A Design Environment for Counterflow Pipeline Synthesis

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

Abstract

The Counterflow Pipeline (CFP) organization may be a good target for synthesis of application-specific microprocessors for embedded systems because it has a regular and simple structure. This paper describes a design environment for tailoring CFP's to an embedded application to improve performance. Our system allows exploring the design space of all possible CFP's for a given embedded application to understand the impact of different design decisions on performance. We have used the environment to derive heuristics that help to find the best CFP for an application. Preliminary results using our heuristics indicate that speed-up for several small graphs range from 1.3 to 2.0 over a general-purpose CFP and that the heuristics find designs that are within 10% of optimal.

1. Introduction

Application-specific microprocessor design is a good way to improve the cost-performance ratio of an application. This is 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 [15], has several characteristics that make it a possible target organization for the synthesis of application-specific microprocessors. The CFP has a simple and regular structure, local control, high degree of modularity, asynchronous implementations, and inherent handling of complex structures such as result forwarding and speculative execution.

Our research aims to use an application expressed algorithmically in a high-level language (such as C) as a specification for an embedded application-specific microprocessor. The counterflow pipeline organization is a good candidate for this type of fast, aggressive high-level 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.

We have built a framework for exploring the design space of application-specific counterflow pipelines. The environment includes tools for program optimization, synthesis experimentation, performance analysis, and design simulation.

Because a CFP design space is potentially very large, we have developed heuristics that narrow the space to pipeline configurations with good performance. Our design framework has been instrumental in deriving these heuristics and understanding the factors that are important for good performance from CFP designs.

This paper presents our initial design environment and synthesis heuristics. The first section contains introductory material covering our synthesis strategy, related work, and the counterflow pipeline organization. The second section describes our design environment and the third section discusses our synthesis heuristics. The fourth section presents preliminary results for several benchmarks using those heuristics, and the final section concludes with future work.

1.1. Synthesis 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. According to Amdahl's Law [9], increasing the performance of the most frequently executed portion of an application increases overall performance. Thus, synthesizing custom hardware for the computation-intensive portion may be an effective technique to increase performance.

The type of applications we are considering need only a modest kernel speed-up to effectively improve overall performance. For example, JPEG has a function `j_rev_dct()` that accounts for 60.38% 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 CFP synthesis. Figure 1 shows a plot of Amdahl's Law for various speed-up

values of `j_rev_dct()`. The figure shows that a small speed-up of the kernel loop of 6 or 7 achieves most of the overall speed-up.

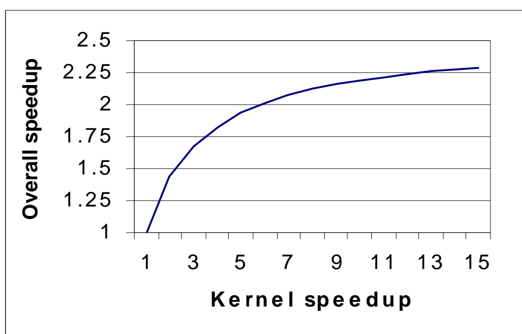

Figure 1: Overall speed-up for JPEG

Our synthesis system uses the data dependency graph of an application's kernel to determine processor functionality and interconnection. Processor functionality is determined from the type of operations in the graph and processor topology is determined by exploring the design space of all possible interconnection network.

The target system architecture for our synthesis technique has a single CFP processor that executes the control and computation portions of an application. Synthesis customizes a general-purpose CFP for the kernel computation to improve performance. This is similar to software acceleration using a co-processor; however, there is only *one* processor in this scheme and, as a result, overall cost should be lower than with a co-processor scheme while still improving performance. A single CPU architecture has the advantage that it does not need any support for synchronization with an attached coprocessor (e.g., interface logic, handling of live-in/out data, etc.).

1.2. Related Work

Recently there has 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 [6]. ASIP techniques typically address two broad problems: instruction set and micro-architecture synthesis. Instruction set synthesis attempts to discover micro-operations in a program (or set of programs) that can be combined to create instructions [10,11]. The synthesized instruction set is optimized to meet design goals such as minimum program size and execution latency. Micro-architecture synthesis derives a microprocessor implementation from an application (or set of applications). Many micro-architecture 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 [4,8,13]. Another micro-architecture synthesis approach tailors a single processor to the resource requirements of the target application [3,5]. Although instruction set and micro-architecture synthesis can be treated independently, many co-design systems attempt to unify them into a single framework [2,7].

Our current research focus is micro-architecture synthesis. We do not presently synthesize an instruction set for an embedded application. Instead, we customize a counterflow pipeline micro-architecture to an application using a standard RISC instruction set and information about the data flow of the target application. Our micro-architecture synthesis technique has the advantage that the design space is well defined (although potentially very large), making it easier to search for pipeline configurations that meet design goals.

1.3. Counterflow Pipeline

The counterflow pipeline has two elastic pipelines flowing in opposite directions. One is the instruction pipeline. It 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 is the results pipeline that conveys results from the register file to the instruction fetch stage. The instruction and results pipelines interact: instructions copy values to and from the result pipe.

Functional units (or *sidings*) are connected to the pipeline through *launch* and *return* stages. Launch stages issue instructions into functional units and return stages extract results from functional units. Instructions may execute in either pipeline stages or functional units.

A memory unit connected to a CFP pipeline is shown in Figure 2. Load instructions are fetched and issued into the pipeline at the `instr_fetch` stage. A bundle is created that holds the load's memory address and destination register operands. The bundle flows towards the `mem_launch` stage where it is issued into the memory subsystem.

When the memory unit reads a value, it inserts the value into the result pipeline at the `mem_return` stage. In the load example, when the load reaches the `mem_return` stage, it extracts its destination operand value from the memory unit. This value is copied to the destination register value in the load's instruction bundle and inserted into the result pipe. A *result bundle* is created whenever a value is inserted into the result pipeline. A result bundle has space for the result's name (i.e., register name) and value. Results from sidings or other pipeline devices flow down the result pipe to the `instr_fetch` stage. Whenever an instruction and a

result bundle meet in the pipeline, a comparison is done between the instruction operand names and the result name. If a result name matches an operand name, its value is copied to the corresponding operand value in the instruction bundle. When instructions reach the `reg_file` stage, their destination values are written back to the register file and when results reach the `instr_fetch` stage, they are discarded. In effect, the register file stores results that have exited the pipe.

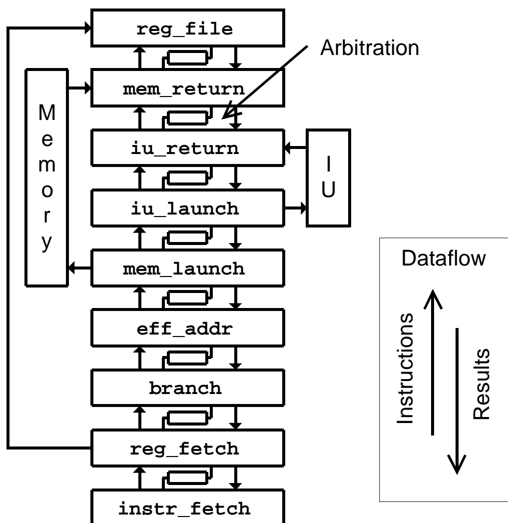

Figure 2: Example Counterflow Pipeline

The interaction between instruction and result bundles are governed by special *pipeline* and *matching rules* that ensure sequential execution semantics. These rules govern the execution and movement of instructions and results and how they interact.

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 controls purging the pipeline on an exception. A *poison pill* is inserted in the result pipeline whenever a fault is detected. The poison pill purges both pipelines of all instruction and result bundles. This purge mechanism can also be used for speculative execution when a branch target is mispredicted.

As Figure 2 shows, stages and functional units are connected in a very simple and regular way. The connections correspond to bundled interfaces of micropipelines. 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.

1.4. Execution Model

Our synthesis system models asynchronous counterflow pipelines by varying computational latencies. Table 1 shows the latencies we use in our simulation models.

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

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 instruction operation such as addition takes 5 cycles. 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 cycles. In the rest of this paper, we refer to time units as “clock cycles”, although this should not be confused with the notion of a clock cycle in a synchronous processor. A clock cycle for the counterflow pipeline is a very small time unit such as a few gate delays.

The instruction set we use is a subset of the SPARC V8 [14] instruction set architecture. The instruction subset does not support register windows or tagged operations, but is representative of modern instruction sets for embedded RISC cores.

2. Experimental Framework

Figure 3 depicts the synthesis framework. The system

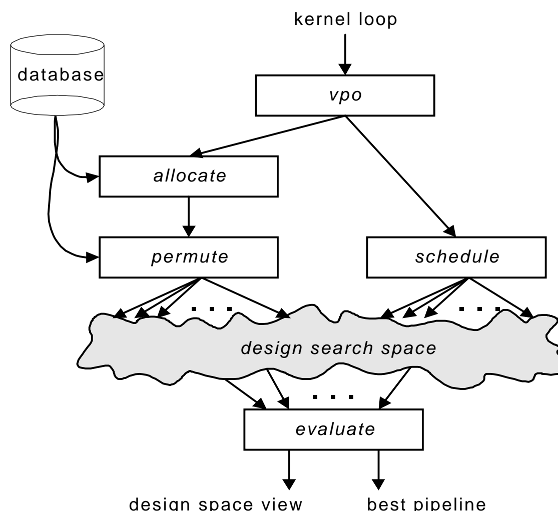

Figure 3: Synthesis framework

accepts a C source file containing a loop that drives customizing a CFP processor. The loop is compiled by the optimizer *vpo* [1] into instructions for the SPARC architecture. During optimization, *vpo* builds the data dependency graph for the kernel loop. The graph serves as an input to the rest of the synthesis system, which consists of modules for hardware allocation, processor topology determination, instruction scheduling, simulation, and performance analysis.

2.1. Hardware Allocation

The first step of our synthesis system is hardware allocation, which is done by the *allocate* module. Hardware allocation determines processor functionality by assigning computational elements to data dependency graph nodes. After allocating hardware resources, *allocate* emits a list of functional elements in the synthesized processor and a mapping of graph nodes to those elements.

allocate uses a design database to determine what hardware devices to assign graph nodes. The database has entries for pipeline stages and functional sidings. Each entry has fields for a device name, a list of instruction opcodes, operation latency, siding depth and device type. The device name is a string that uniquely identifies the device in the database. The opcode list gives the semantics of the device in terms of the instructions it executes. The list is used by *allocate* to select what kind of devices to allocate to graph nodes. In the current implementation of *allocate*, an instruction opcode may be listed in only one database entry (i.e., multiple devices may not implement the same opcode). The operation latency field gives the latency of the device, and the depth field, if the device is a siding, gives the length of the siding's pipeline. The final field is a device type and may be one of three values: STAGE, LU, and RLU. The STAGE type identifies the database entry as a pipeline stage, the LU type as a functional siding that does not return a result (e.g., a unit for stores) and the RLU type as a functional unit that does return a result.

An example database with four entries is shown in Figure 4. The first two entries are pipeline stages for addition and branch resolution. The `add` stage handles two addition opcodes, one that generates a condition code and one that does not. The `branch` stage resolves branch predictions for eight different branch opcodes. The database in Figure 4 also has entries for two functional sidings. The `multiply` siding does either signed or unsigned multiplication in 20 cycles and has a siding depth of 4. The `memory` siding has a pipeline depth of 3 and executes load and store instructions in 15 cycles. This siding handles all types of load and store instructions, including variants for word, halfword, and byte data types.

```
database “example” {
   entry “add” {
      type=STAGE; latency=5;
      opcodes={add, addcc}; };
   entry “branch” {
      type=STAGE; latency=5;
      opcodes={bne, be, bg, ble,
         bge, bl, bgu, bleu}; };
   entry “multiply” {
      type=RLU; latency=20;
      depth=4;
      opcodes={umul, smul}; };
   entry “memory” {
      type=RLU; latency=15;
      depth=3;
      opcodes={ldsb, ldsh, ldsw,
         stb, sth, stw}; };
}
```

Figure 4: Example design database

The design database makes it easy to partition processor functionality among pipeline stages and functional sidings. A designer can use the database to experiment with different partitions of functionality. In our experiments, we use a database that consists of unique devices for every opcode group in the SPARC instruction set architecture. For example, there is a single device in the database for unsigned and signed multiplication. We also partition functionality according to operation latency: pipeline stages execute low latency operations and functional sidings execute high latency operations.

Pipeline stages in the synthesized processor are determined from the type of device a graph node is assigned. *allocate* generates an unique pipeline stage of an appropriate type for every graph node that is assigned a STAGE device. If a node is assigned a functional siding, then *allocate* generates two pipeline stages for RLU devices and one stage for LU devices. An RLU device has two stages since it launches an instruction and returns a result. An LU siding needs only a single stage to launch instructions since it does not return results to the main pipeline. Some pipeline stages are always included in a CFP such as stages for instruction fetch, a register cache, and a register file.

allocate may constrain the position of pipeline stages in a given processor topology. For example, *allocate* ensures that a siding's return stage appears after the siding's launch stage. It further ensures that the return stage appears at least *depth* stages later than the launch stage, where *depth* is the depth of the siding's pipeline. *allocate* also consults design heuristics to determine pipeline stage position constraints. These constraints limit

the number of pipeline topologies considered by the synthesis system.

A hardware functionality description is emitted by *allocate* that lists the types of functional elements a synthesized processor should contain and their position constraints. The description is a list of instantiated computational elements from the device database and the mapping of graph nodes to pipeline stages and sidings.

2.2. Processor Topology

The *permute* module uses the specification from *allocate* to derive CFP topologies that obey the position constraints imposed by *allocate*. The topologies are determined by permuting the order of pipeline stages, which fully specifies a CFP because processor structural elements are interconnected using pipeline stages.

permute emits a processor design specification for each topology it generates. The specification includes the order of pipeline stages and the functional characteristics of each device in the synthesized processor. The specification has attributes for pipeline stages and sidings that specify device latency, type, siding depth, and semantics (as instruction opcodes). The attributes are used by the CFP simulator to model device behavior and pipeline resources.

```
pipeline "example" {
  stage "fetch" {
     type=instr_fetch; };
  stage "mulL0" {
     latency=3; type=launch;
     operations={smul, umul}; };
  stage "add0" {
     latency=5;
     operations={add, addcc}; };
  stage "mulR0" {
     latency=3; type=return;
     operations={smul, umul}; };
  stage "rf" {
     type=reg_file; };
  siding "mul" {
     latency=20; depth=4; type=RLU;
     launch="mulL0"; return="mulR0";
     operations={smul, umul}; };
}
```

Figure 5: Example CFP specification

Figure 5 shows an example processor specification. The example has pipeline stages for instruction fetch, multiply launch and return, addition, and register file write-back, as well as a siding for multiplication. The addition stage has a latency of 5 and executes `add` and `addcc` instructions. The multiply launch and return stages (`mulL0` and `mulR0`) launch and return unsigned and signed multiplication operations into and from the multiplier siding. The multiplier siding has a latency of 20 and a pipeline depth of 4.

2.3. Instruction Scheduling

vpo emits optimized SPARC instructions that serve as input to a *schedule* module that determines all legal instruction schedules. Branch and bound techniques can be used during design evaluation to reduce the number of schedules considered for each pipeline configuration.

2.4. Processor Simulation

The *cfpsim* module simulates a counterflow pipeline design. A CFP simulation is specified by a pipeline configuration, an instruction schedule, and a mapping of instructions to pipeline devices. The pipeline specification is emitted by *permute*, the mapping of graph nodes to computational devices by *allocate*, and the instruction schedule by *schedule*. The simulator generates an execution trace of a program indicating the state of pipeline resources on every clock cycle. The trace is used by a performance analysis tool to evaluate a design.

The CFP simulator models resource usage and instruction and result flow; it does not model microarchitectural devices. For this reason, the simulator is very fast and cycle accurate. The simulator uses a "tiling approach" to model hardware resources and processor data flow. *cfpsim* tiles a time-resource diagram with the resource usage and data flow of every dynamically executed instruction.

The resource usage of an instruction is given by a reservation table, called an *instruction tile,* that depicts the resources the instruction uses for every clock cycle starting at cycle 0. Results generated by an instruction are modeled in the reservation table as the resource usage of result pipeline registers.

The instruction tile may be inferred from the pipeline specification. The specification indicates what operations use which resources and for how long, and from this information, the simulator is able to construct a tile for every instruction in the ISA.

Figure 6 shows an example tile for an addition instruction using the CFP specification from Figure 5. The tile shows how long and when the instruction uses each pipeline resource. The add instruction uses the `fetch`, `mulL0`, `mulR0`, and `rf` stages for one cycle since they are needed only to convey the instruction to the next stage. Because the `add0` stage executes the add instruction, it is used for 5 cycles. A result is produced on the final execution cycle and conveyed down the result pipeline using the result registers in stages `add0`,

mulL0, and fetch for one cycle each.

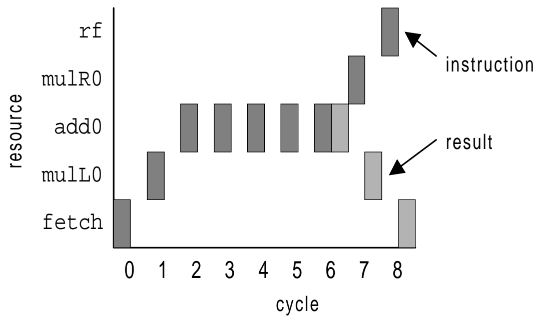

Figure 6: Instruction tile

For every instruction in the execution trace, the simulator *places* a tile in a time-resource diagram. The tile is placed starting at the most recent cycle in which the instruction fetch stage is empty. The tile is placed in the diagram so that if a particular instruction pipeline stage is not available at a needed cycle, the instruction is delayed by a cycle in the previous stage until the resource becomes available.

Whenever a result is encountered in a stage, the simulator models the interaction of the result and the instruction (i.e., the comparison operation). This is done by delaying the movement of both the instruction and result until the comparison operation is finished. Because the result has already been placed in the time-resource diagram, its resource usage must be undone for all cycles after the cycle when the comparison operation began. After undoing the result's resource usage, it can be re-placed in the diagram beginning on the last cycle of the comparison operation. It is possible to introduce result register conflicts during the re-placement of a result. Such conflicts are resolved by delaying the conflicting result one cycle in the previous stage (and, hence, removing the resource conflict). This can be done by recursively delaying each result that introduces a conflict.

Care must be taken when a result is delayed in a pipeline stage that contains an instruction. In this case, the instruction may need to be re-placed if delaying the result also causes the instruction to be delayed. Our simulator handles this by backtracking the placement of instructions to that point. We have found that backtracking is rarely required with most programs, so it does not typically reduce simulation performance.

2.5. Performance Analysis

The cross-product of pipelines from *permute* and instruction schedules from *schedule* defines a CFP design space. This space is large. For example, the finite impulse response filter data dependency graph (12 nodes) produces 2,580,480 pipelines and 42,735 schedules.

A module *evaluate* traverses the design space to collect performance statistics for every processor/schedule combination. *evaluate* post-processes the execution traces generated by *cfpsim* to gather statistics about instruction and result flow, pipeline utilization and throughput. These statistics are used to pick the best

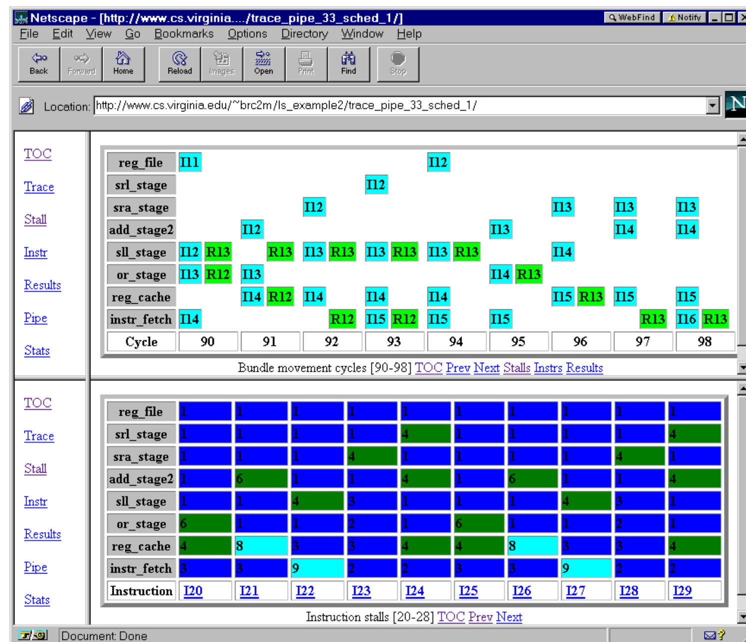

Figure 7: Visual representation of a design point

pipeline/schedule combination and to emit a visual representation of the design space. The visual representation shows the overall design space and simulation details for each design point.

Figure 7 shows the visual representation of a single design point. The top half of the figure shows resource usage for every instruction and result per cycle. The boxes labeled with **I** indicate instructions as they flow from the instruction fetch stage toward the register file. The boxes labeled with **R** indicate results flowing from the stage where they are produced to where they are consumed.

The time-resource diagram is useful for understanding instruction and result flow. The tool allows any two design points side-by-side, so we can, for example, examine the best and worst designs to understand what factors affect performance. The side-by-side comparison is also useful for studying the effect of different instructions schedules on the same pipeline organization (and vice-versa).

The bottom half of Figure 7 shows the *stall density* of instructions. This graph shows how long an instruction spends in each pipeline stage. The effect of pipeline stalls can be readily identified from this diagram since cumulative pipeline stalls appear as colored diagonal bands. Whenever an instruction stalls in the pipeline, it may eventually cause resource conflicts during later cycles. An example of this can be seen in Figure 7 starting with instruction I23 and ending with I27. The stall density diagram is useful for identifying bottleneck regions in the program execution trace. Once such a region is identified, the time-resource diagram can be used to view instruction and result data flow in that region. The stall density diagram is a helpful abstraction for identifying areas in the program instruction trace for detailed study without having to examine the trace on a cycle-by-cycle basis.

3. Synthesis Methodology

We are exploring techniques for customizing counterflow pipelines to embedded applications using the design framework presented in the previous section. The environment is helpful for understanding what factors are important to get good performance from a CFP design.

The synthesis technique we use generates all processor topologies for a given set of functional units and pipeline stages to pick the best one. The processor functionality is determined from the data flow graph and design database. Although it is not apparent how to build the full design space for a traditional microprocessor organization, it is straightforward for the CFP since pipeline stage order specifies processor topology.

It is not practical to exhaustively search the entire design space for most dependency graphs because the space is not likely to be small when aggressive instruction-level parallelism transformations such as speculative and predicated execution, software pipelining, if-conversion, etc. are used. We are evaluating heuristics that reduce the size of the space while finding pipeline/schedule combinations with good performance.

From preliminary work, we have identified several factors that affect performance: the latency of conveying source operands, overlapping movement of instructions and results, avoiding resource conflicts due to the unavailability of instruction pipeline registers, concurrently executing operations from adjacent loop iterations, and not over/under speculating instructions.

3.1. Performance Factors

The distance results flow between their production and consumption affects a CFP's performance because the latency of executing instructions includes the amount of time it takes to acquire source operands. This time is affected by the result flow distance; the further a result flows, the greater the latency of the instruction. For example, widely separated producer and consumer stages can cause the consumer instruction to stall waiting for its source operands, which may also delay subsequent instructions. Although careful instruction scheduling can mask a portion of result flow latency, it is not likely to hide the entire latency. Thus, it is important to find the pipeline structure *and* instruction schedule that best hides result flow latency.

The latency of conveying a result is also affected by the number of instructions a result encounters as it flows from its producer stage to its consumer stage. A result flows more slowly through a stage containing an instruction than it flows through an empty stage because the result's register names must be compared with the instruction's source and destination names. This suggests producer and consumer instructions should be placed close together in the instruction schedule (the opposite of what traditional scheduling does) to minimize flow latency. However, from our experience, it is not always necessary for producer and consumer to be immediately adjacent. They can be separated by up to a few instructions because there is usually “enough time” to overlap the movement of the consumer instruction and its source operands, ensuring they meet before the consumer reaches its execution stage.

From our studies, it appears that overlapping the movement of results and instructions is very important for good performance. The effect of adjusting both the pipeline structure and instruction schedule is to balance the pipeline using the program's dynamic instruction and result flow. A traditional synchronous pipeline is

balanced *a priori* by a designer; however, this is not possible with the counterflow pipeline because it has dynamically varying latencies.

The importance of pipeline balancing is demonstrated by the effect of adding *blank stages* that do not perform any operation to a CFP design. One way to balance the pipeline is to insert blank stages before pipeline regions that have high resource contention. These areas can be identified by looking for diagonal bands in the stall density diagram. Inserting blanks allows an instruction contending for a busy pipeline stage to move up one position, occupying a blank stage (or series of blanks). By ensuring that instructions always make progress, subsequent instructions are more likely to flow to stages where they can begin executing. In this way, blank stages serve as queues into heavily contended pipeline regions.

It may not be necessary to insert actual blank stages. Instead, stages that are unused during a period of execution can be arranged to serve as pipeline queues. Such stages have the dual purpose to execute instructions and to act as place-holders. Using our design environment we have found that stages executing off-critical path instructions can effectively serve as queues and execute instructions in the "delay" of the critical path. It is, however, difficult to statically predict where placing these stages is most effective.

Another important performance issue is overlapping the execution of adjacent loop iterations to expose instruction-level parallelism. A CFP micro-architecture can be arranged to achieve hardware loop unrolling by ensuring that hardware resources needed by the following loop iteration are available early in the pipeline. It is best to place resources for operations which do not have loop carried dependences (between the *ith* and *ith + 1* iterations) near the beginning of the pipeline because they are the least likely to stall while waiting for source operands.

A question related to hardware loop unrolling is where to resolve branches in the pipeline. If branches are resolved near the instruction fetch stage, then very little speculative execution is possible and an opportunity to execute operations across loop iterations may be lost. However, if branch resolution is done late in the pipeline, then the misprediction penalty is very high. The location of branch resolution is important because a good CFP design should not over or under speculatively execute instructions. The ideal location for branch resolution can be found by trying the branch in all possible places in the pipeline and picking the best one.

3.2. Design Heuristics

Heuristics that use the factors mentioned above to guide the exploration of a CFP design space may narrow the space sufficiently and accurately so a good stage order and schedule are found. Using our design framework, we have been experimenting with such heuristics. These heuristics are hosted in the hardware allocation module to constrain the position of pipeline stages. Our present heuristics do not consider the instruction schedule because the data dependency graph constrains the number of schedules. In the future we will consider instruction scheduling as well.

In this paper, we discuss two pipeline layout heuristics. The first confines the search space to designs that have pipeline stages in order of the critical path:

> *Heuristic 1: For all nodes* $\{n_1, n_2, \ldots, n_k\}$ *and edges* $\{(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)\}$ *on the critical path, evaluate only designs that have the partial order* $\{n_1 \ll n_2, n_2 \ll n_3, \ldots, n_{k-1} \ll n_k\}$ *wrt. pipeline stages.*

The pipeline order is with respect to the instruction fetch stage, so the root of the critical path is the closest to instruction fetch. This order overlaps the execution of instructions from different loop iterations (loop-carried dependences may affect this, of course) while minimizing the distance results flow along the critical path. If the pipeline is arranged in the reverse order so the root of the critical path is placed near the register file, then there is no overlap (in execution) between loop iterations. In this case, an entire iteration flows into the pipeline with the last critical path instruction stalled at the bottom of the pipeline. This keeps the next iteration from entering the pipeline and beginning execution.

This heuristic lets stages that execute non-critical path instructions occur any place in the pipeline (no constraints are placed on the order). This ensures that the synthesis system finds the position for each stage that both executes instructions in the delay of the critical path and serves as a place-holder to let instructions move up the pipeline.

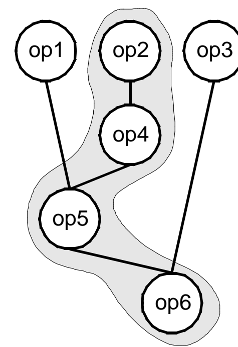

Figure 8: Example data dependency graph

Figure 8 shows an example data dependency graph

with the critical path highlighted. The partial order for this example using heuristic 1 is:

$$\{op2 \ll op4, op4 \ll op5, op5 \ll op6\}$$

For example, *op1* can appear any place, while *op5* must appear after *op4* and before *op6*. The hardware allocation module imposes this partial order on stage positions. Although this heuristic determines good pipeline layouts, in many cases synthesis must consider a large number of design points.

A second heuristic considers many fewer pipeline configurations by generating a more refined partial order. The heuristic preserves critical path order using the instruction dependency graph to define operation partitions by drawing cuts across each level of the graph.

> *Heuristic 2: For the graph $G = (N, E)$, divide N into K partitions, where K is the maximum path length in E. Use the partitions to impose a partial order on pipeline stages such that $\forall n_i \in partition_k$ and $\forall n_j \in partition_{k+1}$ then $n_i \ll n_j$ is in the partial order. Assign nodes to a partition according to some assignment heuristic and the dependence edges E. Evaluate only designs that have the partial order wrt. pipeline stages.*

The cuts determine a partial order that places operations from level *n* before operations from level *n+1* (root is level 0) in a pipeline. Figure 9 shows an example of assigning graph nodes to instruction partitions.

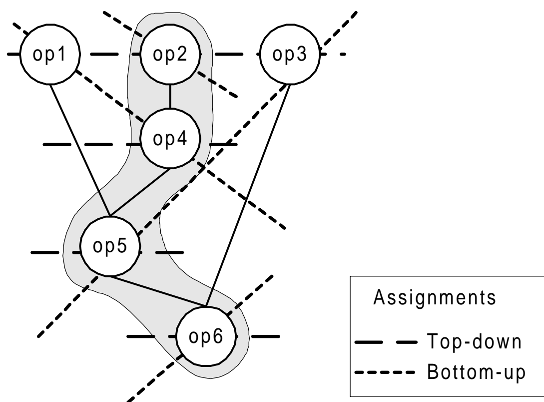

Figure 9: Greedy assignment of graph cuts

In the figure, the "top-down" assignment of operations gives the partial order:

$$\{op1 \ll op4, op2 \ll op4, op3 \ll op4, op4 \ll op5, op5 \ll op6\}$$

In this example, *op1*, *op2*, and *op3* all must occur within the first three pipeline stages (in any order), *op4* occurs in the fourth position, *op5* in the fifth, and *op6* in the sixth position.

Nodes can be assigned to different instruction partitions. There are two straightforward assignments: top-down and bottom-up. Top-down conceptually may work well since assigning nodes to early instruction cuts ensures those operations begin executing as soon as possible. Bottom-up may also work since it can minimize the distance results flow between their definition and use. For example, Figure 9 shows top-down and bottom-up assignments of operations to graph cuts. Bottom-up works best for this graph since it minimizes the distance results move between their production and consumption. Indeed, we have found that bottom-up assignment typically works well for most graphs. The pipeline stage partial order for Figure 9 with bottom-up assignment is:

$$\{op1 \ll op5, op1 \ll op3, op2 \ll op4, op2 \ll op1, op3 \ll op6, op4 \ll op5, op4 \ll op3, op5 \ll op6\}$$

This partial order puts *op2* in the first position of the pipeline followed by *op1* and *op4* in the second or third position The fourth and fifth positions have *op3* or *op5* and the sixth position has *op6*.

Heuristic 2 forms instruction groups that execute in parallel in a way similar to forming instructions for very long instruction word (VLIW) architectures. After forming instruction partitions, there is a single path through the dependency graph. Thus, the instruction groups are arranged in the order of this path (like heuristic 1). The order of operations is not constrained in an instruction partition to let the synthesis system find the optimal local arrangement of pipeline stages within a partition while minimizing result flow distance between partitions.

4. Results

Preliminary results for several small graphs are shown in Figure 10. The speed-up in the figure is relative to a general-purpose pipeline that has separate sidings for memory, multiplication, and integer operations and a pipeline stage for branch resolution. The figure shows speed-up for three pipeline orders: *optimal*, *heuristic 1*, and *heuristic 2*. In all cases, the partitioning of functionality is the same: every graph node is assigned an unique pipeline stage or siding. The *optimal* pipeline had the best performance from all pipeline stage permutations. The *heuristic 2* pipelines use late assignment to allocate graph nodes to partitions.

The benchmarks in Figure 10 are small data dependency graphs that have less than 8 nodes. The graphs contain mostly low latency integer operations, although graph 10 has two memory references and a multiplica-

tion. We selected these initial benchmarks because they were small enough to generate the full design space and demonstrate the effectiveness of the search heuristics. As we evaluate more realistic kernels with more instruction-level parallelism we expect that greater speedups will be achieved. Furthermore, a CFP may be faster than conventional processors due to the absence of global signals and better implementation technology.

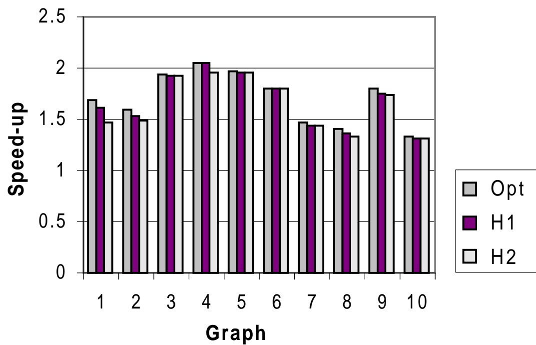

Figure 10: Speed-up for several small graphs

The figure shows that the search heuristics find pipelines that are nearly as good as optimal. The performance difference between the heuristically determined pipelines and the optimal pipeline is generally less than 10%. This difference is partly influenced by start-up cost: The optimal stage orders usually have a lower start-up penalty because they order stages to favor requesting source operands from the register file.

Both search heuristics work well. *Heuristic 2* does nearly as well as *heuristic 1,* while evaluating fewer designs. Indeed, for graphs 3, 5, 7, and 10, *heuristic 2* finds the same pipelines as *heuristic 1.*

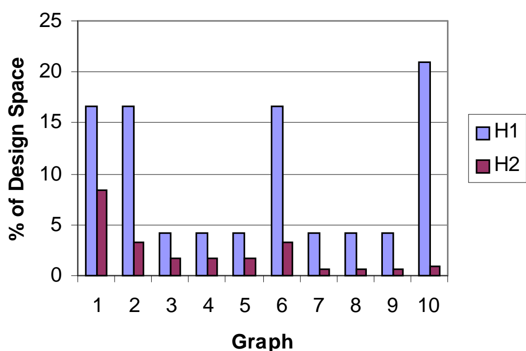

Figure 11: Size of subspace

Figure 11 shows a comparison of the number of designs evaluation by each heuristic as a percentage of the overall design space. Both heuristics reduce the search space; however, the amount of the reduction is dependent on the shape of the dependency graph.

Figure 10 and Figure 11 show that the heuristics constrain the search space to a small number of pipelines and find designs that are nearly as good as optimal. Although these initial experiments are small, we expect the heuristics to also work for full applications. Future work however, must include heuristics for instruction scheduling as well pipeline layout.

5. Conclusion

This paper describes a design environment for studying counterflow pipeline organizations. Our preliminary experiments demonstrate that the CFP is a flexible target for high-level synthesis of application-specific microprocessors. The work also shows the importance of exploring a large design space to further understanding of a new computer organization. We are continuing to research the performance potential of custom CFP's, including micro-architecture extensions that may greatly improve performance without sacrificing ease of design.

References

[1] Benitez M. E. and Davidson, J. W., "A Portable Global Optimizer and Linker", *Proc. of the SIGPLAN Notices 1988 Symp. on Programming Language Design and Implementation*, pp. 329–338, Atlanta, Georgia, June 1988.

[2] Binh N. N., Imai M, Shiomi A, et al., "A Hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts", *Proc. of 33rd Design Automation Conf.*, pp. 527–532, Las Vegas, Nevada, June 1996.

[3] 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.

[4] 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.

[5] 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.

[6] Goering R., "Codesign turns workplace on its head", *EE Times*, Monday, Jan. 12, 1998.

[7] 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.

[8] Hauser J. R., and Wawrzynek J., "Garp: A MIPS processor with a reconfigurable coprocessor", *IEEE 5th Annual Symp. on Field-Programmable Custom Comput-*

ing Machines, pp. 12–21, Napa Valley, California, April 16-18, 1997.

[9] Hennessy J. L. and Patterson D. A., *Computer Architecture: A Quantitative Approach*, *2nd edition*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

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

[11] 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.

[12] 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.

[13] 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.

[14] SPARC International, Inc., *The SPARC Architecture Manual, Version 8*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[15] 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.