An Infrastructure for Designing Custom Embedded Counterflow Pipelines

Bruce R. Childers, Jack W. Davidson
Department of Computer Science
University of Virginia
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 an embedded system. We have developed a new microarchitecture for automatically constructing ASIPs. This new architecture, called a wide counterflow pipeline (WCFP), is based on the counterflow pipeline (CFP). Our ASIP synthesis technique uses software pipelining and design-space exploration to generate a custom WCFP and instruction set for an embedded application. In this paper, we first present a brief sketch of WCFPs and our design strategy. Second, we describe a software infrastructure for prototyping WCFPs to evaluate design trade-offs. Finally, based on preliminary experiments using several kernel loops, we show that WCFPs achieve speedups of 1.8–6.6 over a general-purpose CFP.

1. Introduction

Application-specific instruction set processor (ASIP) 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. We are researching techniques for automatically designing such processors based on the counterflow pipeline (CFP) architecture. The CFP has several characteristics that make it attractive for synthesis of custom embedded processors: it has a simple and regular structure, local communication, a high degree of modularity, and inherent handling of complex structures such as register renaming and speculative execution.

Because cost and time-to-market constraints are very important to embedded systems [17], an ASIP architecture should permit automatic design, including high-level architectural design. As we have shown previously, the CFP is a good candidate for this type of fast, aggressive synthesis because of its superior composability and simplicity [4]. This composability 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 also makes it easy to incorporate functional devices for the specific needs of an application and to explore design trade-offs such as the number and type of functional units.

We are investigating microarchitecture features for custom CFPs, including a proposal for a wide instruction architecture that is appropriate for the design of instruction-level parallel (ILP) embedded processors. This new architecture, which we call a wide counterflow pipeline (WCFP), is an extension to the original CFP that supports automatic design of pipelines that issue an arbitrary number of operations per instruction [6]. The WCFP retains all the advantages of the original CFP: it is a very composable structure with local communication.

To support quick-turn around and automatic design of custom WCFPs, our research uses an application expressed algorithmically in a high-level language as a specification for a WCFP. This type of aggressive design requires a prototyping environment that is highly reconfigurable and extensible to permit fast exploration of microarchitecture alternatives. The prototyping environment should:

- Let a designer describe the design space of an application-specific processor;
- Have a flexible intermediate representation for constructing ASIPs automatically;
- Be able to quickly evaluate cost-performance trade-offs.

This paper describes our prototyping infrastructure, which supports the requirements above using a database to describe computational devices and resource constraints; a graph-based intermediate representation (IR) of custom pipelines; and a fast and reconfigurable simulator and performance analysis tool.

2. Wide Counterflow Pipelines

The WCFP is a VLIW microarchitecture that issues several operations per instruction to exploit ILP in kernel loops. The WCFP has two pipelines flowing in opposite

directions as shown in Figure 1. 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.

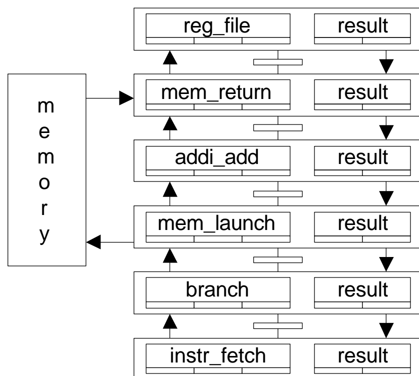


Figure 1: Example WCFP.

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 1 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 register names.

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. 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 that can do 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 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 can be tailored to the dynamic data flow behavior of the application to significantly improve performance. The WCFP supports predicated execution to help eliminate control flow in loops.

As Figure 1 shows, WCFPs 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, local device communication may lead to very fast implementations, especially as global wire delays dominate critical path latencies.

2.1. 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 way 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 WCFP. For JPEG a small speedup of the kernel loop of 6 or 7 achieves most of the overall speedup.

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 interconnection networks.

Our target system architecture has two processors: a traditional processor for executing control code and a WCFP for executing the computation portions of an appli-

cation. 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, many embedded applications have execution profiles similar to JPEG. We profiled several applications from the MediaBench benchmark suite (a collection of popular multimedia applications) 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.

2.2. Custom Pipelines

Our work focuses on techniques for automatically generating WCFPs customized to the resource requirements of an application's kernel loop [3,5]. The customization process operates at the architectural-level on pre-designed devices such as pipeline stages and functional sidings.

The design space of WCFPs 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 for each database entry. WCFP topology is determined by the order of pipeline stages, which fully specifies the interconnection of functional devices in a WCFP.

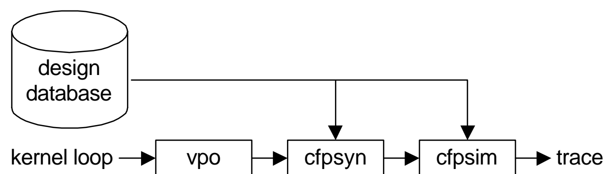


Figure 2: Workflow of WCFP synthesis.

Figure 2 is a diagram of the design system. The system accepts an application with its kernel loop annotated as an input to the code optimizer *vpo* [1], which compiles the application and transforms the loop using optimizations such as strength reduction, induction variable elimination, global register allocation, loop code motion, global common subexpression elimination, etc.

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

```
1   genCounterflowPipeline(SynthesisDB DB, Loop L) {
2   // step 1: form the software pipeline kernel
3   PipelineKernel kernel ← moduloSchedule(L);
4   // step 2: generate WCFP
5   WCFP P ←genWCFP(kernel, DB);
6   // step 3: generate instruction set architecture
7   InstructionSet ISA ← generateInstrSet(P);
8   // step 4: scheduled & emit WCFP
9   InstrSched sched ← softwarePipeline(kernel, ISA);
10  emit(sched, P); }
```

Figure 3: WCFP synthesis process.

The synthesis phase, *cfpsyn*, uses software pipelining and design space exploration to derive a WCFP. Software pipelining is used because the type of applications our work targets greatly benefit from aggressive ILP optimizations. Design exploration lets the synthesis system empirically refine and pick a pipeline that is tailored to a particular application's execution behavior. To generate a custom WCFP, the optimized instructions emitted by *vpo* and the design database are inputs to the synthesis system, which does four steps as shown in Figure 3.

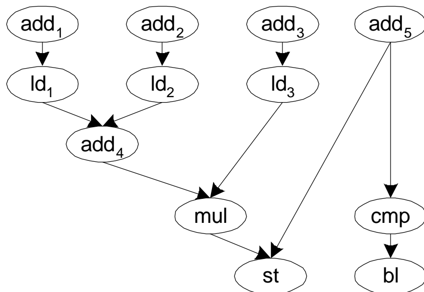
Step 1 of synthesis constructs a software pipelined loop using iterative modulo scheduling [15] for the instruction sequence emitted by *vpo*. The design database gives resource and latency constraints used during the formation of the software pipeline loop. The software pipeline kernel specifies the operations and functional elements to include in a WCFP. Step 2 does *pipeline extraction*, which instantiates pipeline stages and sidings for kernel instructions from the design database. Individual pipeline stages in the database may be combined into composite stages that execute wide instructions. After pipeline extraction, step 3 creates an instruction set architecture (ISA) from the software pipeline kernel and the WCFP. Once the custom pipeline and ISA are determined, step 4 generates the full instruction schedule for running the software pipelined loop on the WCFP.

A final process, called pipeline refinement, is performed after WCFP synthesis. Pipeline refinement iteratively adjusts the order of pipeline stages to match a loop's execution behavior. Stage order is refined using a simple heuristic that identifies locations in the pipeline with heavy resource contention and re-arranges stages to reduce this contention.

Figure 4 shows an example of generating a custom WCFP. An example loop is shown in (a) with its dependency graph in (b). To keep the example simple, the dependency graph is not optimized: several of the addi-

```
for (i=0; i < 100; i++)
  a[i] = (b[i] + c[i]) * d[i];
```

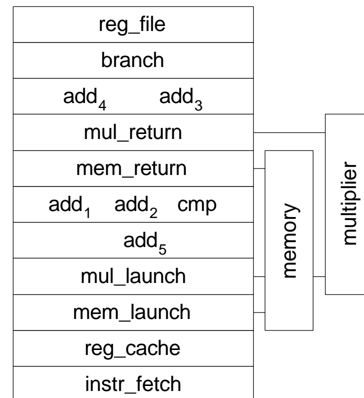
(a) Example code sequence



(b) Dependency graph

No.	Instruction			
1	ld_1	ld_2	mul	add_5
2	ld_3	add_1	add_2	cmp
3	add_4	add_3	st	
4	bl			

(c) Software pipeline kernel



(d) Custom pipeline

Figure 4: An example of pipeline synthesis.

tions can be removed using induction variable elimination. Also, the graph does not show loop-carried dependence edges. Using the dependency graph, a software pipeline kernel is formed in (c). In this example, the instruction width is constrained to four operations per instruction.

The software pipeline has an initiation interval of 3, which is optimal because the loop is resource constrained—it takes at least three instructions to issue the 12 operations in the dependence graph. Currently, the synthesis system places the branch at the end of the software pipelined loop in its own instruction rather than attempting to schedule it elsewhere in the loop.

From the software pipeline kernel, a WCFP is formed (d) using an iterative process that considers each instruction in the pipeline kernel to determine processor functionality. In this example, starting with the first instruction, 11 pipeline stages and two functional units are generated. An addition stage is generated for the add operation in the instruction and a memory and multiply functional units are generated for the load and multiply operations. In our current implementation, operations that execute in functional units can not be combined with any other operation in the same pipeline stage. For instruction 1, this means separate launch and return stages are generated for the load and multiply operations. As a result, four stages are inserted into the pipeline for the first instruction: `mem_launch`, `mem_return`, `mul_launch`, and `mul_return`. Instruction 2 of the pipeline kernel generates one stage: `add_add_cmp`. This composite stage executes two addition and one comparison operation. The load operation in the second instruction does not require generation of any new functional devices since a memory unit was generated by the previous instruction. The third instruction is similar to the second one, and a stage that executes two addition operations is generated. The store in the third instruction does not generate an additional pipeline stage because `mem_launch` can initiate both store and load operations.

3. Design Infrastructure

The key to automating WCFP synthesis is having a flexible and extensible prototyping environment that can be automatically changed and configured. This flexibility is especially important since the design methodology adjusts a WCFP organization, which requires a method for representing WCFP designs and automatically reconfiguring a simulator. We have built a prototype system that accepts an application as a C program with its kernel loop annotated and generates a custom computing system for the application, which includes a WCFP, an instruction schedule for the WCFP, and a control program. To support automatic design of WCFPs, our system includes a database to describe computational devices and resource constraints; a graph-based intermediate representation of custom pipelines; and a fast and reconfigurable simulator.

3.1. Design Database

The design infrastructure uses a database to specify design elements and resource constraints. The database is constructed once by a designer—it can be re-used for different applications that have similar requirements. The database should be relatively small: our current database has 34 entries and 261 lines of text.

Pipeline extraction uses the database to select devices for a WCFP. The database lists characteristics and semantics of pipeline stages and functional sidings. Device characteristics are attributes such as latency, opcode repertoire, cost, and semantics. For example, Figure 5 gives latency and cost attributes for an addition stage on line 4.

Database entries have an attribute for device type. Pipeline stages have six types: instruction fetch (`instr_fetch`), register cache (`reg_cache`), execution (`execute`), launch (`launch`), return (`return`), and register file (`reg_file`). An instruction fetch stage fetches instructions and inserts them into the pipeline. This stage

is also responsible for branch prediction. A register cache is a stage that caches result values that have exited the result pipeline so that instructions just entering the instruction pipeline can quickly acquire their source operands from the register cache. An execution stage does some operation(s) in the main pipeline; execution stages can be combined during synthesis to form composite stages that execute multiple operations. Launch and return stages initiate and complete pipeline siding operations. They are automatically derived during synthesis depending on what functional units are included in a design. Finally, a register file stage is a buffer that records destination register values as they exit the pipeline.

```
1    database “Example” {
2    // an addition stage
3    stage “addition” {
4       type=execute; latency=5; cost=0.5;
5       opcodes={ADD,ADDI};
6       semantics={r[$]=r[$]+r[$];
7          r[$]=r[$]+imm16; };
8       ports={i_in:in ,i_out:out, r_in:in ,r_out:out }; }
9    // a memory siding
10   siding “memory” {
11      type=rlu; latency=15; depth=3; width=2;
12      opcodes={LD.W,ST.W};
13      launch_opcodes={LD.W,ST.W};
14      return_opcodes={LD.W};
15      semantics={ ld.w: r[$]=M[r[$]+r[$]];
16         st.w: M[r[$]+r[$]]=r[$]; };
17      disallow={<st.w,st.w>};
18      ports={l_in:in,r_out:out}; }
```

Figure 5: Example WCFP database.

There are two functional siding types: launch-unit (`lu`) and return-launch-unit (`rlu`). A launch unit is a siding that only initiates an operation; e.g., a store unit. A return-launch unit initiates an operation and returns a value to the main pipeline following the initiation of an operation; e.g., a load unit, multiplier, etc.

A database entry also has an attribute for device opcode repertoire, which maps instruction dependency graph nodes to functional devices. Pipeline extraction uses the opcodes to determine which devices to include for a given dependence graph node. Database entries also have an attribute for listing a device's ports, which describe how stages and sidings are interconnected. For example, pipeline stages have four ports to connect to adjacent stages: instruction input (`i_in`), instruction output (`i_out`), result input (`r_in`), and result output (`r_out`).

Database entries also have an attribute for device semantics, which are used to automatically configure WCFP simulations to support different ISAs. Device semantics specify the effect of opcodes in a device's repertoire. The semantics are interpreted by the WCFP simulator to do what action is required by a pipeline device. Register transfer lists (RTLs) are used to describe semantics because RTL is a small language that concisely captures instruction effects [9] and is easily interpreted.

Figure 5 shows an example database that has an addition stage and a memory siding. The addition stage does two operations according to its semantics: *register–register* addition and *register–immediate* addition. The RTL on lines 6–7 indicates that two registers may be added and that a register and a 16-bit immediate may be added.

The memory unit in Figure 5 on lines 9–18 supports loading and storing 32-bit words. This device has several attributes not present in the addition stage, including *depth* and *width* attributes. *Depth* indicates the length of a siding's pipeline, and *width* indicates how many operations the siding can issue simultaneously. In this case, the memory siding has three stages and can issue two memory accesses per instruction.

The memory siding in Figure 5 is a return-launch unit. However, because the siding handles stores, it does not always return a value. This is indicated by the *launch_opcodes* and *return_opcodes* attributes. During synthesis, these attributes are inspected to determine the operation repertoire of launch and return stages. For this example, when *cfpsyn* generates a memory siding, it instantiates a memory launch stage that initiates read and write accesses, and a return stage that extracts values read from memory.

3.2. Pipeline Intermediate Representation

Because the synthesis system generates WCFPs on-the-fly, it needs a way to represent a design's architecture that allows for describing incomplete designs as synthesis progresses. To solve this problem, the design system uses a graph-based intermediate representation (IR) that makes it easy to add, delete, and modify functionality of a custom pipeline. The IR is called a *pipeline intermediate representation* (PIR). The requirements of the PIR include:

1) *Flexibility*: The PIR should be usable for *both* synthesis and simulation.
2) *Simplicity*: The PIR should be easy to manipulate and modify during architectural synthesis.
3) *Abstraction*: The PIR should represent devices at the architectural level without low-level detail that does not affect synthesis decisions.

These requirements led to a representation that describes devices and their attributes; pipeline structure; device semantics; and instruction operations. The PIR is a structural representation that has device entities, port connections, and design literals. An entity is an instantiated pipeline stage or siding, and a port is a communication channel between two devices. A literal is a value supplied to a device attribute; e.g., latency, memory size, etc.

The PIR is a directed graph as shown in Figure 6. Nodes are entities with attributes, and edges are port con-

nections. The example graph shows six entities, including five pipeline stages and one functional siding. In the example, the pipeline stages are connected through the `i_out`, `i_in`, `r_out`, and `r_in` ports, which specify the instruction and result pipelines.

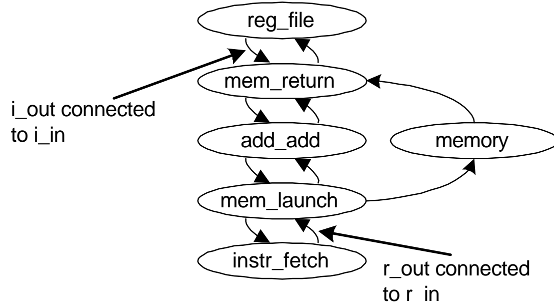


Figure 6: Pipeline intermediate representation.

A *pipeline description* is a textual form of the PIR, which can be stored in a file for use by other tools (or edited for experimentation). The WCFP simulator, in particular, is configured using pipeline descriptions. To specify a WCFP, the PIR has three parts: global literals, stage instantiation, and siding instantiation. Global literals specify memory size, register file geometry, result pipeline width, and pipeline latencies. Figure 7 illustrates a WCFP with a 64K memory, a register file with 32 general-purpose registers, and a double wide result pipeline. This pipeline moves instructions and results between stages in one time unit, and it garners results in three time units.

Stages are instantiated using *description blocks* that connect ports and assign literal values to attributes. Stages are listed in the order they should be connected in a pipeline, starting with the first stage. A description block has a type to specify the simulation model to instantiate for a stage. Figure 7 lists an addition stage on lines 14–17. The `addi_add` stage is an execution stage that does two additions in five simulation time units, as shown by the RTL:

r[$4]=r[$0]+imm16[$1], r[$5]=r[$2]+r[$3];

These semantics were generated by combining the RTLs of the operations that compose the `addi_add` stage. The stage adds the first and second source operands of an `ADDI_ADD` operation, writing the sum to the fifth register operand. Similarly, the third and fourth operands are added with their sum written to the sixth operand.

An instantiated stage's semantics describe instruction format by listing operand type and position in a bit-field independent way. For example, the `addi_add` stage's RTL specifies that the `ADDI_ADD` operation has one immediate and five general-purpose register operands (three sources and two destinations). For this operation, the source operands appear at positions 0, 1, 2, and 3, and the destination operands appear at positions 4 and 5. A slightly more complicated RTL is shown for the `BRANCH` operation on line 10. In this case, the RTL specifies a control transfer that compares a register and immediate operand according to a condition given in the instruction. If the condition is true, control is transferred to the PC-relative location using the 23–bit immediate operand.

```
1   pipeline “Example” {
2     memory { size=64K; }
3     registers { r:<size=32,width=32> };
4     resultwidth=2; garnerlatency=3;
5     instrcopylatency=1; resultcopylatency=1;
6     stage “instr_fetch” { type=fetch; latency=3; }
7     stage “branch” { type=branch; latency=3;
8       opcode={BRANCH};
9       mispredictlatency=5; policy=2bit; size=2048;
10      semantics={<PC=(r[$1] cond[$0]
11        imm16[$2])?imm23[$3],PC>}; }
12    stage “mem_launch” { type=launch;
13      opcode={LD,ST}; latency=3; }
14    stage “addi_add” { type=execute;
15      opcode={ADDI_ADD}; latency=5;
16      semantics={<r[$4]=r[$0]+imm16[$1],
17        r[$5]=r[$2]+r[$3]>}; }
18    stage “mem_return” { type=return;
19      opcode={LD}; latency=3; }
20    stage “reg_file” { type=regfile; }
21    siding “memory” { type=rlu; opcode={LD,ST};
22    latency=15; depth=3;
23    semantics={r[$2]=M[r[$0]+r[$1]];
24      M[r[$1]]+r[$2]]=r[$0]; } }
25    l_in=mem_laumch.l_out; r_out=mem_return.r_in; }}
```

Figure 7: An example pipeline description.

The final section of a WCFP description lists functional sidings. Lines 18–25 of Figure 7 show a memory siding that has latency of 15 simulation time units and a pipeline depth of 3 stages. The latency and depth attributes describe a resource vector for modeling a siding's pipeline. The memory siding lists port connections that determine where the siding is connected to the main pipeline. In this case, the memory siding is connected to the `mem_launch` and `mem_return` stages.

WCFP pipeline descriptions are small structural specifications. For several benchmarks applications (the benchmarks are discussed in Section 4), the descriptions range in size from 108 to 153 lines of text, with an average of 128 lines. Because pipeline descriptions are small and simple, they are easily manipulated by pipeline synthesis to construct a WCFP. Their simple nature also makes it easy to automatically configure the WCFP simulator to evaluate designs.

3.3. Simulation

Simulation is an important step in the development of a new computer architecture because it lets engineers validate ideas and techniques prior to building actual hardware. For embedded system design, it also helps designers

make trade-offs between hardware and software functionality. We have built a simulator for WCFPs that has several features for exploring microarchitecture design alternatives: rapid and easy reconfiguration using pipeline descriptions; extension by writing new object-oriented simulation models; flexible instruction format and opcode assignment; an easily modified instruction operation repertoire; and parameterized architectural features, such as register file and memory geometry.

Device Modeling. Parameterized simulators such as SimpleScalar [2] and Shade [7] use design arguments (e.g., cache size, branch prediction strategy, etc.) to configure a simulation. Most parameterized simulators do not permit changing the underlying microarchitecture without modifying simulation models. Changing the functionality of such a simulator can be a source of considerable development time and expensive that adversely affects time to market. We address this issue by using a simulator that can be re-configured quickly via PIR descriptions that permit configuring a WCFP's structure and ISA features.

The WCFP simulator separates instruction semantics and architectural state from pipeline behavior and timing using a structural simulator and an execution engine. The structural simulator models the movement of instructions and results in a WCFP. This includes modeling pipeline rules, latency, resource usage and conflicts, etc. The execution engine interprets instructions to maintain architecturally visible state. The structural simulator invokes the execution engine, which interprets device semantics (RTLs) to simulate instruction effects.

The separation of a simulation into a microarchitecture simulator and execution engine makes the simulator highly reconfigurable and extensible to support a variety of WCFP organizations. For example, to add a new functional device, only the microarchitecture simulator needs to be modified. Likewise, to change a WCFP's operation repertoire, only the execution engine needs to be changed. Modifications can be made in each simulation component without affecting the other component. Because stage semantics are specified by a pipeline description, simulation models do not need to be written for every WCFP. Separating semantics from structural models also has the advantage that the execution engine can be used independently to verify functional correctness without the expense of timing-accurate simulation.

Configurable simulation is key to WCFP synthesis because a WCFP can be automatically simulated to collect performance statistics and to evaluate design trade-offs. The flexibility of the simulator eliminates the need to write simulation models for every custom WCFP. This lead to a very maintainable and stable system because the likelihood of introducing errors by writing new simulation models is eliminated. Furthermore, the simulator needs to be verified only once during initial development. The absence of these steps—writing and verifying models—makes automatic design more feasible.

Execution-Driven Simulation. We use execution-driven simulation to model WCFPs because the underlying instruction set and microarchitecture are modified. The alternative technique is trace-driven simulation which uses a program execution trace to drive a simulation. However, reliance on a previously collected trace implies that the instruction set (and microarchitecture) can not be changed from the trace's instruction set. This makes it inappropriate for WCFP design because WCFP microarchitectures and instruction sets differ per application. Execution-driven simulation also makes the WCFP simulator extensible: it can easily model different WCFP structures, devices, and instruction sets.

Timing Model. The WCFP simulator needs to be fast because it is used to iteratively adjust a pipeline design; our prototype synthesis system (implemented in Java) evaluates up to 137 designs per hour on average (including synthesis and simulation) using a Sun UltraSPARC II 360 MHz workstation. To accomplish this, the simulator uses a *fixed-increment timing* model. Every simulation device receives a "tick" per simulation cycle to indicate advancement of time. This differs from event-driven simulation where devices schedule events in an queue (a "timing wheel") that is advanced to the next cycle only when there are no more events for the current cycle. The simulator models behavior at the device level, which lends itself well to fixed-increment timing.

Performance Analysis. The execution trace generated by the WCFP simulation environment is post-processed to gather statistics about instruction and result flow, pipeline utilization, and throughput. These statistics are used to evaluate a custom WCFP and to make changes in a WCFP organization to improve performance. The performance analysis tool generates a visual representation of a simulation run, including the movement of instructions and results, resource conflicts, and summary statistics. The visual representation is emitted as an HTML marked-up version of the execution trace that is used for archiving a simulation run and can be published on the Internet for future analysis and study. Publishing design assumptions and simulation runs ensures that WCFP experiments can be reproduced by other researchers.

4. Experimental Results

We have used our design infrastructure to generate WCFPs customized to the resource and data flow requirements of several benchmark applications. The benchmarks include integer versions of the Livermore loops *k1*, *k5*, 7

k7 and *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 speech decoding (*gsm*).

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, so an operation such as multiplication—assuming it is four times slower than addition—takes 20 units.

4.1. Design Space

The design infrastructure presented in Section 3 collects performance statistics and an execution trace for each design considered by pipeline refinement. The refinement process iteratively adjusts a pipeline to find one that is well matched to the execution behavior of an application. In this section, we focus on pipeline refinement since it greatly affected the design of our prototyping infrastructure. Indeed, the performance of the pipeline refinement system accounts for the majority of execution time required to synthesize a WCFP.

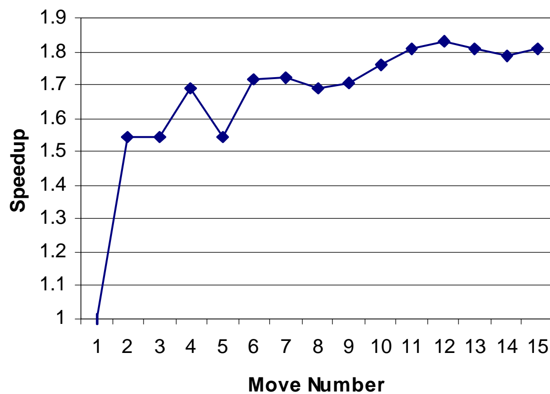


Figure 8: Pipeline refinement for *fir*

To give an indication of how pipeline refinement works, Figure 8 shows the effect of making pipeline adjustments (i.e., moving a stage to a different position in the pipeline) on performance of *fir*. The figure plots speedup of the pipelines after several moves made by pipeline refinement. The figure shows the refinement process for pipelines that have an issue width of 4 operations per instruction (i.e., the instruction pipeline width is 4 for these two benchmarks), and the speedup is relative to the initial starting point determined by the software pipelining and pipeline extraction steps. In the figure, the moves along the *x*-axis represent changing the position of a single pipeline stage. Each move corresponds to a simulation run by the design infrastructure.

The figure shows a common pattern exhibited by pipeline refinement: refinement initially finds an operation that is causing many pipeline stalls, and moves that stage to a position where there are fewer stalls. This initial move improves performance dramatically, followed by several additional moves that improve performance gradually. Whenever refinement makes a move that degrades performance, it discards the move and reverts to the previous pipeline and tries a new move. There is another refinement pattern that is common: performance is gradually improved over a number of pipeline moves. This pattern is typical of benchmarks that do not have a significant overall speedup resulting from refinement.

Table 1 shows that the number of moves made by pipeline refinement is usually small. The table also lists the optimal number (*Opt.*) of moves required to go from the initial design determined by pipeline extraction to the final one determined by pipeline refinement. The optimal number of moves is given as the minimal number of stage insertions (i.e., moving a stage from one pipeline position to another) required to transform one design into another.

Program	Moves	Opt.	Program	Moves	Opt.
fir	15	7	*gsm*	24	6
k1	13	3	*dither*	23	5
k5	17	3	*dct*	28	6
k7	21	4	*dot*	11	5
k12	19	4	*mexp*	9	4

Table 1: Moves made by pipeline refinement.

As Table 1 shows, it takes 9–28 moves to refine a pipeline. The number of moves is dependent on the complexity of the pipeline: the more stages in a pipeline, the more moves refinement makes. For example, *gsm* is a complex design with 15 stages, and it takes pipeline refinement 24 moves to identify the final pipeline. As the table shows, refinement takes 2–5 times the number of optimal moves to find a good design. Refinement makes more moves than optimal because it may move a particular stage a number of times before converging on a good design. Although pipeline refinement does not make an optimal number of moves, it does find a good pipeline quickly, especially in comparison to exhaustive search. For *gsm,* a simple implementation of exhaustive search considers approximately 6.2×10^9 designs (estimated by *13!* since the instruction fetch and register file stages are not moved), while iterative refinement considers only 24 designs. However,

exhaustive search finds an optimal design; iterative refinement does not necessarily do so.

Program	Sim.	Refine	Program	Sim.	Refine
fir	47s	11m48s	*gsm*	56s	22m15s
k1	12s	2m34s	*dither*	17s	6m20s
k5	20s	5m35s	*dct*	36s	16m56s
k7	40s	13m56s	*dot*	17s	3m10s
k12	6s	1m55s	*mexp*	13s	1m54s

Table 2: Simulation and refinement time

As an indication of how long the pipeline refinement process takes to identify the best design, Table 2 lists average simulation time for each benchmark (for a single design) and total time for refinement. The refinement times were estimated by multiplying the average simulation time for each benchmark by the total number of moves made by pipeline refinement. These numbers were collected on a Sun UltraSPARC II 360 MHz.

The simulation time for each benchmark is affected by the number of instructions executed and the complexity of each design. For the benchmarks in Table 2, the WCFP simulator executes 3509–6630 operations per second, with an average of 5039 operations per second. These simulator latencies include JVM and simulator initialization; the actual number of operations executed per second after initialization is much higher. Table 2 also shows total refinement time. The table shows to refine a pipeline takes between 1 minute and 54 seconds to 22 minutes and 15 seconds (the average is 8 minutes and 29 seconds). This performance is very good for an initial prototype and demonstrates the effectiveness of our design system. It is worth noting that the performance of the design system could be greatly improved by using a native Java compiler or re-implementing the simulator as a production quality system. As a basis of comparison, the simulator's execution engine is 3.2–17.4 times faster when not modeling microarchitecture behavior. For our initial prototype, the trade-off of performance for ease of design from using Java was well worth it.

4.2. Performance

Figure 9 shows the relative speedup of custom WCFPs built using our design system versus a general-purpose CFP. The custom WCFPs have an instruction width of 4 with memory, multiplier, and divider sidings that can initiate up to two such operations per instruction. The general-purpose CFP has an instruction width of 1 with an integer ALU stage that executes integer operations, a branch stage, and memory, multiplier, and divider sidings. The general-purpose and custom pipelines have the same operation latencies

The figure shows that the custom pipelines achieve speedups of 1.7–6.6 with an average speedup of 3.7. The speedup of the custom pipelines can be attributed to two factors: first, the custom pipelines can issue up to four operations per instruction, and second, the custom pipelines are arranged to match the data flow and execution behavior of each benchmark. The increase in instruction width has a large impact on performance and accounts for much of the speedup. For several of the benchmarks (*fir*, *k1*, *gsm*, *dot*, and *dct*), the performance improvement is approximately four times greater than the general-purpose pipeline, which is the same as the increase in instruction width. However, this improvement is not entirely due to an increase in instruction width. For these benchmarks, the average width of kernel instructions (i.e., instructions in the software pipeline kernel) is 3.4 operations, while the performance improvement is closer or greater than 4. In fact, the initial pipelines determined by pipeline extraction for these five benchmarks have an average speedup of 3.1 versus a speedup of 4.1 after refinement.

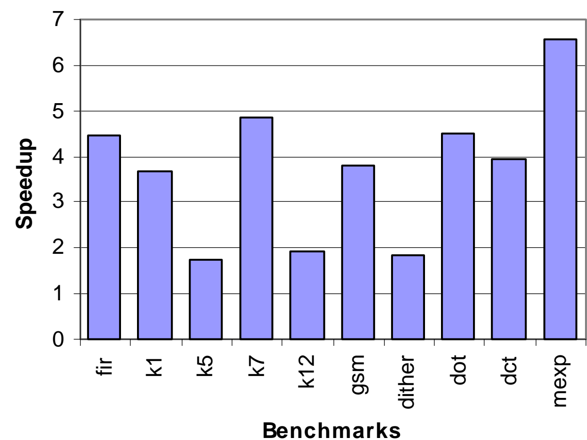


Figure 9: Speedup of custom WCFPs.

Following refinement, stages are organized to take advantage of the dynamic movement of instructions and results, which increases performance to close or above the improvement implied by the increase in instruction width. The *k5*, *k12*, and *dither* benchmarks have a performance improvement of only 2. In this case, the speedup is constrained by instruction width: these benchmarks have 1.6, 2.2, and 2.52 operations on average per kernel instruction. Two of the benchmarks—*k7* and *mexp*—have a speedup that is well above 4. In these cases, the custom pipelines are well balanced and allow instructions to progress rapidly without causing many pipeline stalls.

The results in Figure 9 demonstrate the WCFP works well for the design of ASIPs. The results are encouraging given the simple and composable nature of the WCFP. Furthermore, we have shown elsewhere that WCFP's achieve performance that is within 10% of custom VLIW organizations that have had their resources tailored to the

needs of specific embedded applications [6]. This is very encouraging since the WCFP has local communication, which should lead to very fast implementations.

5. Related Work

Although the CFP was proposed as an asynchronous general-purpose processors [18], there has also been a proposal for synchronous version [14]. However, this work adds significant hardware structures to the original design to get good performance on a wide variety of applications.

There has also been much interest in automated design of ASIPs because of the increasing importance of high-performance and quick turn-around in the embedded systems market. ASIP techniques typically address two problems: ISA and microarchitecture synthesis. ISA synthesis identifies micro-operations in a program that can be combined to create instructions [13]. The generated ISA is optimized to meet design goals such as minimum program size and execution latency. Microarchitecture synthesis derives a processor organization for an application. Many microarchitecture synthesis systems use a co-processor to synthesize custom logic for a portion of an application and integrate the custom hardware with an embedded processor core [10, 16]. Another approach tailors a single processor to the resource requirements of the target application [8, 11]. Many co-design systems unify ISA and microarchitecture synthesis into a single framework [12].

6. Summary

This paper describes an infrastructure for the automatic synthesis of ASIPs using the wide-issue counterflow pipeline organization. In the paper, we describe our design environment, which includes a design database for describing functional devices to include in a custom processor, a graph-based intermediate representation for synthesis and simulation, and a highly reconfigurable simulator and performance analysis tool. We also present a brief overview of our WCFP architecture and design methodology. The paper demonstrates that custom WCFPs improve application performance for several embedded computation kernel loops by 1.8 to 6.6 times over a general-purpose CFP.

References

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

[2] Burger D. and Austin T. M., “The SimpleScalar tool set, version 2.0”, TR #1342, Computer Science Dept., Univ. of Wisconsin-Madison, June 1997.

[3] Childers B. R. and Davidson J. W., “A design environment for counterflow pipeline synthesis”, *Workshop on Languages, Compilers, and Tools for Embedded Systems*, Montreal, Canada, June 1998.

[4] Childers B. R. and Davidson J. W., “Architectural considerations for application-specific counterflow pipelines”, *20th Conf. on Advanced Research in VLSI*, Atlanta, GA, March 1999.

[5] Childers B. R. and Davidson J. W., “Automatic architectural synthesis of wide-issue counterflow pipelines”, *Compiler and Architecture Support for Embedded Systems Workshop*, Washington, DC, October 1999.

[6] Childers B. R. and Davidson J. W., “Automatic design of custom wide-issue counterflow pipelines”, UVA Computer Science Technical Report CS–99–02, January 1999.

[7] Cmelik B. and Keppel D., “Shade: A fast instruction set simulator for execution profiling”, *Conf. on the Measurement and Modeling of Computer Systems*, pp. 128–137, May 1994.

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

[9] Davidson J. W. and Fraser C. W., “The design and application of a retargetable peephole optimizer”, *ACM Trans. on Programming Languages and Systems*, pp. 191–202, Vol. 2, No. 2, April 1980.

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

[11] 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, CA, 1996.

[12] 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.

[13] 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.

[14] Miller M. F., Janik K. J., and Lu S-L, “Non-stalling counterflow architecture”, *4th Symp. on High-Performance Computer Architecture*, pp. 334–341, Las Vegas, NV, Feb. 1998.

[15] 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, CA.

[16] 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, CA.

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

[18] 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.