Rapid Prototyping of Application-Specific 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 (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 ASIP's. This new architecture, called a wide counterflow pipeline (WCFP), is based on the counterflow pipeline organization proposed by Sproull, Sutherland, and Molnar. Our ASIP synthesis technique uses software pipelining and design-space exploration to generate a custom WCFP and instruction set for an embedded application. This type of architecture synthesis requires an infrastructure for rapidly prototyping ASIP's to evaluate design trade-offs. This paper presents the requirements and implementation of such an environment for automatic design of WCFP's. First, we describe a database for specifying design elements and architectural constraints. Second, we present an intermediate representation for WCFP synthesis and reconfigurable simulation. Finally, we describe a fast and reconfigurable simulation methodology for WCFP's.

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. We are researching techniques for automatically designing such processors based on the counterflow pipeline (CFP) architecture.

Our research uses an application expressed algorithmically in a high-level language as a specification for a CFP. This type of aggressive and automatic 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 ASIP's automatically;
- Be able to quickly evaluate cost–performance trade-offs.

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

2: Background

We have extended the original CFP proposal [9] to a wide-issue counterflow pipeline (WCFP) that is appropriate for the automatic design of custom instruction-level parallel processors [2,3]. We have used the infrastructure described in this paper to construct WCFP's tailored to embedded kernel loops. This work showed that automatically designed WCFP's achieve performance competitive with traditional application-specific VLIW architectures at a low development cost.

In this section, we briefly describe the WCFP architecture and our methodology for constructing custom WCFP's. This background information motivates the discussion of our prototyping environment.

2.1: Wide Counterflow Pipelines

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 register's 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 registers of instructions that have exited the pipeline.

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 pipe-

line stage of an appropriate type without using a siding. For example, the `add_add` stage executes two addition operations simultaneously.

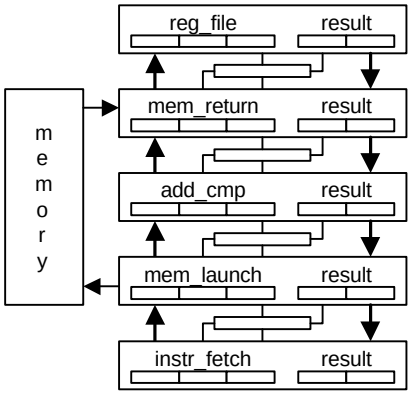

Figure 1: An example wide counterflow pipeline.

The instruction and results pipelines interact: instructions copy values to and from the result pipe. This interaction is governed by rules that ensure sequential execution semantics. For a more complete description of the WCFP architecture, see Childers and Davidson [2].

2.2: Design Strategy

Our design technique generates a WCFP customized to the resource requirements of an application's kernel loop to improve overall performance [2]. The customization process operates at the architectural-level on predesigned functional devices such as pipeline stages, register files, and functional sidings.

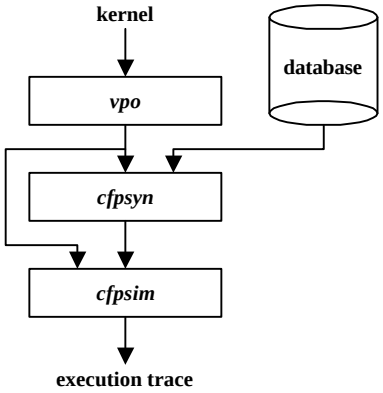

Figure 2: The synthesis system uses a kernel loop optimized by *vpo* as a specification for a custom WCFP determined by *cfpsyn*. The custom WCFP is simulated and analyzed by *cfpsim*.

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 *design database* of computational elements that indicates device type (siding or stage) and semantics for each database entry.

Figure 2 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 code optimizations such as strength reduction, induction variable elimination, global register allocation, loop invariant code motion, etc.

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.

2.3: Custom Pipelines

The optimized instructions emitted by *vpo* and the synthesis database are used to derive a WCFP. The synthesis process has four steps:

1) *Software pipelining*: A software pipelined loop is constructed from the instruction sequence emitted by *vpo*. The design database gives resource and latency constraints used during the formation of the software pipeline loop.

2) *Pipeline extraction*: The software pipeline kernel specifies the operations and functional elements to include in a WCFP. Pipeline extraction 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. During pipeline extraction, an intermediate representation of a WCFP is constructed.

3) *Instruction set extraction*: This step determines an instruction set architecture (ISA) for a WCFP, including instruction format layout and opcode assignment. The synthesized processor's IR is annotated with information about its ISA.

4) *Code generation*: This step generates the full instruction schedule using the IR to emit and build WCFP instructions.

After constructing a WCFP, it is simulated to gather performance statistics. We have found that performance can be improved by executing the kernel loop and adjusting pipeline stage order to match the kernel loop's execution behavior.

3: Design Database

Our prototyping 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. We expect the design database to 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.

The database has the minimum information needed for synthesis and simulation. This includes 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 dependence graph node. Database entries also have an attribute for listing a device's external connection ports. Ports describe how a WCFP's 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`).

```
1   database "Example" {
2   // an addition stage
3   stage "addition" {type=execute;
4       latency=5; cost=0.5;
5       opcodes={ADD,ADDI};
6       semantics={
7           r[$]=r[$]+r[$];
8           r[$]=r[$]+imm(16); };
9       ports={i_in:in,i_out:out,
10          r_in:in,r_out:out}; }
11  // a memory siding
12  siding "memory" {type=rlu;
13      latency=15; depth=3; width=2;
14      cost=2.0; opcodes={LD,ST};
15      launch_opcodes={LD,ST};
16      return_opcodes={LD};
17      semantics={
18          ld: r[$]=M[r[$]+r[$]];
19          st: M[r[$]+r[$]]=r[$]; };
20      disallow={<st,st>};
21      ports={l_in:in,r_out:out}; }
```

Table 1: An example CFP database with an addition stage and a memory siding.

Database entries also have an attribute for device semantics, which are used to automatically configure WCFP simulations to support different instruction sets. Device semantics specify the effect of opcodes in a device's repertoire. Register transfer lists (RTL's) are used to describe semantics because RTL is a small language that concisely captures instruction effects [5].

An example of device semantics is shown on lines 6–8 of Table 1 for an addition stage. This stage does two operations: *register–register* addition and r*egister–immediate* addition. The RTL on line 7 indicates that arbitrary registers may be added together with the sum placed in a register, and the RTL on line 8 specifies that a register and a 16-bit immediate can be added together.

4: Pipeline Intermediate Representation

WCFP synthesis uses a *pipeline intermediate representation* (PIR) to specify a WCFP configuration. 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 on-the-fly 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 lead to 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.

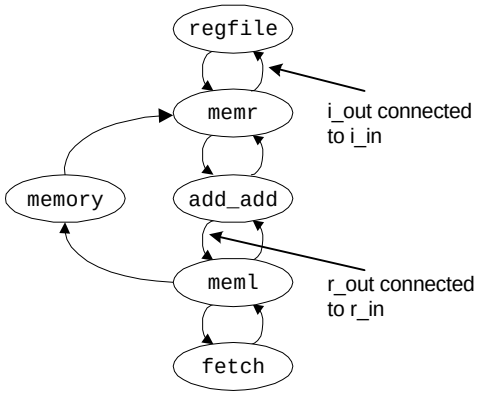

Figure 3: The PIR is a directed graph with device entities (nodes) and port connections (edges).

The PIR is a directed graph as shown in Figure 3. Nodes are entities with attributes and edges are port connections. The PIR can be used to generate simple structural VHDL (which is useful for timing analysis, cost-estimation, etc.)

A *pipeline description* is a textual form of the PIR. Pipeline extraction emits a description which is used to configure a WCFP simulation. The example in Table 2 configures a counterflow pipeline with five stages and one siding. To specify a WCFP, the PIR has three parts: global literals, stage instantiation, and siding instantiation. Global literals specify memory size, register file geometry, and result pipeline width. Table 2 illustrates a CFP with a 64K memory, a register file with 32 general-purpose registers, and a double wide result pipeline.

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.

```
1   pipeline “Example” {
2   // memory has 64K bytes
3   memory {size=64K;}
4   // reg file has 32 32-bit regs (r0-r31)
5   registers {r:<size=32,width=32>};
6   // results hold 2 reg name-value bindings
7   resultwidth=2;
8   // there are 5 pipeline stages
9   stage “fetch” {type=fetch;
10      latency=3; }
11  stage “meml” {type=launch;
12      opcode={LD,ST}; latency=3; }
13  stage “add_add” {type=execute;
14      opcode={ADD_ADD}; latency=5;
15      operations={<add,add>}; }
16      semantics={<r[$4]=r[$0]+r[$1],
17          r[$5]=r[$2]+r[$3]>; }
18  stage “memr” {type=return;
19      opcode={LD}; latency=3; }
20  stage “regfile” {type=regfile;}
21  // a memory siding for loads & stores
22  siding “memory” {type=rlu;
23      opcode={LD,ST}; latency=15;
24      depth=3;
25      semantics={
26          r[$2]=M[r[$0]+r[$1]];
27          M[r[$1]]+r[$2]]=r[$0]; }
28      l_in=meml.l_out;
29      r_out=memr.r_in; } }
```

Table 2: An example CFP description with a memory siding and five stages.

Table 2 lists an addition stage on lines 13–17. The `add_add` stage is an execution stage that does two addition operations in five simulation time units. A stage’s opcode corresponds to the instruction that executes in the stage (in this case, it is given as the `ADD_ADD` mnemonic.) The description gives semantics of synthesized devices; the `add_add` stage has an RTL that indicates the stage can do two register–register additions:

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

These semantics were generated by combining the RTL’s of the operations that compose the `add_add` stage. The stage adds the first and second source operands of an `ADD_ADD` instruction, 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.

The final section of a CFP description lists functional sidings. Lines 18–20 of Table 2 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 `meml` launch and `memr` return stages.

CFP pipeline descriptions are small structural specifications. The largest description our synthesis system has generated was for an implementation of Floyd-Steinberg image dithering. This description listed 17 devices (stages and sidings) in 131 lines of text. Most descriptions are under 100 lines long.

Because the descriptions are simple, they are easily manipulated by pipeline extraction to construct a WCFP. Their simple nature also makes it easy to automatically configure our simulator to evaluate designs.

5: CFP Simulation

Simulation is an important step in the development of a computer architecture because it lets engineers test ideas and techniques prior to building actual hardware. For embedded system development, it lets designers make trade-offs between hardware and software functionality.

5.1: Device Modeling

The WCFP simulator can be configured for different counterflow pipeline architectures. This includes changing pipeline structure, device repertoire, and instruction set features. To make automatic architecture design feasible, a structural IR is used to configure the simulator. This differs from parameterized simulators such as SimpleScalar [1] and Shade [4] which 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.

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 (RTL’s) to simulate instruction effects.

Because stage semantics are specified by a pipeline description, simulation models do not need to be written for every custom WCFP. Separating semantics from

structural pipeline models also has the advantage that the execution engine can be used independently to verify functional correctness without the expense of timing-accurate simulation.

Reconfigurable simulation is key to WCFP synthesis because it makes it possible to automatically construct and run a simulation for a WCFP to collect statistics and evaluate design trade-offs. The flexibility of the simulator eliminates the need to write simulation models for every custom pipeline. This has lead to a highly maintainable and stable system because the likelihood of introducing errors by writing new simulation models is non-existent. Furthermore, verification of the simulator needs to be done only once during initial development. It is the absence of these steps—writing and verifying models—that make automatic design feasible.

5.2: Execution-Driven Simulation

We use execution-driven simulation to model WCFP's 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.

5.3: 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 240 designs per hour (including synthesis and simulation) on an Intel Pentium II 333 MHz Linux 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.

6: Related Work

There have been many proposals for description languages that could be used in an ASIP design framework. This includes VHDL and the domain-specific HMDES [6] (used by the IMPACT compilation system.) However, it is not easy to describe instruction semantics with these languages. nML is an instruction description language for hardware/software co-design; however, it doesn't describe processor structure [7]. Hawk combines processor structure and instruction semantics in a single description [8]. Nevertheless, this language is too powerful for our needs: it requires full lexical analysis, parsing, and semantic handling.

7: Summary

The automatic design of application-specific integrated processors requires a prototyping environment to make cost–performance trade-offs. This paper describes such an environment for the design of custom counterflow pipelines. First, we present a database for specifying WCFP design elements and resource constraints. Second, we describe an intermediate representation for constructing and simulating WCFP's. Finally, we describe a methodology for quickly simulating and evaluating WCFP design alternatives.

References

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

[2] B.R. Childers and J.W. Davidson, "Application-specific wide-issue counterflow pipelines", submitted for conference publication, available from: `http://www.cs.virginia.edu/~brc2m/cfp`.

[3] B.R. Childers and J.W. Davidson, "Architectural considerations for application-specific counterflow pipelines", to appear, *25th Conf. on Advanced Research in VLSI*, Atlanta, GA, March 1999.

[4] B. Cmelik and D. Keppel, "Shade: A fast instructionset simulator for execution profiling", *Proc. of Conf. on the Measurement and Modeling of Computer Systems*, pp. 128–137, May 1994.

[5] J.W. Davidson and C.W. Fraser, "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.

[6] J.C.Gyllenhaul, W.M. Hwu, and B.R. Rau, "HMDES version 2.0 specification", Tech. Report IMPACT–96–3, Univ. of Illinois at Urbana-Champaign, 1996.

[7] M.R. Hartoog et al., "Generation of software tools from processor descriptions for hardware/software co-design", *Design Automation Conference*, pp. 303–306, Anaheim, CA, 1997.

[8] J. Matthews, B. Cook, and J. Launchbury, "Microprocessor specification in Hawk", *IEEE Int'l Conf. on Computer Languages*, pp. 90–101, May 1998.

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