Automatic Design of Custom Wide-Issue Counterflow Pipelines

Bruce R. Childers[†], Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

Telephone: 804-982-2292, Fax: 804-982-2214

{brc2m[†],jwd}@cs.virginia.edu

[†]Corresponding author.

Abstract

Application-specific processor design is a promising approach for meeting the performance and cost goals of a system. Application-specific integrated processors (ASIP's) are especially promising for embedded systems (e.g., automobile control systems, avionics, 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 new pipeline organization called the Counterflow Pipeline (CFP) that may be appropriate for ASIP design. This paper extends the original CFP microarchitecture to a very long instruction word (VLIW) organization for the custom design of instruction-level parallel processors tailored to the requirements of computation-intensive inner loops. First, we describe our extensions to the CFP to support automatic customization of application-specific VLIW processors. Second, we present an automatic design system that customizes a wide-issue counterflow pipeline (WCFP) to the resource and data flow requirements of a software pipeline loop. Third, we show that custom asynchronous WCFP's achieve cycles per operation measurements that are competitive with custom VLIW organizations at a potentially low design complexity. Finally, the paper describes several enhancements that can be made to WCFP's to further improve performance of kernel loops.

Automatic Design of Custom Wide-Issue Counterflow Pipelines

Bruce R. Childers, Jack W. Davidson
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
{brc2m, jwd}@cs.virginia.edu

Abstract

Application-specific processor design is a promising approach for meeting the performance and cost goals of a system. Application-specific integrated processors (ASIP's) are especially promising for embedded systems (e.g., automobile control systems, avionics, 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 new pipeline organization called the Counterflow Pipeline (CFP) that may be appropriate for ASIP design. This paper extends the original CFP microarchitecture to a very long instruction word (VLIW) organization for the custom design of instruction-level parallel processors tailored to the requirements of computation-intensive inner loops. First, we describe our extensions to the CFP to support automatic customization of application-specific VLIW processors. Second, we present an automatic design system that customizes a wide-issue counterflow pipeline (WCFP) to the resource and data flow requirements of a software pipeline loop. Third, we show that custom asynchronous WCFP's achieve cycles per operation measurements that are competitive with custom VLIW organizations at a potentially low design complexity. Finally, the paper describes several enhancements that can be made to WCFP's to further improve performance of kernel loops.

1. Introduction

Application-specific processor design is a promising approach for improving the cost-performance ratio of an application. Application-specific processors are especially useful for embedded systems (e.g., automobile control systems, avionics, cellular phones, etc.) where a small increase in performance and decrease in cost can have a large impact on a product's viability. A new computer organization called the *Counterflow Pipeline* (CFP), proposed by Sproull, Sutherland, and Molnar [29], has several characteristics that make it an ideal target organization for the synthesis of application-specific 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.

General-purpose processors are good for average workloads, as typified by popular benchmark suites such as SPEC'95. These architectures are optimized to execute applications containing a set of common operations and exhibiting similar behavior (e.g., such as the frequency of control-transfer operations). The ever increasing demand for performance and the need to handle *arbitrary* code has lead to very complex and costly general-purpose microarchitectures. For example, the 4-way superscalar HP PA-8000 microprocessor [15] is optimized for exploiting instruction-level parallelism (ILP) across a variety of application types; this goal requires the PA-8000 to tolerate high-latency operations such as memory accesses and the frequent presence of control transfer operations. To keep such an aggressive design busy requires large instruction windows and complex structures (e.g., register rename buffers, data prefetching support) to overcome high-latency operations and control dependences. The PA-8000 does this with a 56-entry instruction re-order buffer, data prefetch instructions, predicated execution, and branch prediction and history tables.

The cost of high-performance general-purpose processors is often prohibitive for many embedded applications; however, many such applications would benefit from this level of performance. An alternative to an architecture such as the PA-8000 is a custom processor matched to an application's performance and cost goals. A custom architecture has the flexibility to include the minimal instruction set and microarchitecture elements that give good performance and low cost for a *single* code without the complexity of devices for general-purpose codes. Because cost and time-to-market constraints are very important to embedded systems [28], an ASIP architecture should permit automatic design, including high-level architectural design. Such an architecture and design system is the focus of this paper.

Our research uses an application expressed algorithmically in a high-level language as a specification for an emedded microprocessor. By using aggressive hardware/software co-design techniques, an ASIP synthesis system lets an expert specify a system without consideration for low-level implementation details. This design style can lead to a significant reduction in a product's development cycle and cost because fewer people are needed and more design trade-offs can be explored in a limited time than in a traditional work flow [19].

The counterflow pipeline is a good candidate for this type of fast, aggressive synthesis because of its extreme 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. The modularity of the CFP is also an enabling technology: it is easy to incorporate functional devices for the specific needs of an application (e.g., including a multiply-accumulate unit) and to explore design trade-offs such as the number and type of functional units.

Our previous work discusses the advantages and disadvantages of application-specific CFP's [4]. This paper builds on that work and proposes a new wide instruction word CFP, which we call a *wide counterflow pipeline* (WCFP), that is appropriate for the automatic design of statically scheduled ILP-processors. We also present a simple and effective design methodology for automatically building and evaluating application-specific VLIW counterflow pipelines. The paper demonstrates that custom WCFP's achieve performance that is competitive with custom VLIW processors.

This paper is organized as follows. The first section has introductory material about our design strategy and the counterflow pipeline. The second section describes our extensions to the original CFP to make it a VLIW architecture appropriate for application-specific processor design. The third section presents a simple and effective methodology for deriving WCFP's tailored to the requirements of an application's kernel loop. The fourth section contains experimental results that indicate asynchronous WCFP's achieve performance on par with custom VLIW organizations. This section also suggests several enhancements that may further improve performance. The fifth section has related work and the final section concludes the paper.

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 may be an effective technique to increase performance.

The type of applications we are considering 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 consists of applying a single loop twice (to do the inverse discrete cosine transformation), so it is a good candidate for a custom counterflow pipeline. Figure 1 shows a plot of Amdahl's Law for various speedup values of `j_rev_dct()`. The figure shows that a small speedup of the kernel loop of 6 or 7 achieves most of the overall speedup.

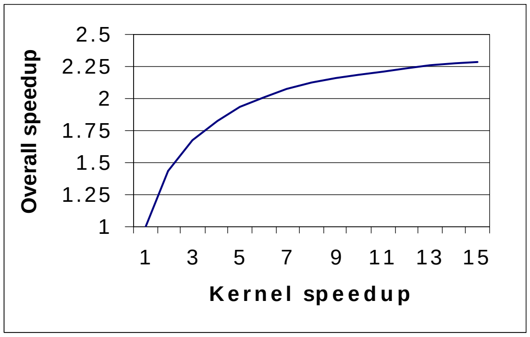

Figure 1: Overall speed-up for JPEG

We use the dependency graph of an application's kernel loop to determine processor functionality and interconnection network. Processor functionality is determined from the type of operations in the graph and processor interconnection is determined by exploring the design space of all possible interconnection network.

The target system architecture for our synthesis technique has two processors: a traditional processor for executing control code and a WCFP processor for executing the computation portions 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 (a collection of popular multimedia applications) [18] and found that most of these applications had a single loop that accounted for the majority of execution time. These applications included programs for GSM 6.10 full rate speech coding, adaptive differential pulse code modulation (ADPCM), image compression/decompression (EPIC), MPEG-III audio playback (not included in MediaBench), and CCITT G.721 voice compression/decompression. For these benchmarks, the kernel computation accounted for 53% to 85% of execution time. In some cases, the loop had a "helper" function which could be inlined by an optimizing compiler to create a self-contained computation kernel (i.e., one that does not have any function calls). In this paper we consider only the requirements of kernel loops.

1.2. Counterflow Pipeline

This section presents a brief overview of the original counterflow pipeline proposal. Sproull, Sutherland, and Molnar given a more detailed discussion of CFP's [29].

The CFP has two pipelines flowing in opposite directions as shown in Figure 2. One is the instruction pipeline, which carries instructions from an instruction fetch stage to a register file stage. When an instruction issues, an *instruction bundle* is formed that flows through the pipeline. The instruction bundle has space for the instruction opcode, operand names, and operand values. The other pipeline is the results pipeline that conveys results from the register file to the instruction fetch stage. Whenever a value is inserted in the result pipeline, a *result bundle* is created that holds a result's name (i.e., 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 register values of instructions that have exited the pipeline. It is updated with an instruction's destination register whenever an instruction enters the stage.

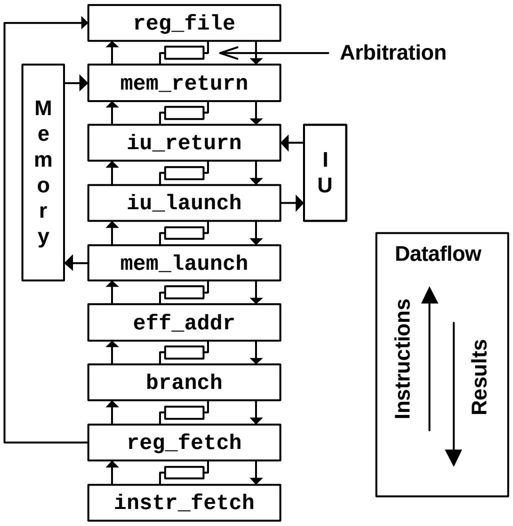

Figure 2: An example counterflow pipeline

The CFP 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 (e.g., an addition stage) 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 semantics. There are four rules that control instructions: 1) instructions must stay in sequential order; 2) an instruction must acquire its source operands prior to executing (called *garnering*); 3) an instruction inserts a copy of its destination register in the result pipeline when it finishes executing (called *updating*); and 4) an unexecuted instruction may not move past the last stage capable of executing it.

There are three other rules that ensure result values are current for their position in the pipeline and not values from previous operations with the same name. The rules are: 1) an instruction copies a result's value if a result register name matches one of its source operands; 2) a result register matching an unexecuted instruction's destination register is invalidated; and 3) a result register matching an executed instruction's destination is updated with the destination's value.

Arbitration is required between stages so that instruction and result bundles do not pass each other without a comparison made on their operand names. In Figure 2, the blocks between stages depict arbitration logic.

A final mechanism called a *poison pill* purges the pipeline when a branch is mispredicted (it is also used for exception handling). A poison pill is a special result that is inserted by a branch stage on a misprediction. A poison pill kills instructions it encounters while flowing through the result pipeline, and for branches, this removes speculatively issued instructions.

As Figure 2 shows, stages and functional units are connected in a very simple and regular way and the behavior of a stage is dependent only on the adjacent stage in the pipeline, which permits local control of stages and avoids the complexity of conventional pipeline synchronization.

2. Wide Counterflow Pipelines

We have extended the original counterflow pipeline organization to a VLIW microarchitecture. This new architecture, which we term a *wide counterflow pipeline* (WCFP), issues several statically scheduled operations per instruction to exploit instruction-level parallelism in kernel loops.

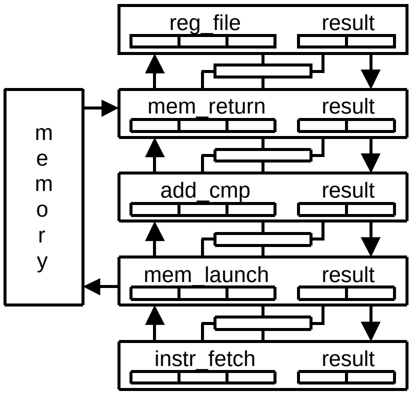

Figure 3: A WCFP that issues two operations per instruction.

An example WCFP is shown in Figure 3 that issues three operations per instruction. In this example, the width of the result pipeline is matched to the maximum width of an instruction (in this example, the width is 2 since the `add` and `cmp` stage generates two results).

2.1. Wide instructions

WCFP instructions have groups of operations that are issued simultaneously. There are no rules for what operations may be scheduled in an instruction; however, every operation in an instruction must use a different destination register to ensure correct execution (i.e., there is no dependence checking between operations in an instruction—the WCFP does, however, enforce dependences between instructions). Restrictions such as issuing multiple memory operations together are determined by the operation repertoire of functional devices. For example, issuing two loads together requires a memory unit with an operation for doing two memory accesses simultaneously.

The operations in an instruction move through a WCFP in lock-step, although they may execute in different stages or sidings. For example, the operations in the instruction:

```
 (ld [r5],r6; add r7,1,r7; cmp r10,0)
```

execute in two stages in Figure 3. The load is started in `mem_launch` and finished in `mem_return`, and the addition and comparison are done in `add_cmp`.

Doing operations in separate stages lets them execute at the most appropriate point in the pipeline. From extensive experimentation, we have found that the best location for an operation to execute is usually in the stage immediately after the point where the operation acquires its last source operand. Because individual 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 data flow behavior of the application to significantly improve performance.

2.2. Pipeline Stage

Figure 4 shows a block-level diagram of a wide counterflow pipeline execution stage. Conceptually, an execution stage has three processes: 1) *garner*, 2) *execute*, and 3) *control*. The garner process checks for matching register names in an instruction and result bundle. If a match is found, the instruction and result are updated according to the pipeline rules from Section 1.2.

The execute process checks whether an instruction executes in the stage. If it does, it waits until the operation it executes has all of its source operands before doing the operation. After executing the operation, a new result is generated and interested into the result pipeline.

The control process monitors the activity of a stage and indicates to adjacent arbitration units whether it is ready to advance an instruction or result, or accept a new instruction or result. The control process also moves instruction and result bundles between stages and inserts results generated by the execute process into the result pipeline.

The writeback buffer in Figure 4 is used to insert new register values into the result pipeline. This buffer holds destination values generated by executing an operation in the stage. The control process moves values from the buffer to the result pipeline. There are two feasible ways to do this. In the first choice, the control process waits until the result register is empty before moving the value from the writeback buffer to the result pipeline register. This inserts a new value into the stream of result bundles flowing through the result pipeline. This approach is relatively simple; however, it increases pressure on result pipeline bandwidth by generating new result packets for every destination register, which in turn causes additional garner operations over the execution lifetime of a loop.

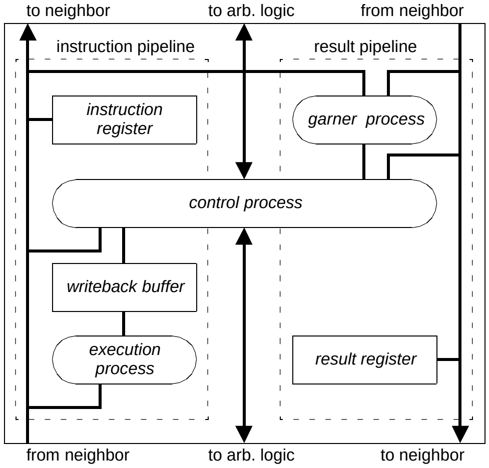

Figure 4: A diagram of a WCFP execution stage.

The second alternative moves the buffer's value immediately to the result register, regardless of whether the register is empty. When there is a bundle in the result register with enough space to hold the buffer's value, the value is *packed* into the existing bundle. Packing results limits the number of individual packets flowing through the result pipeline, which reduces the number of garner operations and improves result pipeline bandwidth utilization. We have found packing results improves performance by 1-11% on several benchmarks.

The writeback buffer has another advantage: it lets results flow through a stage during the execution of an operation by decoupling the execute process from other stage activity. Our experiments indicate that for good performance it is key to avoid delaying the delivery of results to their consumer operations. For this, results need to move through a stage as quickly as possible, for which a writeback buffer is perfectly suited. The inclusion of a buffer improves performance by up to 13%.

Launch and return stages are treated similarly to execution stages, except launch and return operations are done by the execute process, which communicates with a siding to initiate an operation or return a result.

2.3. Predicated Execution

The WCFP supports predicated execution to eliminate control flow in kernel loops. As demonstrated by other work [20], this can expose significant ILP by flattening the control flow graph into a sequence of operations that can be scheduled together.

The wide CFP handles predicated execution naturally by treating predicates as any other source operand and checking the value of the predicate before executing or launching an operation. If the predicate is false, the operation is not done.

Predicated execution in the WCFP requires the original value of the destination register be inserted into the result pipeline when a false predicate is encountered. This is done by including the destination register as a source operand to a predicated operation, and copying the original value of the destination register to the result pipeline on a false predicate. The original value is inserted because dependent operations may be waiting for the destination register, and if a value were not generated, those operations would deadlock.

3. Automatic Pipeline Design

Our work focuses on techniques for automatically generating wide counterflow pipelines customized to the resource requirements of an application's kernel loop. The 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. We characterize processor functionality by an user-supplied *synthesis 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. Thus, given what functional devices a WCFP contains, the topology space is all combinations of pipeline stages, excluding some combinations that do not make sense (e.g., placing a siding's return stage before its launch stage).

Figure 5 is a diagram of the customization process. The customization system accepts an application program (in C) with its kernel loop annotated as an input to the code improver *vpo* [2], which compiles the application and transforms the loop using classic optimizations such as strength reduction, induction variable elimination, global register allocation, loop invariant code motion, etc. Other optimizations are also currently done by hand prior to synthesis, including scalar replacement, path height reduction, and if-conversion.

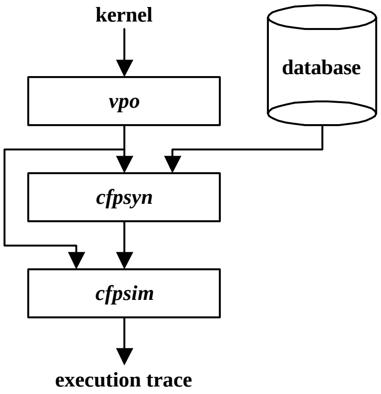

Figure 5: *vpo* optimizes a kernel loop that serves as a specification for a WCFP determined by *cfpsyn*. The WCFP is simulated and analyzed by *cfpsim*.

vpo passes the optimized kernel loop to the synthesis phase, *cfpsyn*, which selects and instantiates computational devices from the design database and derives the processor interconnection network. The synthesis step emits a description of the custom pipeline for the simulator, *cfpsim*, which collects performance statistics and an execution trace.

3.1. Pipeline Customization

The optimized instructions emitted by *vpo* and the syn-

```
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);
  sched.emit();
  WCFP.emit();
}
```

Figure 6: A software pipeline is formed that acts as a specification for a WCFP and instruction set.

thesis database are used to derive a custom WCFP. The synthesis process has four steps as shown in Figure 6. Each step is described in the following sections.

3.1.1 Software Pipelining

The first step of WCFP synthesis generates a software pipelined loop from the instruction sequence emitted by *vpo* using a variation of iterative modulo scheduling [22].

The resource constraints modeled by the software pipelining step are the operation slots in an instruction. The width of the instruction word (i.e., number of operation slots) is a user-supplied parameter to the synthesis system. The synthesis database determines scheduling latencies and what operations may appear together in an instruction word. The database specifies a single constraint: 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 custom WCFP.

3.1.2 Pipeline Extraction

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

```
generatePipeline(InstructionKernel kernel,
    SynthesisDatabase DB) {
  WCFP ← new WideCounterflowPipeline();
  forall word in kernel do {
    groups ← formOperationGroups(word, DB);
    generatePipelineStages(WCFP, groups);
  }
  return WCFP;
}
```

Figure 7: A WCFP is generated by grouping operations from an instruction into sidings and stages.

Figure 7 shows how a WCFP is generated from a software pipeline. The algorithm iterates over instructions in the pipeline kernel to group instruction operations. The operations are placed in 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 pipeline 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 (e.g., loads vs. multiplies).

After operations are grouped, the groups are used to derive pipeline stages and sidings that have the semantics required by each group. 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. Currently, an execution stage is created for every low latency operation group and we do not assign multiple operation groups to a single device. A functional siding and return and launch stages are created for a high latency operation group if one 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 can have several operations in flight at once (they are pipelined). When a siding is created, its launch and return stages are also inserted in the pipeline. The return stage is inserted so the number of stages between launch and return equals the siding's pipeline depth, which ensures the siding can be fully utilized.

3.1.3 Instruction Set Extraction

The synthesis system determines an instruction set for a WCFP, including instruction format design and opcode assignment. Instruction format design is currently done at a high-level and abstracts out many details about bit field widths (e.g., width of immediates and register specifiers), opcode encodings, and field arrangement. Instead, instruction set extraction assigns opcodes to each pipeline operation, identifies the status information kept for every operation, and canonicalizes the order of operations in an instruction in a bit-field independent way. It also creates an intermediate representation of a WCFP's instruction set that is used during code generation to build and emit WCFP instructions.

3.1.4 Code Generation

The final step of WCFP synthesis is code generation, which forms the complete instruction schedule, including loop preconditioning and control processor setup and tear-down code for the kernel processor.

The full software pipeline is built using the pipeline kernel. The first step of creating the full instruction sequence is *modulo variable expansion* (MVE), which eliminates dependence conflicts on registers whose lifetimes are greater than the software pipeline initiation interval (*II*). MVE works by unrolling the kernel enough times to remove register name conflicts by renaming. The unroll factor is computed using the maximum lifetime of all registers in the kernel. If the maximum lifetime is $f > II$, then the kernel is unrolled $F = \left\lceil \frac{f}{II} \right\rceil$ times. Static register renaming is used to eliminate resource conflicts for registers with *lifetime* > *II*. An alternative method eliminates these conflicts using rotating registers [24]. One advantage to rotating registers is only a small number of architecturally visible registers are needed; however, the scheme does require hardware and instruction set support. Because our synthesis system has control over register file geometry and instruction set architecture, static register renaming is used for MVE.

After unrolling the kernel, prologue and epilogue code is generated to begin and end the software pipeline. Although WCFP's support predicated execution, it is not used to eliminate prologue and epilogue code. Indeed, generating prologue and epilogue code and using static register renaming presents a limitation: the number of iterations of the pipeline kernel must be $(s - F) + F \times i$, where s the iteration span of the kernel and $i \geq 0$. The trip count of the original loop must be adjusted to fit this equation. The standard technique to do this is *loop preconditioning,* which executes the original loop a limited number of times until the iteration count matches the requirements of the software pipeline. Our system does loop preconditioning on the control processor and transfers control to the kernel processor at the end of the preconditioning loop. We are investigating adding support for rotating registers and branch-based predication as developed by Rau et al. [25] to eliminate the need for prologue and epilogue code and loop preconditioning.

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

4. Experimental Results

In this section we demonstrate that custom wide counterflow pipelines achieve performance that is equivalent to traditional VLIW architectures tailored to an application's resource requirements.

4.1. Methodology

The goal of our present work is to see how far WCFP's 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 WCFP's customized to the resource and data flow requirements of benchmark applications.

4.1.1 Evaluation

The performance statistics in this paper were collected using several common benchmarks. The benchmarks have integer versions of the 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 2-D discrete cosine transformation (*dct*) used in image compression and an implementation of the Floyd-Steinberg image dithering algorithm (*dither).* We also extracted the vector computation $a = b^k \bmod d$ from RSA encryption (*mexp*). The final benchmark is the kernel from the European GSM 6.10 standard for full rate speech decoding (*gsm*). The benchmarks were compiled using the code improver *vpo* [2] for the SPARC architecture.

4.1.2 Pipeline Simulation

We have built a behavioral microarchitecture simulator for asynchronous wide CFP's. The simulator is highly reconfigurable to permit microarchitecture experimentation, and it generates a detailed program execution trace that is post-processed by a separate analysis tool to collect performance statistics.

To model asynchronous pipelines our simulator varies computational latencies. Table 1 shows the latencies we use in our simulation models.

Operation	Latency
Stage copy	1 time unit
Garner, kill, update	3 time units
Return, launch	3 time units
Instruction operation	5 time units

Table 1: Computational latencies

The latencies in the table are expressed relative to how long it takes an instruction or result to move between adjacent pipeline stages. Using the base values from Table 1, we derive other pipeline latencies. For example, a simple operation such as addition takes 5 time units. High latency operations are scaled relative to low latency ones, so an operation such as multiplication—assuming it is four times slower than addition—takes 20 time units.

4.2. Wide Counterflow Pipelines

We have used the design methodology described in the previous section to generate custom asynchronous wide counterflow pipelines for several benchmark applications.

4.2.1 WCFP Performance

Figure 8 shows measurements of cycles per operation for custom wide counterflow pipelines for each benchmark from Section 4.1.1 using two design databases. The first database has functional sidings that issue a single operation at a time. This means that siding operations may not be scheduled in the same instruction as any other operation of a similar type. The second database uses sidings which can do two operations of a similar type at once.

The measurements in the figure were calculated using *effective cycles per operation* (ECPO). Because an asynchronous WCFP implementation is used, the execution latency must be normalized by an *effective clock cycle* (ECC) to derive CPO measurements. Given ECC, the formula for ECPO is:

ECPO = (latency/ECC)/instruction count.

For the graphs in Figure 8, ECC is 8. This normalization factor is used 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 clock cycle. We believe an ECC of 8 is conservative because simple CFP structures should lead to very fast effective clock cycle speeds, possibly faster than traditional architectures (despite potential penalties due to asynchronous implementation).

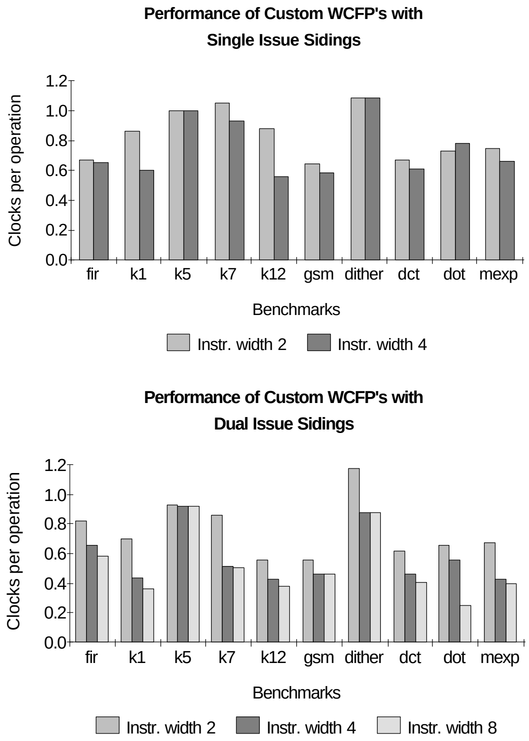

Figure 8: Clocks per operation for custom WCFP's with database with (a) single-issue sidings and (b) dual-issue sidings.

The top graph in the figure shows CPO measurements ranging from 0.56 to 1.09 for instruction widths of 2 and 4. Measurements are reported only for these instruction widths because at higher widths there is no change in performance. However, the single-issue pipelines do not take full advantage of available instruction parallelism in the benchmarks. This is attributable to the number of loads and multiplications in the benchmarks—these operations account for most of the available ILP. The single-issue database has functional devices that can do only a single memory operation or multiplication per instruction, which adversely affects the software pipeline initiation interval.

The dual-issue database has sidings that can do two memory operations or multiplications per instruction, which better matches the type of parallelism available in the benchmarks. The bottom graph in Figure 8 shows that CPO measurements for the dual-issue pipelines range from 0.25 to 1.17. The figure also demonstrates that for some benchmarks, CPO improves for an instruction width of 8 with dual-issue sidings over single-issue sidings. In this case, iterative-modulo scheduling is able to find a software pipeline with a smaller initiation interval, which led to better performance.

The measurements in Figure 8 indicate that custom wide CFP's are able to effectively exploit instruction-level parallelism in the benchmark loops.

4.2.2 Pipeline Refinement

Although the pipelines generated by our synthesis methodology effectively exploit instruction parallelism, the order of pipeline stages could be modified to use a loop's execution behavior to get better performance.

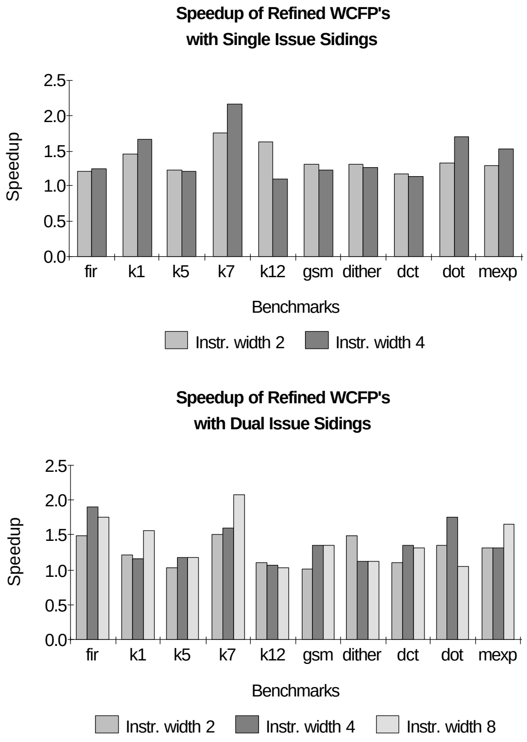

Figure 9: Performance improvement of WCFP's obtained by refining the placement of stages to match the dynamic flow of instructions and results.

As we have shown elsewhere [3], there are several factors that affect the performance of counterflow pipelines, such as the distance results flow between their

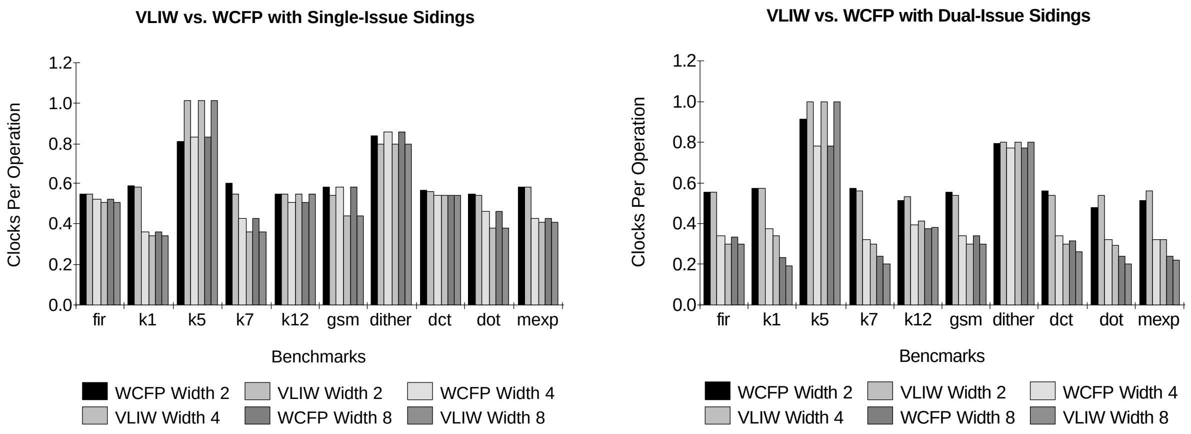

Figure 10: A comparison of clocks per operation for custom WCFP and VLIW organizations.

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 can be mitigated by arranging the order of pipeline stages to match the dynamic flow of results and instructions using a loop's execution trace.

Figure 9 shows the speedup of custom pipelines from Section 4.2.1 whose stage orders have been refined to match the execution behavior of each benchmark. The figure demonstrates that pipeline refinement is very important for good performance. Indeed, in all cases, performance was improved with speedup ranging from 1.03 to 2.16 and an average of 1.38.

A very simple methodology is used to refine a WCFP's stage order. The technique is based on identifying 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 works by selecting each pipeline stage in turn to be moved to a new location. After selecting a stage, the execution trace is examined to determine where the operation that executes in the selected stage becomes ready to execute. There are two possible locations: 1) the operation becomes ready in a stage *before* the selected stage, or 2) the operation becomes ready *at* the selected stage (this happens when an operation stalls in its execution stage while waiting for 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 very 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 selected 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. The refinement methodology 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 serves as a placeholder 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 simply 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 effectively traverses the design space of pipeline stage orders to find one that gives good performance. One limitation of this method is that it moves a single stage at a time and only keeps a move if performance improves. However, it may be advantageous to make a bad move, which may enable a very good move later in the process. We are currently investigating other methods of searching the design space, such as simulated annealing, and automatically including profile-feedback information to the synthesis system.

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

4.2.3 VLIW vs. WCFP Comparison

Figure 10 shows a comparison of traditional synchronous VLIW architectures versus custom wide counterflow pipelines. The data in the figure was collected using the same software pipeline for the traditional VLIW architecture and the WCFP, ensuring a fair comparison between the two organizations using the same instruction schedule. Furthermore, this ensures that the VLIW has the same resource capabilities as the WCFP.

The figure shows a comparison of clocks per operation for both VLIW and WCFP processors with varying instruction and functional siding widths. As the figure indicates, WCFP's achieve CPO measurements that are competitive with traditional VLIW's architectures with similar resource capabilities. The figure shows that 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 custom pipeline 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.

4.3. WCFP Enhancements

Although performance is good for most benchmarks in Figure 10, the more aggressive designs (instruction widths of 4 and 8 for dual-issue sidings) have a greater relative difference in performance between the WCFP and VLIW architectures than for less aggressive designs. This appears to be influenced by partitioning operations into groups. For some benchmarks, like *dct* and *dot*, large operation groups can adversely affect performance because all operations in a group must acquire their source operands prior to any operation executing. This can significantly delay the execution of some operations that become ready very early. For *dct* and *dot*, performance is 18% worse than an equivalent VLIW architecture with an instruction width of 8.

If the execution of operations in a group is decoupled, then those operations which garner their source operands early could execute early and generate their destination value as soon as possible and potentially avoid unnecessary pipeline stalls. For example, consider an operation along a loop-carried dependence path that is grouped with other operations near the bottom of the dependence graph. If the operation is executed late in the pipeline, then it will likely delay the execution of the succeeding loop iteration. However, if the operation is decoupled from the other operations in the group, it could execute as soon as possible.

We are presently studying ways to decouple the execution of instruction operations. One possible solution lets operations execute as soon as they are ready regardless of whether other operations in the same group have acquired their source operands. In this scheme, the operations move together through the instruction pipeline in a single instruction. This may lead to an inefficient design because all operations in a group must wait while any one operation executes. A more aggressive pipeline design fully decouples instruction operations, including the movement of operations in the pipeline. In this scheme, operations in an instruction are issued together to separate pipelines for each instruction slot. This lets operations move independently of their issuing group and execution stage positions can be set to the place most appropriate for each individual operation. Because this scheme allows operations to move out of sequential order, dynamic register renaming or dependence checking is required to ensure correct execution.

We have modified the *dct* and *dot* benchmarks to do a limited form of decoupled execution and have found this improves performance greatly, making the *gsm* and *dot* pipelines competitive with equivalent VLIW organizations. Based on our preliminary work, we believe fully decoupled execution will significantly improve performance of application-specific WCFP's.

4.4. Related Work

In the last ten years, asynchronous microprocessors have gained much attention because of their promise for design ease, high performance, and low cost. There have been several asynchronous microprocessor proposals, including a design from the California Institute of Technology [30], a decoupled access-execute microarchitecture from the University of Utah [27], and a low-power implementation of the ARM architecture from Manchester University [9, 10].

Although the counterflow pipeline was proposed as an asynchronous organization for general-purpose microprocessors [29], there has also been a proposal for synchronous version [21]. However, this work adds significant hardware structures to the original design to get good performance on a wide variety of applications. In our work, we customize counterflow pipelines to a single application to get high performance without introducing new microarchitecture enhancements.

There has also been much interest in automated design of application-specific integrated processors (ASIPs) because of the increasing importance of high-performance and quick turn-around in the embedded systems market. ASIP techniques typically address two broad problems: instruction set and microarchitecture synthesis. Instruction set synthesis attempts to discover micro-operations in a program (or set of programs) that can be combined to create instructions [14, 15]. The synthesized instruction set is optimized to meet design goals such as minimum program size and execution latency. Microarchitecture synthesis derives a microprocessor implementation from an application (or set of applications). Many microarchitecture synthesis systems use a co-processor strategy to synthesize custom logic for a portion of an application and to integrate the custom hardware with an embedded processor core [7, 13, 26]. Another microarchitecture synthesis approach tailors a single processor to the resource requirements of the target application [5, 8]. Although instruction set and microarchitecture synthesis can be treated independently, many co-design systems attempt to unify them into a single framework [11].

Our current research focus is microarchitecture synthesis. We customize a counterflow pipeline microarchitecture to an application using RISC operations and information about the data flow of the target application. Our synthesis technique has the advantage that the design space is well defined, making it easier to derive custom pipeline configurations that meet design goals.

5. Summary

This paper demonstrates that custom wide-issue counterflow pipelines are well-suited for automatic design of application-specific microprocessors. In the paper, we extend the original counterflow pipeline processor to a wide instruction organization that is easily customized to the requirements of an application's kernel loop, and we present a simple and effective technique for customizing wide CFP's to an application. We show that custom asynchronous WCFP's achieve performance that is competitive with traditional VLIW organizations at a potentially low design cost. Finally, this paper suggests several enhancements to the WCFP that may further improve performance of kernel loops.

References

[1] Allan V. H., Jones R. B, Lee R. M., et al., "Software pipelining", *ACM Computing Surveys*, pp. 367–432, Vol. 27, No. 3, September 1995.

[2] Benitez M. E. and Davidson, J. W., "A portable global optimizer and linker", *SIGPLAN Notices 1988 Symp. on Programming Language Design and Implementation*, pp. 329–338, Atlanta, Georgia, June 1988.

[3] Childers B. R. and Davidson J. W., "A design environment for counterflow pipeline synthesis", *Workshop on Languages, Compilers, and Tools for Embedded Systems* (*LCTES '98*), held in conjunction with *PLDI '98*, Montreal, Canada, June 19–20, 1998.

[4] Childers B. R. and Davidson J. W., "Architectural considerations for application-specific counterflow pipelines", to appear, *20th Anniversary Conf. on Advanced Research in VLSI (ARVLSI'99)*, Atlanta, Georgia, March 1999.

[5] Corporaal, H. and Hoogerbrugge J., "Cosynthesis with the MOVE framework", *Symp. on Modelling, Analysis, and Simulation, CESA '96*, pp. 184–189, Lille, France, July 1996.

[6] Davis A. and Nowick S. M., "An introduction to asynchronous circuit design", Technical Report UUCS-97-013, University of Utah, Sept. 1997.

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

[8] Fisher J. A., Faraboschi P., and Desoli G., "Custom-fit processors: Letting applications define architectures", Technical Report HPL–96–144, Hewlett-Packard Laboratories, Palo Alto, California, 1996.

[9] Furber S. B., Day P., Garside J. D., Paver N. C., and Woods J. V., ''AMULET1: A micropipelined ARM", *Proceedings of the IEEE Computer Conference*, March 1994.

[10] Furber S. B., Garside J. D., Temple S. et al., "AMULET2e: an asynchronous embedded controller", *Int'l Conf. on Adv. Research in Asynchronous Circuits and Systems (Async97)*, pp. 290–299, Eindhoven, The Netherlands, April 1997.

[11] 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.

[12] Hauck S., ''Asynchronous design methodologies: An Overview", *Proc. of the IEEE*, Vol. 83, No. 1, January 1995, pp. 69–93.

[13] Hauser J. R., and Wawrzynek J., "Garp: A MIPS processor with a reconfigurable coprocessor", *IEEE 5th Annual Symp. on Field-Programmable Custom Computing Machines*, pp. 12–21, Napa Valley, California, April 16-18, 1997.

[14] Holmer B., *Automatic Design of Instruction Sets*, Ph.D. thesis, University of California, Berkeley, 1993.

[15] 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.

[16] Hunt D., “Advanced performance features of the 64-bit PA-8000”, *COMPCON ‘95 Digest of Papers*, pp. 123–128, March 1995.

[17] Jank K. J., Lu S-L, Miller M. F., “Advances of the counterflow pipeline microarchitecture”, *Proc. 3rd Int’l Symp. on High-Performance Computer Architecture*, pp. 230–236.

[18] Lee C., Potkonjak M., Magione-Smith W. H., “MediaBench: A tool for evaluating and synthesizing multimedia and communications systems”, *Proc. of the 30th Int’l Symp. on Microarchitecture (MICRO-30)*, pp. 330–335, Research Triangle Park, North Caronlina, Dec. 1997.

[19] 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.

[20] Mahlke S. A., Hank R. E., McCormick J. E., et al., “A comparison of full and partial predicated execution support for ILP processors”, *Proc. 22nd Int’l. Symp. on Computer Architecture*, pp. 138–149, Santa Margherita Ligure, Italy, June 1995.

[21] Miller M. F., Janik K. J., and Lu S-L, “Non-stalling counterflow architecture”, *Proc. 4th Int’l Symp. on High-Performance Computer Architecture*, pp. 334–41, Las Vegas, Nevada, Feb. 1-4, 1998.

[22] Rau B. R., “Iterative modulo scheduling: An algorithm for software pipelined loops”, *Proc. of 27th Annual Int’l Symp. on Microarchitecture*, pp. 63–74, Dec.1994, San Jose, California.

[23] Rau B. R. and Fisher J. A., “Instruction-level parallel processing: History, overview, and perspective”, *J. of Supercomputing*, Vol 7, pp. 9–50, May 1993.

[24] Rau B. R., Lee M., Tirumalai P., et al., “Register allocation for modulo scheduled loops”, *SIGPLAN Notices 1992 Symp. on Programming Language Design and Implementation*, pp. 283–299, June 1992, San Francisco, California.

[25] Rau B. R., Schlansker M. S., and Tirumalai P. P., “Code generation schema for modulo scheduled loops”, *Proc. of 25th Annual Int’l Symp. on Microarchitecture*, pp. 158–169, Dec.1992.

[26] Razdan R. and Smith M. D., “A high-performance microarchitecture with hardware-programmable functional units”, *Proc. of 27th Annual Int’l Symp. on Microarchitecture*, pp. 172–180, Dec.1994, San Jose, California.

[27] Richardson W. and Brunvand E., "Fred: A Decoupled Self-Timed Computer," *Int’l Conf. on Adv. Research in Asynchronous Circuits and Systems (Async96)*, pp. 60–68, Aizu, Japan, March 1996.

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

[29] 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.

[30] Tierno J., Martin A. J., Borkovic D., and Lee T. K., “A 100-MIPS GaAs asynchronous microprocessor”, *IEEE Design and Test of Computers*, Vol 11., No 2., pp. 43–49, 1994.