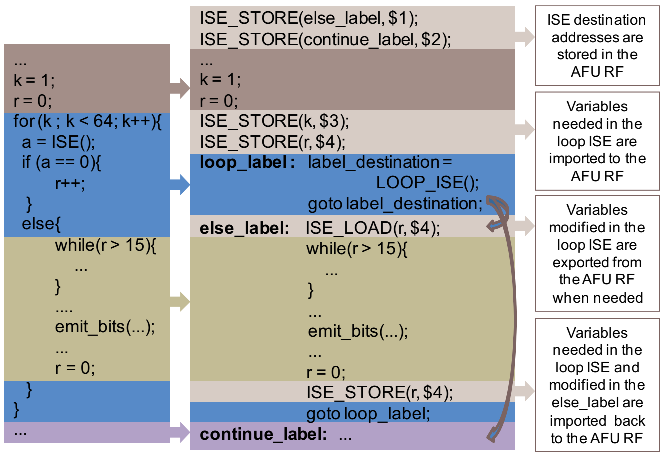
Fig. 4 shows the high-level code transformations that are required to facilitate the use of a loop ISE for the JPEG entropy encoding kernel shown in Fig. 2. Before the loop, the addresses of the exit points are loaded, as discussed above, using a dedicated instruction, `ISE_STORE`. The `for` loop is replaced with a `LOOP_ISE`, effectively the high-level language equivalent of a zero-overhead loop assembly instruction. Unlike a traditional zero-overhead loop instruction, `LOOP_ISE` returns the destination address of the exit point. A jump instruction following the loop ISE then transfers control to the appropriate point of continuation.

Recall that the loop ISE may contain control flow that exits the ISE, executes some statements in software, and then re-enters the ISE for the next iteration. To facilitate software execution, data must be moved from the AFU back into the processor in advance; this is accomplished with the `ISE_LOAD` instruction, which moves variable `r` back into the register file. If data in the AFU local memory is required, then a DMA transfer may be initiated as well. Next, the software code executes until it finishes. Afterwards, any values modified by the software code that may be needed by future iterations of the loop are written back to the AFU. In this case, `r` is re-initialized to 0 and sent back to the AFU using an `ISE_STORE` instruction.

If the loop terminates, the destination address provided by the ISE corresponds to the `continue_label`. In this case, there is no return to the ISE, so normal software execution continues after the loop.

VII. Experimental Setup

We used the JPEG encoding/decoding chain, taken from the EEMBC benchmark suite [12], to evaluate loop ISEs. JPEG compression is comprised of three kernels: Discrete Cosine Transformation (DCT), quantization, and entropy encoding;



```
...
k = 1;
r = 0;
for (k ; k < 64; k++){
  a = ISE();
  if (a == 0){
       r++;
  }
  else{
       while(r > 15){
          ...
       }
       ....
       emit_bits(...);
       ...
       r = 0;
  }
}
...
```

```
ISE_STORE(else_label, $1);
ISE_STORE(continue_label, $2);
...
k = 1;
r = 0;
ISE_STORE(k, $3);
ISE_STORE(r, $4);
loop_label :  label_destination =
                          LOOP_ISE();
              goto label_destination;
else_label:   ISE_LOAD(r, $4);
              while(r > 15){
                 ...
              }
              ...
              emit_bits(...);
              ...
              r = 0;
              ISE_STORE(r, $4);
              goto loop_label;
continue_label:  ...
```

ISE destination addresses are stored in the AFU RF
Variables needed in the loop ISE are imported to the AFU RF
Variables modified in the loop ISE are exported from the AFU RF when needed
Variables needed in the loop ISE and modified in the else_label are imported back to the AFU RF

Fig. 4. Code abstraction, taken from the original application, showing the transformations needed in order to use a loop ISE.

likewise, JPEG decompression is composed of entropy decoding, de-quantization, and Inverse DCT (IDCT). For all of the experiments, we used a 24-bit RGB-encoded picture with a resolution of 1024x768 pixels, which is comparable to the image resolution found in current web-cams and mobile phone cameras. JPEG was chosen because many of the smaller kernels in EEMBC contain straightforward loops with no internal control flow, and therefore do not require hyperblocks.

For the purpose of comparison, we implemented prior methods for ISE generation that do not encompass full loops. Our evaluation platform is an OpenRISC processor, with an interface for custom instructions that is similar in principle to Altera's Nios II soft processor.

We compare with ISEs identified by Pozzi *et al.* [23] which cannot access data in memory. These type of ISEs can only interface with the processor's register file; the register file of our target processor has two read ports and one write port, so ISE identification uses these constraints. For this approach, instruction identification, C code generation and ISE implementation in VHDL have been done by automated tools.

We also compare with ISEs identified by Biswas *et al.* [3], which can access architecturally visible local memory; The latter was reimplementated including speculative DMA transfer techniques to ensure coherence between the processor's data cache and the AFU's local memory, as described in [17]. For this approach, instruction identification, C code generation and ISE implementation in VHDL have been done by hand.

The proposed approached can also access architecturally visible local memory and was also implementated including speculative DMA transfer techniques as described in [17]. Instruction identification, C code generation and ISE implementation in VHDL have been done by hand but we hope to automate this processes in the future. Nevertheless, in order to make a fair comparison with other methods, we endeavored to keep generality in our manipulations in order to enable future automation.

For ISEs enhanced with architecturally visible storage, we assumed I/O constraints of 8 reads and 8 writes per cycle: each macroblock in JPEG is an 8×8 array of 16-bit integers, and can be placed into a single local memory that has 8 independent read ports and 8 independant write ports. We have implemented this local memory as a 64-entry register multi-ported register file. This was an application-specific decision that was only feasible because of the small memory size; a 1 kB memory with 8 read and 8 write ports would be prohibitive in terms of both delay and area.

The modified C programs are cross-compiled using gcc 3.4.4 based on newlib for the OpenRISC. The OpenRISC, including AFUs, is synthesized on a Xilinx Virtex II FPGA with 32 MB of external SDRAM. The performance numbers reported here are taken from the system running on the FPGA. Our experiments used a 8 kB 2-way set associative instruction cache and a 8 kB 4-way set associative data cache; both caches use the LRU replacement policy; coherence between the AFU local memory and data cache is maintained by a MESI Level 1 protocol.

VIII. Results

The complete JPEG application was run on four configurations of our soft processor platform. The baseline uses the processor with no AFUs. Then, the three ISE-based strategies outlined in the preceding section were used for comparison. RF 2-1 refers to the strategy where ISEs are identified as described by Pozzi *et al.* [23] and read their data from the processor's register file, which has 2 read ports and 1 write port. AVS 8-8 refers to the strategy where the ISEs may use architecturally visible storage in the form of local memories, as described by Biswas *et al.* [3] and a coherence protocol as described by Kluter *et al.* [17]. loop ISE AVS 8-8 refers to the strategy proposed in this paper, which uses loop ISEs with the same architecturally visible storage organization as AVS 8-8.

Fig. 5 shows the relative execution times of the three strategies listed above, normalized to software execution without ISEs. Additionally, this figure decomposes the execution time of the complete application into the execution time of each individual kernel. RF 2-1 could only speed up quantization. Our OpenRISC processor does not include a hardware divider, so the baseline implementation performs division in software. The ISE found by RF 2-1 is a hardware divider. This yields a speedup of $1.4\times$. In addition to the hardware divider, AVS 8-8 finds multi-cycle ISEs that speed up the DCT and IDCT kernels, yielding an overall speedup of $2.2\times$. loop ISE AVS 8-8 finds speedups in DCT, IDCT, quantization, and entropy encoding. The ISEs found in each kernel are different from RF 2-1 and AVS 8-8, because they are loop bodies, which include some control flow operations. Additionally, it is important to observe that RF 2-1 and AVS 8-8 achieve the same execution time for the quantization kernel, i.e., they both find the same hardware divider. loop ISE AVS 8-8 is able to find an ISE that includes local memories in addition to the divider. Although AVS 8-8 can find these types of ISEs in general, the ISE identification method estimated that control flow in the loop would lead to excessive DMA transfers, which would eliminate much of the speedup achieved by the ISE. By converting the loop to a hyperblock, loop ISE AVS 8-8 is able to find an ISE that includes local memories. Overall, the speedup achieved by loop ISE AVS 8-8 is $3.1\times$ compared to the baseline.

Fig. 6 compares the execution time of each kernel using AVS 8-8 and loop ISE AVS 8-8; the results are normalized to the former. These execution times only account for the computation time of the kernels and do not account for the overhead of DMA transfers, which occur prior to the kernel invocation; moreover, some of the DMA transfer activity may overlap with software execution of earlier parts of the application.

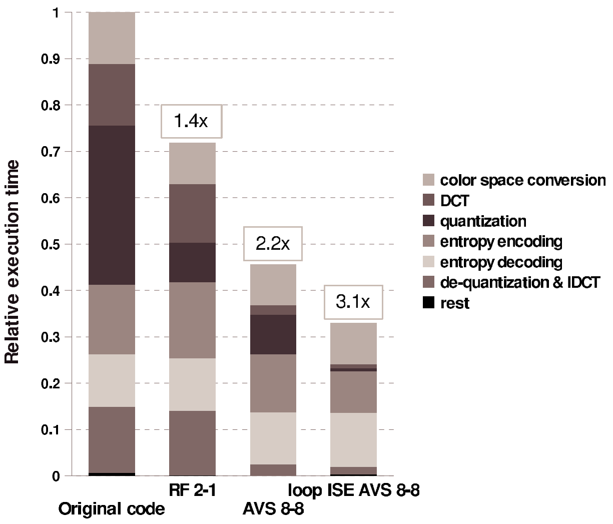


Fig. 5. Relative execution time and speed ups obtained by the different approaches in comparison with the original execution time. RF 2-1: AFU with a maximum of 2-1 inputs-output from the RF. AVS 8-8: AFU with a maximum of 8-8 inputs-outputs from local memories. loop ISE AVS 8-8: AFU with loop ISE capabilities and a maximum of 8-8 inputs-outputs from local memories. Over the complete JPEG compression and decompression algorithms we achieved a speed up of $3.1\times$ compared to a $2.2\times$ speed up achieved by the state-of-the-art.

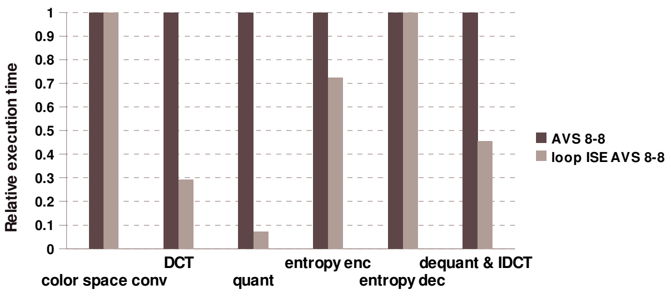


Fig. 6. Relative execution time of each of the kernels obtained by the presented method, loop ISE AVS 8-8, in comparison with the state-of-the-art, AVS 8-8. For most of the kernels, loop ISE AVS 8-8 achieves higher speed ups.

Table I shows the number of flip flops and the number of Lookup Tables (LUTs) used by the base processor and the

Strategy	N. of flip flops	N. of LUTs
base processor	10,924	21,879
AVS 8-8	11,124	29,196
loop ISE AVS 8-8	12,006	32,988

TABLE I
FPGA USAGE OF THE BASE PROCESSOR AND OF THE EXTENDED SOFT PROCESSOR WITH THE PROPOSED APPOACHED AND WITH THE STATE-OF-THE-ART.

strategies AVS 8-8 and loop ISE AVS 8-8. The base processor used 10,924 flip flops and 21,879 LUTs, while the processor extended with strategy AVS 8-8 used 11,124 flip flops and 29,196 LUTs, and the processor extended with strategy loop ISE AVS 8-8 used 12,006 flip flops and 32,988 LUTs. This shows that the area required to facilitate the AFU's control logic does not represent an important overhead in the design.

IX. Related Work

Numerous ISE identification methods have been proposed within the last decade [2], [6], [11], [23], [24], [29], [32], far too many to exhaustively enumerate here. Most notably, Biswas *et al.* [3] advocated the inclusion of local memory elements in ISEs, and Kluter *et al.* [17] described and provided a solution for the coherence problem that results from including a local memory in a processor that also contains an L1 cache. Pozzi *et al.* [24] described a method to pipeline the AFU that implements an ISE; however, their pipeline was only used to execute multi-cycle ISEs; their method does not permit the execution of multiple iterations of a loop concurrently.

Callahan and Wawrzynek [5] proposed a hyperblock loop acceleration model for use in the Garp reconfigurable coprocessor. They decomposed the runtime of two applications with two distinct data sets, into single-exit loops, multi-exit loops, hyperblock loops, unfruitful loops, and other; the hyperblock loops ranged from 9.0% to 57.6% of the different applications. The speedups achieved by implementing the different loops as hardware accelerators was not reported. Sias *et al.* [26] advocated the use of hyperblock formation to implement efficient loops in VLIW multimedia processors. The loops were stored in an external loop buffer which was distinct from the instruction cache; the use of the buffer reduces instruction fetch power and eliminates branch penalties, similar in principle to zero-overhead loops.

Our work is also closely related to the design and implementation of application-specific loop accelerators and software pipelining using hyperblocks, which were originally proposed to facilitate predicated execution.

Loop Accelerators

Behavioral synthesis for loops expressed in high-level languages is a mature field [15]. Loop accelerators are typically realized as external co-processors that are connected to the main processor by a system bus, unlike ISEs, which are more tightly coupled. Sun *et al.* in [27], propose a mixed approach to accelerate applications using a mixture of coarse-grained co-processors and fine-grained ISEs.

What sets our work apart from many prior loop accelerators is that there are no restrictions on the hardware implementations of the AFUs, beyond the I/O interface to the processor. For example, an expert arithmetic designer or arithmetically-oriented hardware synthesis tool [30] can generate whatever custom hardware is desired. Many other loop accelerator architectures, in contrast, are modeled with more traditional arithmetic units.

For example, the *Program-in-Chip-Out (PICO)* project [22] proposed a loop accelerator that consisted of a synchronous array of processor datapaths, including register files and a programmable interconnect. The transfer of data between datapaths in the accelerator is similar in principle to a transport-triggered processor architecture. Fan *et al.* [9] developed modulo scheduling and hardware synthesis methods for this type of accelerator. The problem formulation accounted for the hardware cost of the functional units, as well as storage: the output of each unit is written to a shift register, whose size is optimized during the synthesis procedure. The *Streamroller* synthesis system extended these ideas by generating multi-accelerator pipelines, connected by double buffers [18].

Shee *et al.* [25] compared several ASIP acceleration methods for JPEG, making their work quite similar to ours. They compared traditional non-loop ISEs, and two co-processors, one generated automatically from a behavioral synthesis tool, and one developed by hand. The best result was the lattermost, which achieved a speedup of $2.57\times$ over an original processor. It is difficult to compare our results directly against theirs, as the base processors are different, and they used cycle-accurate simulation while we used soft processor emulation. In particular, the differences in the ways in which the cost of off-chip SDRAM accesses were simulated and emulated are difficult to contrast.

A number of other papers have focused on the appropriate use of dependence analysis and loop optimizations, such as unrolling, pipelining, and vectorization in the context of hardware synthesis of loop-based accelerators [11], [14], [20], [31]. Our work emphasizes hyperblock formation and irregular control flow in the presence of loops, but could easily benefit from these analyses and transformations as well. We do not consider the possibility of unrolling loops prior to pipelining, and our scheduling method is not sophisticated. We readily acknowledge that more aggressive optimization and scheduling techniques could improve the quality of our loop ISEs.

Software Pipelining

Software pipelining techniques seek to improve loop performance by overlapping the execution of different iterations; typically, software pipelining is used for VLIW processors that have parallel functional units and require static scheduling at compile-time. The works most relevant to ours are those that focus on software pipelining loops that have multiple control flow paths or function calls [19], [28]. Our solution is to generate a hardware pipeline rather than a software pipeline, and to form hyperblocks to best deal with control flow within loops.

X. Conclusions

This paper has demonstrated a method to create ISEs that cover the execution of loops with exits, with the main purpose of supporting pipelined functional units. This approach broadens the scope of instruction-level parallelism for ISEs and obtains higher speed ups compared to traditional methods, primarily through pipelining, the exploitation of spatial parallelism, and reducing the overhead of control flow statements and branches. Specific examples have been analyzed in order to show the benefits of the approach. Performance improvements have been shown in the context of the JPEG application, which contains a wide variety of kernels where the technique has been applied and tested in our FPGA emulation platform.

Acknowledgment

The authors would like to thank HiPEAC network of excellence for their financial support under grant IST-217068.

References

[1] ARC International, San Jose, CA. *ARCompact-ISA Programmer's Reference Manual*, 2005.

[2] Atasu, K., Pozzi, L., and Ienne, P. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In *DAC '03: Proceedings of the 40th Conference on Design Automation* (New York, NY, USA, 2003), ACM Press, pp. 256–261.

[3] Biswas, P., Dutt, S., Pozzi, L., and Ienne, P. Introduction of Architecturally Visible Storage in Instruction Set Extensions. *IEEE Trans. on CAD of Integrated Circuits and Systems 26*, 3 (2007), 435–446.

[4] Bonzini, P., and Pozzi, L. Code Transformation Strategies for Extensible Embedded Processors. In *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (New York, NY, USA, 2006), ACM, pp. 242–252.

[5] Callahan, T. J., and Wawrzynek, J. Instruction-Level Parallelism for Reconfigurable Computing. In *In Proc. International Workshop on Field Programmable Logic* (1998), Springer-Verlag, pp. 248–257.

[6] Clark, N., Zhong, H., and Mahlke, S. Processor Acceleration Through Automated Instruction Set Customization. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), IEEE Computer Society, p. 129.

[7] Cong, J., Han, G., and Zhang, Z. Architecture and Compiler Optimizations for Data Bandwidth Improvement in Configurable Processors. *IEEE Trans. Very Large Scale Integr. Syst. 14*, 9 (2006), 986–997.

[8] De Micheli, G. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[9] Fan, K., Kudlur, M., Park, H., and Mahlke, S. Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 219–232.

[10] Gonzalez, R. E. Xtensa: a Configurable and Extensible Processor. *IEEE Micro 20*, 2 (2000), 60–70.

[11] Goodwin, D., and Petkov, D. Automatic Generation of Application Specific Processors. In *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (New York, NY, USA, 2003), ACM, pp. 137–147.

[12] Halfhill, T. R. EEMBC Releases First Benchmarks, 2000.

[13] Jayaseelan, R., Liu, H., and Mitra, T. Exploiting Forwarding to Improve Data Bandwidth of Instruction-Set Extensions. In *DAC '06: Proceedings of the 43rd Annual Conference on Design Automation* (New York, NY, USA, 2006), ACM, pp. 43–48.

[14] Jeon, J., and Choi, K. Loop Pipelining in Hardware-Software Partitioning. In *ASP-DAC* (1998), pp. 361–366.

[15] Karri, R., and Orailoglu, A. ALPS: An Algorithm for Pipeline Data Path Synthesis. In *MICRO* (1991), pp. 124–132.

[16] Karuri, K., Chattopadhyay, A., Hohenauer, M., Leupers, R., Ascheid, G., and Meyr, H. Increasing Data-Bandwidth to Instruction-Set Extensions through Register Clustering. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design* (Piscataway, NJ, USA, 2007), IEEE Press, pp. 166–171.

[17] Kluter, T., Brisk, P., Ienne, P., and Charbon, E. Speculative DMA for Architecturally Visible Storage in Instruction Set Extensions. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (New York, NY, USA, 2008), ACM, pp. 243–248.

[18] Kudlur, M., Fan, K., and Mahlke, S. Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines. In *CODES+ISSS '06: Proceedings of the 4th International Conference on Hardware/software Codesign and System Synthesis* (New York, NY, USA, 2006), ACM, pp. 270–275.

[19] Lavery, D. M., and Hwu, W.-m. W. Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 126–137.

[20] Liao, J., Wong, W.-F., and Mitra, T. A Model for Hardware Realization of Kernel Loops. In *FPL* (2003), pp. 334–344.

[21] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., and Bringmann, R. A. Effective Compiler Support for Predicated Execution Using the Hyperblock. *SIGMICRO Newsl. 23*, 1-2 (1992), 45–54.

[22] Mahlke, S. A., Ravindran, R. A., Schlansker, M. S., Schreiber, R., and Sherwood, T. Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators. *IEEE Trans. on CAD of Integrated Circuits and Systems 20*, 11 (2001), 1355–1371.

[23] Pozzi, L., Atasu, K., and Ienne, P. Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 25*, 7 (July 2006), 1209–1229.

[24] Pozzi, L., and Ienne, P. Exploiting Pipelining to Relax Register-File Port Constraints of Instruction-Set Extensions. In *CASES '05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (New York, NY, USA, 2005), ACM, pp. 2–10.

[25] Shee, S. L., Parameswaran, S., and Cheung, N. Novel Architecture for Loop Acceleration: a Case Study. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis* (New York, NY, USA, 2005), ACM, pp. 297–302.

[26] Sias, J. W., Hunter, H. C., and Hwu, W.-m. W. Enhancing Loop Buffering of Media and Telecommunications Applications Using Low-Overhead Predication. In *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 262–273.

[27] Sun, F., Ravi, S., Raghunathan, A., and Jha, N. A Synthesis Methodology for Hybrid Custom Instruction and Coprocessor Generation for Extensible Processors. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 26* (Nov. 2007), 2035 – 2045.

[28] Tirumalai, P., Lee, M., and Schlansker, M. Parallelization of Loops with Exits on Pipelined Architectures. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 1990), IEEE Computer Society, pp. 200–212.

[29] Verma, A. K., Brisk, P., and Ienne, P. Rethinking Custom ISE Identification: a New Processor-Agnostic Method. In *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (New York, NY, USA, 2007), ACM, pp. 125–134.

[30] Verma, A. K., Brisk, P., and Ienne, P. Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems 27*, 10 (2008), 1761–1774.

[31] Weinhardt, M., and Luk, W. Pipeline Vectorization. *IEEE Trans. on CAD of Integrated Circuits and Systems 20*, 2 (2001), 234–248.

[32] Yu, P., and Mitra, T. Scalable Custom Instructions Identification for Instruction-Set Extensible Processors. In *CASES '04: Proceedings of the 2004 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (New York, NY, USA, 2004), ACM, pp. 69–78.