Automatic Architectural Design of Wide-Issue Counterflow Pipelines

Bruce R. Childers, Jack W. Davidson
University of Virginia, Department of Computer Science
Thornton Hall, Charlottesville, Virginia 22901
{brc2m, jwd}@cs.virginia.edu

Abstract

Application-specific instruction set processor (ASIP) design is a promising approach for meeting the performance and cost goals of a system. ASIPs are especially promising for embedded systems (e.g., digital cameras, cellular phones, etc.) where a small increase in performance and decrease in cost can have a large impact on a product's viability. Sutherland, Sproull, and Molnar have proposed a novel pipeline organization called the Counterflow Pipeline (CFP) that is appropriate for ASIP design. We have extended the CFP to a wide-issue architecture for designing custom instruction-level parallel (ILP) processors. In this paper, we first outline our extensions to the CFP to support automatic design of application-specific ILP processors. Second, we present a design system that automatically customizes a wide-issue counterflow pipeline (WCFP) to the resource and data flow requirements of a software pipeline loop. Finally, we show that custom WCFP's achieve cycles per operation measurements that are competitive with custom VLIW organizations at a lower design complexity.

1. Introduction

Application-specific instruction set (ASIP) processor design is a promising approach for improving the cost-performance ratio of an application. ASIPs are especially useful for embedded systems (e.g., digital cameras, cellular phones, etc.) where a small increase in performance and decrease in cost can have a large impact on a product's viability. A novel computer organization called the *Counterflow Pipeline* (CFP), proposed by Sproull, Sutherland, and Molnar [14], has several characteristics that make it an ideal target organization for the synthesis of custom instruction-level parallel (ILP) processors. The CFP has a simple and regular structure, local control, high degree of modularity, asynchronous implementations, and inherent handling of complex structures such as register renaming and speculative execution.

Our research uses an application expressed in a high-level language as a specification for an embedded processor. By using aggressive hardware/software co-design techniques, ASIP synthesis lets an expert specify a system without consideration for low-level implementation details. This design style may significantly reduce development time and cost because fewer people are needed and more design trade-offs may be explored in a limited time than in a traditional work flow [11].

The counterflow pipeline is a good candidate for this type of fast, aggressive synthesis because of its superior composability and simplicity. This substantially reduces the complexity of synthesis because a CFP synthesis system does not have to design control paths, determine complex bus and bypass networks, etc.

Our previous work discusses the advantages and disadvantages of application-specific CFP's [3]. This paper briefly introduces a new wide-issue CFP, which we call a *wide counterflow pipeline* (WCFP), that is appropriate for the automatic design of ILP-processors. In this paper, we present a simple methodology for building and evaluating custom WCFP's. The paper shows that custom WCFP's achieve performance that is competitive with custom VLIW processors that have similar resources without requiring expensive global communication.

1.1. Design Strategy

Most high-performance embedded applications have two parts: a control and a computation-intensive part. The computation part is typically a kernel loop that accounts for the majority of execution time. Increasing the performance of the most frequently executed portion of an application increases overall performance. Thus, synthesizing custom hardware for the computation-intensive portion of an application is an effective technique to improve performance.

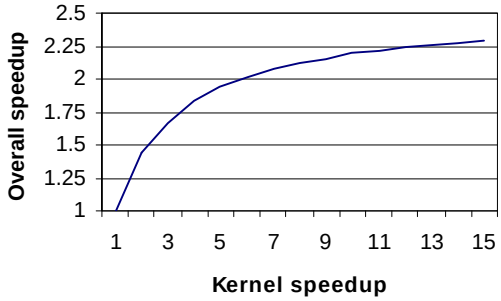

Figure 1: Overall speedup for JPEG.

We are considering applications that need only a modest kernel speedup to effectively improve overall performance. For example, JPEG has a function `j_rev_dct()` that accounts for 60% of total execution time. This function applies a single loop twice (to do the inverse discrete cosine transformation), so it is a good candidate for a custom CFP. Using Amdahl's Law, Figure 1 shows that a small speedup of `j_rev_dct()` of 6 or 7 achieves most of the overall speedup.

Our synthesis system targets an architecture that has two processors: a conventional architecture for executing control code and a WCFP for executing the kernel loop of an application. Our work customizes a WCFP to the kernel computation for improved performance.

For applications where there is not a clearly identifiable kernel, the above strategy will not be as effective. However, most applications we have examined have execution profiles similar to JPEG—one kernel that consists of over 50% of the overall execution of the application. We profiled several applications from the MediaBench benchmark suite[10] and found that most of these applications had a single function with a kernel loop that accounted for the majority of execution time.

2. Wide Counterflow Pipelines

This section presents a brief overview of WCFPs—a detailed discussion of the organization is in [4]. We have extended the original CFP [14] to an architecture that issues several operations per instruction to exploit instruction-level parallelism in kernel loops.

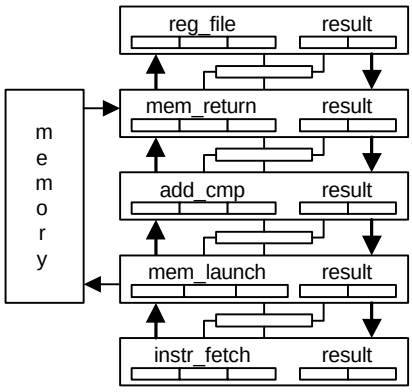

Figure 2: An example WCFP.

The WCFP has two pipelines flowing in opposite directions as shown in Figure 2. One is the instruction pipeline, which carries instructions from a fetch stage to a register file stage. When an instruction issues, an *instruction bundle* is formed that flows through the pipeline. The instruction bundle holds the instruction opcode, operand names, and operand values. The other pipeline is the results pipeline that carries results from the register file to the fetch stage. Whenever a value is inserted in the result pipeline, a *result bundle* is created that holds a result's register name and value.

The instruction fetch stage decodes and issues instructions and creates their instruction bundles. It also discards results from the pipeline. The register file holds destination values of instructions that have exited the pipeline. It is updated with an instruction's destination register whenever an instruction enters the stage.

The WCFP has pipelined functional units called *sidings* that execute instructions. Sidings are connected to the processor through *launch* and *return* stages, which initiate siding operations and return values from sidings. Figure 2 shows an example siding for memory that is connected to the pipeline by `mem_launch` and `mem_return`. Instructions may also execute in a pipeline stage of an appropriate type without using a siding.

The instruction and result pipelines interact: instructions copy values to and from the result pipeline. This interaction is governed by rules that ensure sequential execution. There are also rules that ensure result values are current for their position in the pipeline and not values from previous operations that use the same registers.

WCFP instructions have groups of operations that are issued together. Restrictions about which operations may be scheduled together are determined by the operation repertoire of functional devices. For example, issuing two loads together requires a memory unit that does two simultaneous reads.

The operations in an instruction move through a WCFP in lock-step, although they may execute in different stages or sidings. Doing operations in separate stages lets them execute at the best point in the pipeline. From extensive experimentation, we have found that the ideal location to execute an operation is usually in the stage immediately after the point where the operation acquires its last source operand. Because operations in an instruction may garner their operands in different stages and become ready to execute at different times, the location where each operation executes should be tailored to the dynamic data flow of the application.

As Figure 2 shows, WCFP's have local communication: functional devices communicate only with their neighbors. This has two advantages. First, architectural synthesis does not need to determine functional unit interconnection; it is implicit in the pipeline stage order. Second, purely local device communication may lead to very fast implementations, especially as global wire delays begin to dominate critical path latencies.

3. Automatic Pipeline Design

The WCFP customization process operates at the architectural-level on pre-designed functional devices such as pipeline stages, register files, and functional sidings. The design space of WCFP's is defined by processor functionality and topology. Processor functionality is the type and number of devices in a pipeline and topology is the interconnection of those elements. Processor functionality is characterized by an user-supplied *design database* of computational elements that indicates device type (siding or stage) and semantics (as instruction opcodes) for each database entry. WCFP topology is determined by the order of pipeline stages because WCFP functional devices are interconnected via stages.

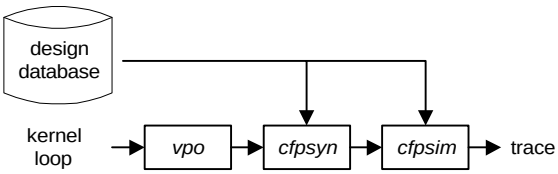

Figure 3: *vpo* optimizes a kernel loop, which serves as a specification for a WCFP determined by *cfpsyn*. The WCFP is simulated and analyzed by *cfpsim*.

Figure 3 is a diagram of the design system. The system accepts an application program (in C) with its kernel loop annotated as an input to the code optimizer *vpo* [1], which compiles the application using optimizations such as strength reduction, induction variable elimination, global register allocation, etc. Other optimizations are done by hand prior to synthesis, including scalar replacement and if-conversion.

vpo passes the optimized kernel loop to the synthesis phase, *cfpsyn*, which selects and instantiates computational devices from the design database and derives pipeline stage order. Synthesis emits a description of a WCFP for the simulator, *cfpsim*, which collects performance statistics and an execution trace.

3.1. Pipeline Customization

The optimized instructions emitted by *vpo* and the design database are used to derive a custom WCFP. Figure 4 shows the four steps of synthesis, which are described in the following sections.

Software Pipelining. The first step of WCFP synthesis generates a software pipelined loop from the instruction

```
genCounterflowPipeline(SynthesisDB DB, Loop L) {
  // step 1: form the software pipeline kernel
  PipelineKernel kernel ← moduloSchedule(L);
  // step 2: generate wide counterflow pipeline
  WideCounterflowPipeline WCFP ←
      genPipeline(kernel, DB);
  // step 3: generate instruction set architecture
  InstructionSet ISA ← generateInstrSet(WCFP);
  // step 4: form kernel and emit ISA & WCFP
  InstrSched sched ← softwarePipeline(kernel, ISA);
  emit(sched, WCFP); }
```

Figure 4: A software pipeline is formed that acts as a specification for a WCFP and instruction set.

sequence emitted by *vpo* using a version of iterative modulo scheduling [12]. The resource constraints modeled by the software pipelining step are the operation slots in an instruction. The number of operation slots is supplied by the user. The design database determines scheduling latencies and what operations may appear together in an instruction by specifying a constraint for each database entry that indicates the number of siding operations of a particular type that can be initiated in an instruction. Using this information, a software pipeline is formed that serves as a specification for a WCFP.

Pipeline Extraction. The software pipeline kernel is used to generate a WCFP and to specify the operations the processor supports. The current system assigns every operation in the kernel a pipeline stage. A functional siding is also assigned for high latency operations.

A WCFP is formed by iterating over instructions in the pipeline kernel to partition instruction operations into groups that represent the functionality a custom WCFP supports. Groups are formed based on latency: low latency operations are grouped together to generate a single execution stage and high latency operations are grouped with similar operations from the same instruction to generate functional sidings. For low latency operations, there is always a single group for an instruction, and for high latency operations there may be multiple groups for each distinct operation class.

After operations are grouped, the groups are used to derive pipeline stages and sidings that have the semantics required by each group. Currently, an unique execution stage is created for every low latency operation group. A functional siding and return and launch stages are created for a high latency operation group if a siding with the required semantics does not already exist. This ensures only one siding of a particular type is ever created (e.g., there is one multiplier) because these devices are expensive and execute several operations in at once.

Pipeline stages are inserted in the order of the software pipeline kernel, beginning with instruction fetch. This order allows multiple kernel iterations to be present in the pipeline at the same time, which lets one iteration complete while another is speculatively issued.

Because the WCFP is composable and has local communication, a simple "building block" strategy as described above may be used. In traditional architectures, the interconnection of functional elements must be considered. This makes synthesis much more complicated, especially for VLIW organizations which typically have fully interconnected communication paths.

Instruction Set Extraction. The synthesis system determines an instruction set for a WCFP. This includes assigning opcodes to pipeline operations, identifying status information kept for every operation, forming instruction fields, and canonicalizing operation order in an instruction. It also creates an intermediate representation of a WCFP's instruction set that is used during code generation to emit WCFP instructions.

Code Generation. The final step of WCFP synthesis is code generation, which forms the complete instruction schedule using the software pipeline kernel. The first step of code generation is *modulo variable expansion* (MVE), which eliminates dependence conflicts on registers whose lifetimes are greater than the software pipeline initiation interval using static register renaming.

After doing MVE, prologue and epilogue code is generated to begin and end the software pipeline. The prologue and epilogue code requires that the number of iterations of the pipeline kernel be $(s - F) + F \times i$, where s the iteration span of the kernel, F is the loop unroll factor computed by MVE, and $i \geq 0$. The trip count of the original loop must be adjusted to fit this equation by executing the original loop a limited number of times until the iteration count matches the requirements of the software pipeline. This is called loop preconditioning. Our system does preconditioning on the control processor and transfers control to the WCFP after the preconditioning loop completes execution.

The code generation step also generates start-up and tear-down code that is stitched into the application in place of the original kernel loop. This code initiates and finishes WCFP processing by initializing loop live-in registers and copying live-out registers from the WCFP.

4. Experimental Results

We are studying how far WCFPs can be pushed with minimal microarchitecture changes to get good performance in an application-specific setting. To that end, the experiments in this paper use WCFPs customized to several benchmark applications. The benchmarks have integer versions of Livermore loops number 1 (*k1*), 5 (*k5*), 7 (*k7*) and 12 (*k12*), the finite impulse response filter (*fir*), vector dot product (*dot*), and other kernels extracted from large applications. These loops include the discrete cosine transformation (*dct*) and Floyd-Steinberg image dithering algorithm (*dither)*. We also extracted the vector computation $a = b^k \bmod d$ from RSA encryption (*mexp*). The final benchmark comes from the GSM 6.10 standard for speech decoding (*gsm*).

We have built a reconfigurable simulator for asynchronous WCFP's that generates an execution trace for post-mortem performance analysis. The WCFP simulator models asynchronous pipelines by varying computational latencies. To move an instruction or result between stages takes 1 time unit, to garner a result takes 3 time units, and to launch or return an instruction from a siding takes 3 time units. To execute an operation such as an addition takes 5 time units. High latency operations are scaled relative to low latency ones. For example, multiplication, assuming it is four times slower than addition, takes 20 time units.

4.1. Wide Counterflow Pipelines

To measure the effectiveness of custom WCFP's, we use *effective cycles per operation* (ECPO). Because an asyn-

chronous WCFP implementation is used, the execution latency must be normalized by an *effective clock cycle* (ECC) to derive ECPO measurements. Given ECC, *ECPO = (latency/ECC)/instruction count*.

We use an ECC of 8 because it takes 3 time units to garner a source operand and 5 time units to execute an instruction. A synchronous CFP must perform at least these two operations in a single clock cycle. An ECC of 8 is conservative because simple CFP structures should lead to very fast effective clock cycle speeds.

We have used our design system to generate custom pipelines for the benchmarks above. The design database used to synthesize these custom pipelines have sidings that do two operations of a similar type at once. The ECPO measurements for these custom pipelines range from 0.25 to 1.17. These measurements show that WCFP's effectively exploit ILP in the benchmark loops.

Pipeline Refinement. Although the pipelines generated by our design methodology effectively exploit ILP, the order of pipeline stages could be modified to match a loop's execution behavior to get better performance.

As we have shown elsewhere [2], there are several factors that affect the performance of CFPs, such as the distance results flow between their producer and consumer operations, the number of instructions between producer and consumer, overlapping the movement of results and instructions in the pipeline, and balancing the pipeline using the execution characteristics of the application. The impact of these factors may be mitigated by arranging the order of pipeline stages to match the dynamic flow of results and instructions.

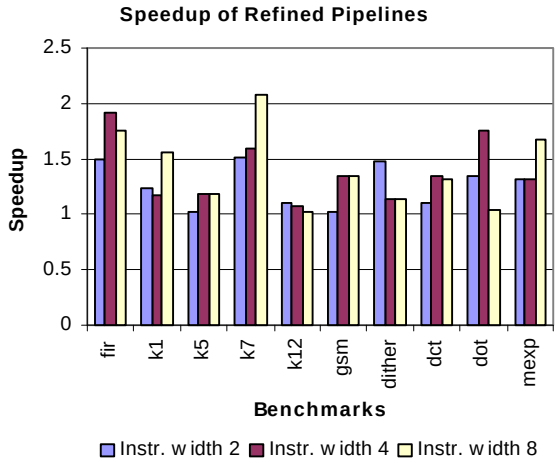

Figure 5: Performance improvement of refined WCFPs.

Figure 5 shows the speedup of pipelines whose stage orders have been refined to match the execution behavior of each benchmark. The figure demonstrates that pipeline refinement is important for good performance. In all cases, performance was improved, with speedup ranging from 1.03 to 2.16 and an average of 1.38.

A simple methodology is used to refine a WCFP's stage order. The technique identifies the ideal stage position to execute an operation in the pipeline. The best location is a place where the operation has garnered its source operands and the operation's execution can be overlapped with the execution of some other instruction.

The refinement process selects each pipeline stage in turn to be moved to a new location. After selecting a stage, an execution trace is generated and examined to determine where the operation that executes in the selected stage becomes ready to execute. There are two possibilities: 1) the operation is ready in a stage *before* the selected stage, or 2) the operation is ready *at* the selected stage (an operation stalls in its execution stage if it has not garnered all of its source operands).

In the first case, the selected stage is moved to a location after the point at which the operation becomes available. The exact position is selected by scanning the execution trace to identify a place in the pipeline where the operation will stall while waiting for some other preceding operation to advance. This moves the selected stage to a point where the execution latency of the operation is masked by the pipeline delay. This point also ensures that succeeding instructions advance until the last possible stage before stalling.

In the second case, the stage is moved late in the pipeline and a new execution trace collected. After collecting the new trace, it is likely that the selected stage's operation will garner its source operands before reaching the selected stage. Thus, the first case applies and the stage is moved to an appropriate pipeline position.

Care must be taken when moving a stage that is between launch and return stages. When a siding is generated by the synthesis system, its launch and return points are separated by enough stages to match the depth of the siding. Pipeline refinement maintains this distance by inserting "empty stages" in the place of a stage moved from between a siding's launch and return point. An empty stage is a place-holder in the pipeline and does no processing. In the special case when a stage is moved to a position occupied by an empty stage, the empty stage is replaced by the new stage.

After moving a stage, a new execution trace is collected. If performance degrades, the process reverts to the previous pipeline and tries moving a different stage. The entire process is repeated for every pipeline stage until there is no further performance improvement for all stages. The refinement process traverses the design space of stage orders to find one that has good performance. This method has the limitation that it moves one stage at a time and only keeps a move if performance improves. However, it may be worthwhile to make a bad move, which may subsequently expose a good move.

The simple refinement methodology described in this section is effective and important for achieving good performance from the WCFP. The WCFP's composability makes such a process practical; it is not apparent how to do this with a traditional microarchitecture.

VLIW vs. WCFP. Figure 6 shows a comparison of traditional VLIW architectures versus custom WCFPs. The data in the figure was collected using the same software pipeline for the VLIW and the WCFP, ensuring a fair comparison between the two organizations using the same instruction schedule. This also ensures that the VLIW and the WCFP have the same resources.

The figure shows a comparison of ECPO for the VLIW and WCFP processors with varying instruction widths. Figure 6 shows that WCFP's achieve ECPO measurements that are competitive with traditional VLIWs with similar resources. The WCFP has performance within 0-18% of the traditional VLIW architecture (the average is 8.6%), and in most cases, the

performance is within 7% of the VLIW. For some cases, the WCFP does better than the VLIW. Indeed, for the *k5* benchmark, the WCFP is 22% better. This benchmark has a good balance between the type of operations and the flow of instructions and results, which led to peak performance with minimal pipeline stalls.

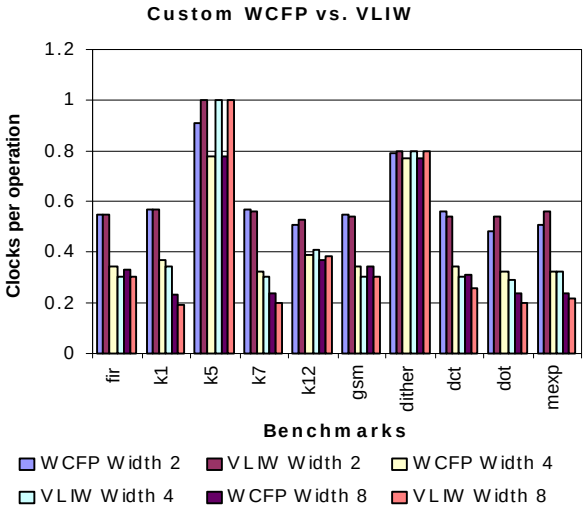

Figure 6: A comparison of cycles per operation for custom WCFP and VLIW organizations.

Although performance is good for most benchmarks, the wider designs have a greater relative difference in performance between the WCFP and VLIW architectures than for less aggressive designs. This is influenced by partitioning operations into groups. For some benchmarks, like *dct* and *dot*, large groups may adversely affect performance because all operations in a group must acquire their source operands prior to any operation executing. This may delay the execution of some operations that become ready very early. We modified *dct* and *dot* to decouple the execution of an instruction's operations. This initial work shows that decoupled execution improves performance significantly and makes *dct* and *dot* competitive with traditional VLIWs.

Unlike traditional VLIWs, functional elements in the WCFP communicate only with their neighbors, and as global wire delays dominate circuit latency, the WCFP's local communication should lead to fast implementations. However, the comparator network (for operand garnering) in a pipeline stage may lessen the performance gained from local communication, especially as pipeline width increases. In the future, we will address these issues and their impact on performance and cost.

4.2. Related Work

There is much interest in automated design of ASIPs because of the increasing importance of high performance and quick turn-around in embedded systems. ASIP techniques address two problems: instruction set and microarchitecture synthesis. Instruction set synthesis identifies micro-operations in a program that may be combined to create instructions that are optimized for program size, power consumption, execution latency, etc. [9]. Microarchitecture synthesis derives a processor from an application by either synthesizing a co-processor for a portion of an application and integrating it with an embedded core [6] or by tailoring a single processor to the entire application [5, 7]. Finally, many co-design systems unify instruction set and microarchitecture synthesis in a single framework [8].

5. Summary

In this paper, we briefly outline extensions to the original counterflow pipeline processor to make it appropriate for the automatic design of application-specific processors. We also present a simple technique for customizing WCFP's to an application and show that custom WCFP's are performance competitive with traditional VLIW organizations at a low design cost.

References

[1] Benitez M. E. and Davidson, J. W., "A portable global optimizer and linker", *Symp. on Programming Lang. Design and Implementation*, pp. 329–338, Atlanta, GA, June 1988.

[2] Childers B. R. and Davidson J. W., "A design environment for counterflow pipeline synthesis", *Workshop on Lang., Compilers, and Tools for Embedded Systems*, Montreal, Canada, June 19–20, 1998.

[3] Childers B. R. and Davidson J. W., "Architectural considerations for application-specific counterflow pipelines", *20th Anniversary Conf. on Adv. Research in VLSI*, Atlanta, GA, March 21–23, 1999.

[4] Childers B. R. and Davidson J. W., "Application-specific wide-issue counterflow pipelines", Technical Report CS–99–02, Computer Science, University of Virginia, Charlottesville, VA, Jan. 1999.

[5] Corporaal, H. and Hoogerbrugge J., "Cosynthesis with the MOVE framework", *Symp. on Modelling, Analysis, and Simulation*, pp. 184–189, Lille, France, July 1996.

[6] Ebeling C., Cronquiest D. C., Franklin P., et al., "Mapping applications to the RaPiD configurable architecture", *5th Symp. on Field-Programmable Custom Computing Machines*, pp. 106-115, Napa Valley, CA, April 16–18, 1997.

[7] Fisher J. A., Faraboschi P., and Desoli G., "Custom-fit processors: Letting applications define architectures", Technical Report HPL–96–144, HP Labs, Palo Alto, CA, 1996.

[8] Gupta R. K., and Micheli G., "Hardware-software co-synthesis for digital systems", *IEEE Design and Test of Computers*, Vol. 10, No. 3, pp. 29–41, Sept. 1993.

[9] Huang I-J and Despain A. M., "Synthesis of application specific instruction sets", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol 14, No. 6, pp. 663–675, June 1995.

[10] Lee C., Potkonjak M., Magione-Smith W. H., "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems", *30th Int'l Symp. on Microarchitecture*, pp. 330–335, Research Triangle Park, NC, Dec. 1997.

[11] Lin Y-L., "Recent developments in high-level synthesis", *ACM Trans. on Design Automation of Electronic Systems*, pp. 2–21, Vol. 2, No. 1, Jan. 1997.

[12] Rau B. R., "Iterative modulo scheduling: An algorithm for software pipelined loops", *27th Int'l Symp. on Microarchitecture*, pp. 63–74, Dec.1994, San Jose, CA.

[13] Schlett M., "Trends in embedded microprocessor design", *IEEE Computer*, pp. 44–49, Aug. 1998.

[14] Sproull R. F., Sutherland I. E., and Molnar C. E., "The counterflow pipeline processor architecture", *IEEE Design and Test of Computers*, pp. 48–59, Vol. 11, No. 3, Fall 1994.