PipeRench Implementation of the Instruction Path Coprocessor

Yuan Chou, Pazhani Pillai, Herman Schmit, John Paul Shen
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{yuanchou,pillai,herman,shen}@ece.cmu.edu

Abstract

This paper demonstrates how an Instruction Path Coprocessor (I-COP) can be efficiently implemented using the PipeRench reconfigurable architecture. An I-COP is a programmable on-chip coprocessor that operates on the core processor's instructions to transform them into a new format that can be more efficiently executed. The I-COP can be used to implement many sophisticated hardware code modification techniques. We show how four specific techniques can be mapped to the PipeRench pipelined computation model. The experimental results show that a PipeRench I-COP used to perform trace construction and trace optimizations for a trace cache fill unit not only achieves good performance gains but can potentially be implemented in less than 10 mm^2 (assuming 0.18 micron technology) or approximately 3% of the die area of a current high-end microprocessor. We believe these results demonstrate the usefulness and feasibility of the I-COP concept.

1 Introduction

1.1 Dynamic Code Modification

Spurred by relentless progress in VLSI design and fabrication, hardware design is evolving at a rapid pace and increasingly sophisticated microarchitectures are being implemented. On the other hand, software is changing much more slowly. One reason is the existence of a large installed base of legacy code that is too expensive to be replaced or recompiled. Another reason is that the deployment of new highly optimizing compilers usually lags behind the deployment of new microarchitectures. The end result is the increasing incompatibility between the compiler-produced object code and the most efficient implementations of fast execution cores that must execute these object code.

One recently proposed approach to solve this problem is to add hardware in the microarchitecture to dynamically modify the object code into an internal format that can be more efficiently processed by fast execution cores. We refer to this general approach as *hardware code modification*. For example, the Intel P6 [1] decoders translate the x86 instructions into an internal format called uops that are then executed by the execution core. Another example is the trace cache [2], which rearranges the ordering of instructions so that frequently executed sequences of instructions are stored in contiguous locations. The trace cache can reduce the complexity of instruction fetching and decoding. There are also proposals to optimize these traces [9][10] before loading them into the trace cache. Recently, there is a proposal to perform run-time program re-layout in hardware [25]. We believe that in the quest for ever higher performance, increasingly sophisticated hardware code modification techniques will be needed in the future.

An *Instruction Path Coprocessor* (I-COP), proposed in [3], is a programmable on-chip coprocessor that allows these hardware code modifications to be implemented in software much like microcode. An I-COP is analogous to a datapath coprocessor, except that it operates on the core processor's *instructions* themselves. The programmable nature of an I-COP affords several advantages. First, complex code modifications that are difficult to implement directly in hardwired logic may be more easily implemented in I-COP code. Second, it allows many code modification techniques to be implemented using the same engine, each of which can be selectively and adaptively invoked at run-time. Third, it allows specialization of microprocessors with the use of different I-COP code or even different I-COP implementations. Fourth, it makes it possible to modify and upgrade the machine simply by changing I-COP code without changing the hardware. We believe an I-COP can potentially be a valuable addition to the microarchitect's toolbox.

In evaluating the feasibility of the I-COP concept, [3] showed that an I-COP programmed to implement trace construction and trace optimizations achieves good performance. The longer latency (as compared to hardwired logic) that the programmable I-COP takes to perform the code modifications had little impact on performance because the I-COP is located at the back-end of the core processor and because of the frequent reuse of the modified code. The prototype I-COP proposed consists of two VLIWs each with four general function units. Such an I-COP implementation can require a significant amount of chip area.

This paper proposes a novel and much more efficient I-COP implementation using a reconfigurable architecture called PipeRench [4]. In such an implementation, I-COP programs are actually *configuration bits* that are downloaded to the reconfigurable fabric at run-time. After configuration, the fabric becomes a hardware design that implements the desired computation. What distinguishes PipeRench from other reconfigurable fabrics is that it supports very fast reconfiguration as well as a virtualization technique called *pipeline reconfiguration*, which allows a large logical design to be implemented on a small piece of hardware through rapid configuration of that hardware. This virtualization enables smaller I-COP implementations, and also allows complex I-COP programs to be written without the concern that they may not fit within the size of the reconfigurable fabric.

It was shown in [4] that the PipeRench reconfigurable fabric provides significant performance benefits for an application that exhibits one or more of the following features:

1. It operates on bit-widths that are different from a processor's basic word size.
2. Its data dependencies allow multiple function units to operate in parallel.
3. It is composed of a series of basic operations that can be combined into one specialized operation.
4. It can be pipelined.
5. Constant propagation can be performed, reducing the complexity of the operations.
6. The input values are reused many times within the computation.

The results in [3] suggest that the potential I-COP applications exhibit many of these features. For example, the data bit-widths in the I-COP applications are odd and varied. There is also abundant parallelism in these I-COP programs, thus allowing multiple functional units to operate in parallel. In addition, large portions of these programs are composed of basic operations that can be combined into specialized operations. In this paper, we show how hardware code modifications can be mapped to the PipeRench pipelined computation model and that the PipeRench I-COP achieves good performance. Furthermore, we demonstrate that a PipeRench I-COP can be implemented at very reasonable hardware cost, and in so doing, further validate the usefulness of the I-COP concept.

The rest of this paper is organized as follows. Section 2 familiarizes the reader with the I-COP concept and the PipeRench reconfigurable architecture. Section 3 describes our PipeRench I-COP design and how I-COP applications are implemented in this design. Section 4 presents the results of our exploration of the PipeRench design space as well as die-area estimates of selected designs. Section 5 concludes this paper.

2 Background

2.1 Instruction Path Coprocessors

An I-COP is a programmable coprocessor that operates on the core processor's instructions to transform them into a new format that can be more efficiently processed by fast execution cores. These transformations can involve the ordering of instructions, the type of instructions (e.g. from the original instruction to a sequence of simpler instructions) and even the instruction set (e.g. from the original ISA to a new ISA tailored to the microarchitecture).

2.1.1 Interface With Core Processor

The I-COP is located on the same chip as the core processor and runs concurrently with the core processor. In order not to negatively impact the core processor's cycle time, it is situated at the core processor's back-end and interacts primarily with the core processor's completion/retirement stage. The I-COP requires minimal explicit control by the core processor and rarely stalls the core processor. Figure 1 shows the interface between the I-COP and the core processor.

An I-COP should be able to access non-architected entities of the core processor, such as instruction and data caches, trace cache, branch and value predictor tables etc. Where such accesses are allowed, careful considerations are made to ensure that they do not affect the core processor's critical timing paths.

In order for the I-COP to intelligently invoke the appropriate I-COP code based on application characteristics, the core processor has built-in monitors to track its currently executing application's behavior. The I-COP can either poll these monitors or the I-COP can be interrupt-driven. In the latter case, when the monitors exceed or dip below threshold levels, they interrupt the I-COP and cause it to vector to specific I-COP routines.

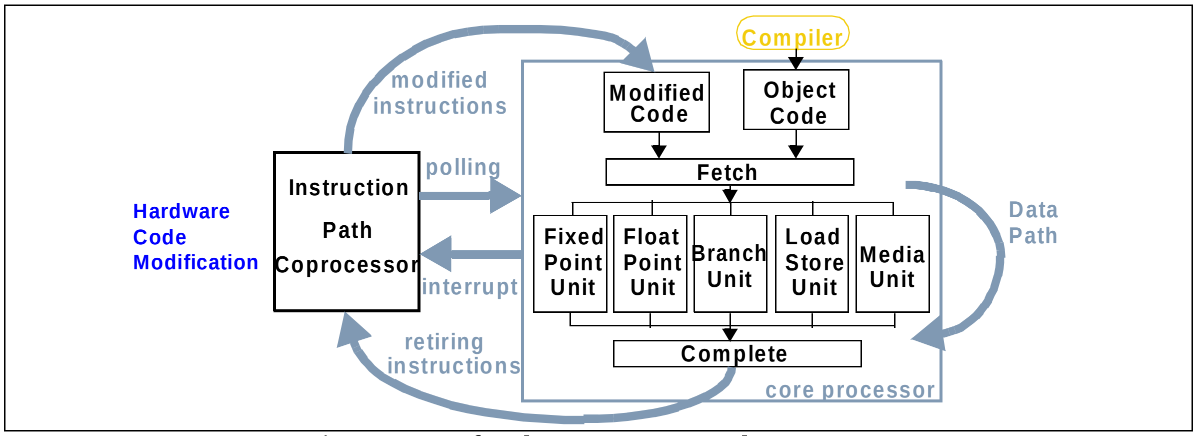


Figure 1. Interface between I-COP and core processor.

2.1.2 Initial Implementation

The initial I-COP implementation [3] was based on conventional CPU design and comprised of one or more VLIW engines (called slices) operating in parallel. For the I-COP applications studied, two VLIW slices with four general functional units each represented a good cost-performance trade-off. The VLIW organization was chosen to minimize hardware complexity, since I-COP programs are relatively small and can easily be statically scheduled. All the slices share a common data memory. Since an I-COP replaces hardwired designs with a programmable engine, slow-down can be expected. To ensure adequate performance, parallelisms in I-COP programs were exploited; instruction-level parallelism was exploited within a VLIW slice and task-level parallelism was exploited across VLIW slices.

The instruction set for the I-COP VLIW slices consisted of 22 instructions. The core of the instruction set was a simple integer-based load/store architecture. In addition, ten specialized instructions were provided to facilitate writing efficient I-COP programs. The most important of these are powerful (and complex) pattern matching instructions to enable regular expression recognition to be performed quickly. Predication support as well as branch delay slots were also provided to eliminate the need for branch prediction. More details about this implementation can be found in [3].

The experimental results showed that this initial I-COP implementation achieved good performance for the I-COP applications studied. However, the drawback is that it requires a significant amount of hardware and can potentially consume sizable chip area.

2.2 PipeRench

PipeRench [4] is a reconfigurable fabric that supports the computational model shown in Figure 2. In this model, a computation on a data stream is expressed as a linearly interconnected set of $\boldsymbol{S}$ pipeline stages, where every stage is a function of the registered output of the previous stage and the registered output of the current stage. Many media and embedded computational kernels can be mapped to this model with many pipeline stages, which allows for high clock speeds and high throughputs. The small amount of feedback allows for efficient implementation. Many instruction transformation techniques can also be mapped to this model. In most instruction transformations, the particular transformations initiated by any instruction only affect subsequent instructions, which fits the limited feedback model.

Assuming that new inputs arrive every cycle, an implementation of this pipeline will require $\boldsymbol{S}$ stages. In PipeRench, the technique of pipeline reconfiguration [5] is used to support the cases when the input stream has an arrival rate, or throughput $\boldsymbol{T}$, which is less than one every cycle. In this case, $\boldsymbol{S}$ physical stages cannot be kept busy. Alternatively the technique is also useful when the cost of $\boldsymbol{S}$ stages is prohibitive. The technique is illustrated in Figure 3, where the number of stages in the application, $\boldsymbol{S}$, is five and the number of physical pipeline stages $\boldsymbol{P}$, is three. As the figure shows, the configuration of stages happens concurrently with the execution of other stages.

Using pipeline reconfiguration, the relationship between S, P and T is given by $T = max\left(\frac{P}{S}, 1\right)$. If $P \geq S$ and the input streams consist of a set of $\boldsymbol{N}$ words, the entire

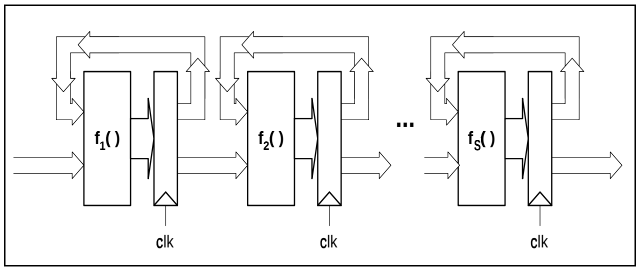


Figure 2. PipeRench computation model.

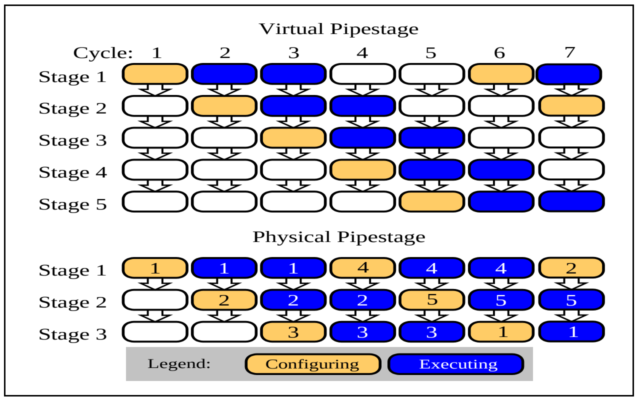


Figure 3. PipeRench pipeline reconfiguration.

computation will have a latency of $N+S$ cycles. If $P<S$, then virtualization is necessary, and the computation will take $S\left\lceil\frac{N}{P}\right\rceil+P$ cycles to complete. In the rest of this paper, we use the term *virtual stripes* to refer to the pipeline stages required by the application and the term *physical stripes* to refer to the physical pipeline stages available. As any virtual stripe can be mapped to any physical stripe, all the physical stripes must have the same functionality and interconnect.

The current architecture of PipeRench is optimized by evaluating a set of media-centric applications and is illustrated in Figure 4. Each physical stripe consists of sixteen ALUs (labelled PEs), which are each eight bits wide, connected with a byte-wise crossbar and an elaborate set of shift registers. The ALUs are capable of all possible bitwise functions on two operands, as well as addition, subtraction and multiplexing. Each of the ALUs also contains an eight entry register file which is pipelined to provide pipeline interconnect to downstream pipeline stages. State values (those feeding back in Figure 2) can only be stored in one specific register in the register file. An input and output bus moves operands on and off the execution fabric.

A physical design of this architecture has been completed in 0.35 micron and 0.18 micron process technologies. In 0.18 micron, a single physical stripe consumes 1.03 sq mm of silicon area, and operates at over 200MHz. Some small additional chip area is required for storage of configuration information and state that needs to be held during virtualization. This is a very conservative design, with static CMOS circuits and fabricated in an ASIC process. We expect considerable headroom in improving both die area and clock speed.

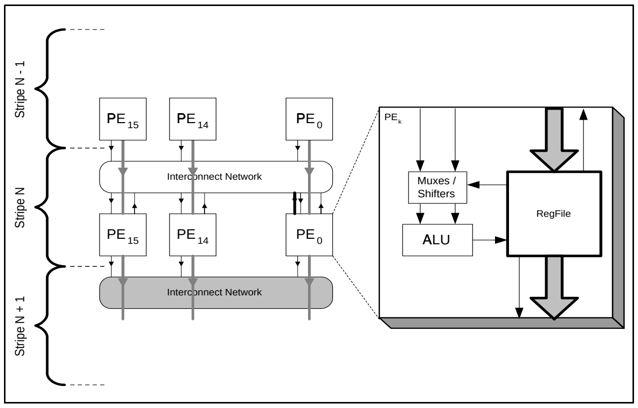


Figure 4. PipeRench Architecture.

PipeRench applications are written in the *Dataflow Intermediate Language* (DIL), which is a single-assignment language with C operators and a type system that allows the bit-width of variables to be specified. The DIL compiler [6] converts the source into a dataflow graph, decomposes this graph into the native operators of the architecture and places and routes the operators on the PipeRench fabric. The output of the compiler is a set of *configuration bits* (actually divided into a number of subsets, one subset per virtual stripe) that are used to configure the physical stripes at run-time.

2.3 PipeRench I-COP Advantages

In addition to being area-efficient, which we will demonstrate in Section 4.3, the PipeRench I-COP implementation also offers a number of other advantages. The PipeRench architecture allows the designer to easily trade off the size of the reconfigurable fabric with other parts of the microarchitecture to optimize the overall design. Since the DIL code for the I-COP applications do not even need to be modified, changes to the number of physical stripes can be made very late in the design cycle. Moreover, when the same microarchitecture is implemented in the next process generation, the designer has the option of increasing the number of physical stripes available to increase performance. Since physical stripes in the reconfigurable fabric are identical, this can be accomplished with minimum redesign. The designer can also choose to upgrade the resident I-COP programs to further enhance performance. All in all, the PipeRench I-COP allows the designer to improve the performance of the core processor with minimal logic and circuit redesign.

The PipeRench I-COP also retains the other I-COP advantages like allowing complex hardware code modification techniques to be implemented as I-COP code and allowing many hardware code modifications to be implemented using the same engine, each of which can be selectively and adaptively invoked at run-time. In addition, the PipeRench I-COP makes it especially easy to specialize the core processor by varying the size of the reconfigurable fabric to achieve different performance goals and support different levels of complexity in the I-COP programs.

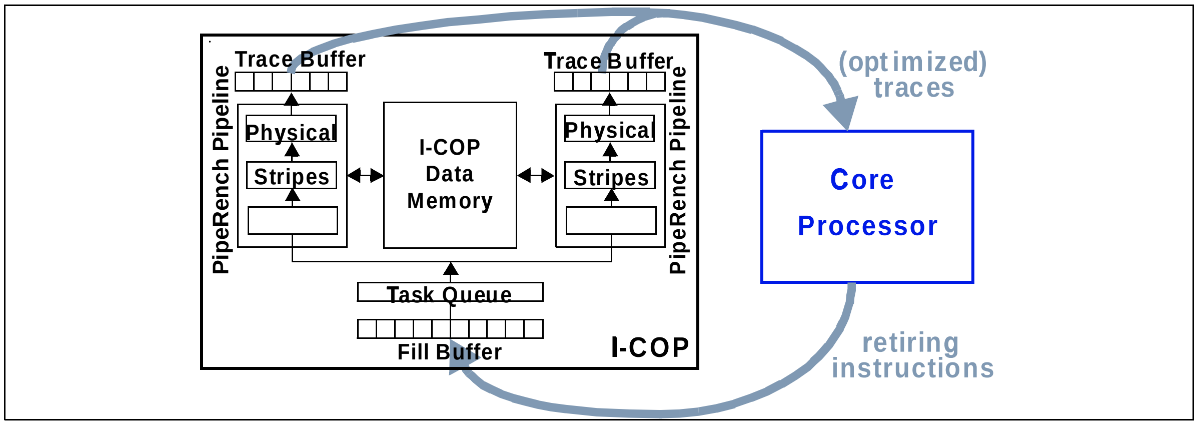


Figure 5. PipeRench I-COP implementation.

3 PipeRench I-COP Implementation

In this section, we describe the design of the PipeRench I-COP. In order to assess its performance and die area requirements, we study how it can implement four specific hardware code modification techniques: namely trace construction, register move trace optimization, stride data prefetch trace optimization, and linked data structure prefetch trace optimization. These are the same techniques implemented in the earlier study [3] and therefore allows us to compare the PipeRench and VLIW implementations in terms of performance and area efficiency. In Section 3.2, we describe how these four code modification techniques are implemented on the PipeRench I-COP as DIL programs. We anticipate that in the future, many other advanced code modifications will be mapped to the PipeRench I-COP computation model.

3.1 PipeRench I-COP Design

The PipeRench I-COP implementation comprises of one or more PipeRench pipelines (each consisting of one or more physical stripes) operating in parallel. A PipeRench pipeline constructs and optimizes traces by treating the retiring instructions from the core processor as *streaming input data*. The outputs of each PipeRench pipeline are written to its local *trace buffer*, which acts as temporary storage to hold a trace as it is being constructed. When a trace is fully constructed, it is copied from the trace buffer to the trace cache. A PipeRench I-COP implementation with two pipelines is shown in Figure 5. The fill buffer collects the retiring instructions from the core processor, and the task queue distributes them to the PipeRench pipelines. When the fill buffer is full, instructions are dropped at basic block boundaries. If a PipeRench pipeline has sufficient physical stripes to match the number of virtual stripes required by the I-COP applications, it accepts one fill buffer instruction per cycle as input and writes one instruction to the trace buffer per cycle as output. Otherwise, the physical stripes are time multiplexed and the throughput of trace processing will be less than one instruction per cycle. In Section 4, we evaluate the performance impact of varying the number of Pipe-Rench pipelines and the number of physical stripes per pipeline.

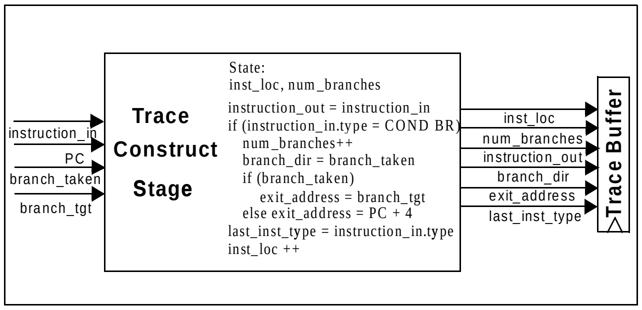
3.2 Implementing Code Modifications Using PipeRench

To implement code modifications on the PipeRench I-COP, they are first mapped to the PipeRench computation model described in Section 2.2. They are then written in the DIL language and compiled by the DIL compiler to produce the configuration bits used to configure the physical stripes of the PipeRench I-COP at run-time.

3.2.1 Trace Construction

The trace cache [2][7][8] stores frequently executed sequences of instructions in physically contiguous storage locations, thus allowing high bandwidth instruction fetch without multiple cache ports nor instruction alignment logic. This dynamic regrouping of instructions is performed by a hardware structure called the *fill unit* which is located at the back-end of the machine. A trace comprises not only of regrouped instructions but also the outcomes of the branches in the trace, the exit addresses of the trace (to facilitate partial matching [7]) and the type of the last instruction in the trace.

In our I-COP implementation, logic associated with the fill buffer examines its first 16 entries and determines the end of a new trace. It then copies those instructions from the fill buffer to the I-COP memory and inserts a task into the task queue. Whenever a PipeRench pipeline is free, it picks up a task from the front of the task queue and treating the fill buffer instructions in I-COP memory as streaming input, processes one instruction in the trace at a time and outputs the processed instructions to the trace buffer (see Figure 6). In the case of branch instructions, the PipeRench pipeline also outputs the branch outcome and



```
State:
inst_loc, num_branches
instruction_out = instruction_in
if (instruction_in.type = COND BR)
   num_branches++
   branch_dir = branch_taken
   if (branch_taken)
       exit_address = branch_tgt
   else exit_address = PC + 4
last_inst_type = instruction_in.type
inst_loc ++
```

Figure 6. Trace construction using PipeRench.

exit address associated with that branch to the trace buffer. When the trace is fully constructed, the trace cache inside the core processor is read to check if there is an existing trace with the same starting PC. The new trace is written to the trace cache as long as it is not a subset of an existing trace. The PipeRench pipeline is then ready to pick up a new task. Based on the reconfigurable fabric's resource constraints (16 8-bit ALUs per stripe), the DIL compiler maps the trace construction logic to 11 virtual stripes. More details on the PipeRench implementation of trace construction and the other code modification techniques can be found in [26].

3.2.2 Register Move Trace Optimization

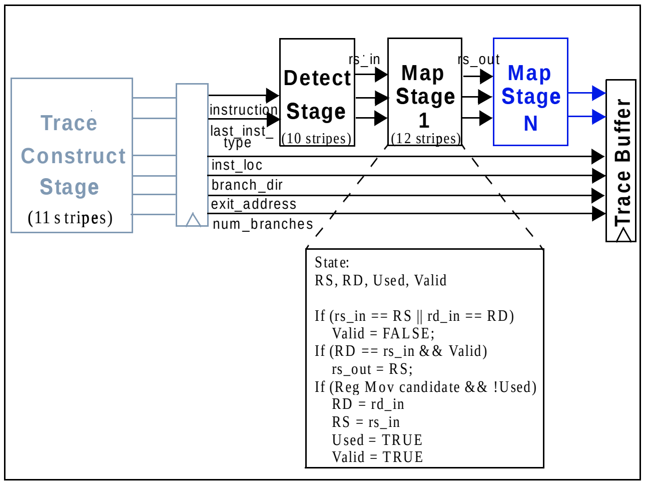
Beyond basic trace construction, the I-COP can perform optimizations on traces to achieve additional performance. Recently, there have been proposals for various trace optimizations [9][10]. The register move optimization [9] is one such example. In this optimization, instructions within a trace which move a value from one register to another register without modifying it are marked as *explicit move* instructions by the fill unit. Examples of such instructions are:

ADD Ra <- Rb + 0

SHIFT Ra <- Rb << 0

Instead of using execution resources to execute these instructions, their output registers are renamed to the same physical registers (or operand tags depending on the register renaming scheme used) as their input registers. Aside from saving execution resources, this also enables dependent instructions to execute earlier. The register renaming logic is modified to handle such explicit moves. A slight complication is that the input registers of dependent instructions within the same trace have to be substituted with the input register of the *explicit move* instruction.

In our PipeRench I-COP implementation shown in Figure 7, the register move optimization is performed after trace construction and before the trace is written from the trace buffer to the trace cache. Because this optimization is fairly expensive, it is not applied the first time a trace is



```
State:
RS, RD, Used, Valid

If (rs_in == RS || rd_in == RD)
   Valid = FALSE;
If (RD == rs_in && Valid)
   rs_out = RS;
If (Reg Mov candidate && !Used)
   RD = rd_in
   RS = rs_in
   Used = TRUE
   Valid = TRUE
```

Figure 7. Register move optimization using PipeRench.

written into the trace cache. It is only applied to a trace that is found to be already in the trace cache and has been accessed ***x*** number of times. We found ***x*** = *5* to be a good choice. Also, a trace is only optimized if it contains more than one conditional branch, since we assume the compiler already performs this optimization within a basic block. The input to the optimization is a stream of instructions from the result of trace construction and the output is a stream of optimized instructions that are written to the trace buffer. For every input instruction, two operations have to be performed. First, it must be determined if this instruction is a candidate for the optimization and if so, its type should be changed to that of an *explicit move*. Second, one or both of its source operand specifiers (i.e. register numbers) must be modified if that operand is dependent on an earlier register move candidate in the trace.

The first operation is essentially combinational logic and is performed by the Detect Stage shown in Figure 7. The DIL compiler produces a design of this stage that requires 10 virtual stripes.

The second operation is accomplished by keeping a set of mappings, labelled as the Mapping Stages in Figure 7. Each stage stores four values:

* a **valid** flag
* a 5-bit value **RD** which represents a register that is being mapped
* another 5-bit value **RS** which represents the register to which **RD** gets mapped
* a single bit, called the **used** flag, which is set if this stage has a valid mapping or ever had a valid mapping

At the beginning of each new trace, all of the stages are set to **invalid** and **unused**. When an instruction enters a stage, if the stage is **valid** and if a source register of the incoming instruction (rs_in) matches **RD**, then that source register will be renamed (rs_out) to **RS**. If the incoming instruction is a register move candidate and the stage is **unused**, and if this instruction's mapping has not yet been stored, then the stage will be marked as **used** and **valid**. The source and destination of the instruction (rs_in and rd_in) will be stored in **RS** and **RD** respectively. A one bit flag will be sent to downstream stages indicating that the mapping for this instruction has already been stored.

If an instruction reaches a stage in which the destination of the instruction (rd_in) matches either **RD** or **RS**, the stage will be set to **invalid**. However, the stage will remain marked as **used**, since it previously had a valid mapping in it. This prevents future register move candidates from storing their mappings ahead (in stage order) of an already stored mapping and ensures that older mappings in the trace always appear earlier in stage order. This in turn ensures that when a new register move candidate stores its mapping, its source register will have already been correctly renamed. Each instruction only needs to pass through the pipeline just once, thus enabling a throughput of one instruction per cycle.

For a simple example of how this design works (it can also handle all the complex cases), consider the following example of a trace with just three instructions:

ADD r2 <- r1 + 0 *(1)*

ADD r4 <- r3 + r2 *(2)*

ADD r2 <- r10 + r11 *(3)*

Instruction *1* is eligible for the optimization and will create the mapping (**RD** = 2, **RS** = 1). Instruction *2* is not eligible for the optimization but one of its source operands matches the stored mapping (**RD** = 2, **RS** = 1) and so the instruction is transformed to ADD r4 <- r3 + r1. Instruction *3* is also not eligible for the optimization and since its destination matches the stored mapping (**RD** = 2, **RS** = 1), the mapping is invalidated.

The DIL compiler produces a design that requires 12 virtual stripes for each Map Stage. In our simulations, we found that having just one set of mapping (i.e. only one *explicit move* is allowed in a trace) achieves most of the performance gains of this trace optimization. This means this trace optimization takes a total of 10 + 12 = 22 virtual stripes in addition to the 11 virtual stripes for trace construction.

3.2.3 Stride Prefetch Trace Optimization

The stride prefetching scheme we implement in the I-COP is based closely on the hardware scheme proposed by Chen and Baer [11]. The basic idea is to record the effective addresses of loads as they are executed, compute the latest stride by comparing this address to the last effective address generated by the same static load, and update a 2-bit state machine. Depending on the resulting state of the state machine, a prefetch request may be generated. All this information is recorded in a table called the Reference Prediction Table (RPT) stored in the I-COP data memory. Whenever a load is encountered during the construction of a trace, the 512-entry RPT is consulted to determine if it has a consistent stride. If so, a prefetch instruction is inserted in the trace before it is written to the trace cache. The prefetch instruction is only inserted if there is an empty slot in that particular trace cache line. This optimization is performed after trace construction and before the trace is written from the trace buffer to the trace cache. The input to the optimization is a stream of instructions from the result of trace construction and the output is a stream of the same instructions plus possibly one or more prefetch instructions. The DIL compiler produces a design that requires 14 virtual stripes.

3.2.4 LDS Prefetch Trace Optimization

Linked data structures (LDS) include linked lists, trees and graphs etc., where individual nodes are dynamically allocated from the heap and linked together through pointers to form the overall structure. The LDS prefetching we implement is based on that proposed by Roth et al. [12]. In this scheme, the goal is to correlate pairs of loads like the following, where the result of the first load is used as the base address for the second load:

LOAD r2<- M[0(r1)]

LOAD r3<- M[8(r2)]

After the correlation is established, whenever the first load is executed, a prefetch can be issued for the second load to hide the potential cache miss latency. Correlations are established by actual values rather than by symbolic means, with the help of two tables stored in the I-COP data memory: the 256-entry Potential Producer Window (PPW) and the 512-entry Correlation Table (CT). Whenever a load is encountered during the construction of a trace, it updates the PPW and CT. It also searches the CT and if it is found to be a producer, a prefetch instruction is inserted as part of the trace before the trace is written to the trace cache. The DIL compiler produces a design that requires 9 virtual stripes.

3.2.5 Comparison With VLIW-based I-COP

Table 1 compares the number of virtual stripes required by the PipeRench I-COP programs to the number of operations and cycles needed by the VLIW-based I-COP for the same programs. For the PipeRench I-COP, the number of cycles required to execute the program depends on the number of physical stripes available and is governed by the equations in Section 2.2. In Section 4.2, we study the performance impact of varying the number of physical stripes.

I-COP Application	Virtual Stripes	VLIW ops	VLIW cycles
Trace construction	11	50	18
Register move trace optimization	22	423	106
Stride prefetch trace optimization	14	130	33
LDS prefetch trace optimization	9	86	22

Table 1: Comparison between PipeRench and VLIW-based I-COP implementations.

4 Experimental Results

4.1 Simulation Methodology

Our performance simulator is built around Digital's ATOM tool [13] and uses the Alpha ISA [14]. Although it is trace-driven, it models the resource contention (but not cache effects) due to instructions on the mispredicted path.

The organization of the core processor is as follows. The trace cache contains 128KB of instructions (2048 lines of 16 instructions) and is 4-way set associative. Partial matching is implemented. The branch predictor is as described in [7]. It is an adaptation of the *gshare* predictor, and makes 3 predictions per cycle. We assume a perfect return address stack which is used to predict subroutine returns. The L1 instruction cache is 16KB and direct-mapped, with a 14 cycle miss latency. Because of the low instruction cache miss rates, an L2 instruction cache is not modeled.

Functional Units	Units	Latency
Simple Integer	8	1
Complex Integer	4	4
Load/Store	4	2/1
Branch	4	1
Floating-Point Add/Multiply	4	3
Floating-Point Divide	4	11(sp), 15(dp)

Table 2: Core processor execution resources.

The front-end pipeline of the core processor, from fetch to dispatch, is four stages deep. Instructions are dispatched to a 512 entry centralized instruction window and are allowed to issue out-of-order. Perfect memory disambiguation is assumed. The functional unit mix and their execution latencies are shown in Table 2; all functional units are fully pipelined. The L1 data cache is 16KB and direct-mapped and the miss latency to the L2 data cache (assumed off chip) is 14 cycles. The L2 data cache is 256KB and 2-way set associative, with a miss latency to main memory of 75 cycles. In our data prefetching experiments, prefetched data are brought into a 64 entry fully-associative prefetch buffer. The PipeRench I-COP model is integrated with the core processor's simulator and is simulated in detail at the machine cycle level.

Seven SPECint95 benchmarks [15] and three pointer-intensive Olden benchmarks [16] are used. Their input sets and dynamic instruction counts are shown in Table 3. The benchmarks are compiled using the default optimization flags of the SPEC distribution and are run to completion.

Benchmark	Input Set	Inst Count
compress	10000 e 2231	54M
ijpeg	tinyrose.ppm	89M
m88ksim	dhry2tiny.lit	99M
go	5 9	78M
gcc	-O genoutput.i	106M
li	queens 6	56M
perl	trainscrabbl	47M
health	5 levels, 500 iters	176M
perimeter	4K x 4K image	43M
treeadd	1024K nodes	98M

Table 3: Benchmark characteristics

4.2 Performance Data

In this section, we show the performance of the core processor under different PipeRench I-COP organizations. In particular, we vary the number of physical stripes per PipeRench pipeline as well as the number of PipeRench pipelines. Since the reconfigurable fabric may have to be clocked at a slower clock speed than the core processor, we show two sets of results. The first assumes the reconfigurable fabric is clocked at the same speed as the core processor while the second assumes it is clocked at half the speed.

While evaluating the different design points of the design space, it is helpful to bear in mind that each physical stripe in a 0.18 micron process occupies 1.03 sq mm of silicon area (approximately 1/300th the area of a 300 sq mm die used in current high-end microprocessors).

4.2.1 Trace Construction

Figure 8 shows the performance of the core processor with its I-COP implemented in different PipeRench organizations for trace construction. In particular, we vary the number of PipeRench pipelines as well as the number of physical stripes per pipeline. The upper graph assumes that the PipeRench I-COP runs at the same speed as the core processor while the lower graph assumes that it runs at half the speed. In both graphs, the *y* axis shows the harmonic mean of the IPCs of the seven SPECint95 benchmarks and the *x* axis represents the total number of physical stripes. The sets of data points on each graph represent varying the number of PipeRench pipelines. The number of physical stripes per pipeline can be derived by dividing the total number of physical stripes by the number of pipelines. For

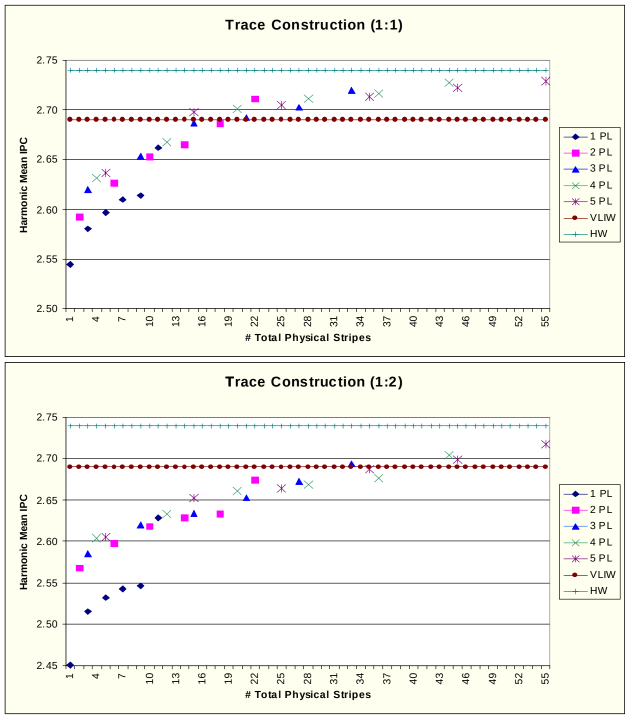


Figure 8. Trace construction performance.

example, the data point [2 pipelines (PL), 14 total physical stripes] implies there are seven physical stripes per PipeRench pipe line. For comparison, the performance of the VLIW-based I-COP (labeled VLIW) as well as a hardwired trace cache fill unit (labeled HW) are also shown.

The throughput at which the I-COP constructs traces is directly proportional to the total number of physical stripes available, while the latency of trace construction is inversely proportional to it. When the throughput of trace construction is reduced, more instructions are dropped from the fill buffer since the I-COP is not able to keep up with the rate at which instructions are retired by the core processor. However, because of the frequent reuse of previously constructed traces, these dropped instructions do not adversely affect overall performance. Moreover, because the I-COP is located at the back end of the core processor, longer latencies in trace construction also do not seriously affect performance. Therefore, there is diminishing returns in performance as the total number of physical stripes is increased.

Given a fixed total number of physical stripes (and throughput), performance varies slightly depending on the exact PipeRench organization. This is due to several factors. First, the latency of trace construction has a ceiling function (see Section 2.2) that produces discontinuities. In particular, 11 physical stripes per pipeline results in particularly good performance because the average trace length is approximately 11 instructions. Second, the number of PipeRench pipelines affects trace selection because instructions are dropped from the fill buffer at a different timing. When there are more pipelines, there will be a longer series of contiguous traces followed by a larger number of dropped instructions. When there are fewer pipelines, there will be a shorter series of contiguous traces followed by a smaller number of dropped instructions. The former situation is more desirable than the latter, so in general, for a given number of total physical stripes, it is better to have more pipelines and fewer physical stripes per pipeline.

For a particular performance level (i.e. fixed value on *y* axis), the most desirable PipeRench I-COP organization is the one with the least total number of physical stripes. For example, if we want to match the performance of the VLIW I-COP implementation, the design point [3 pipelines, 15 total physical stripes] is the best organization when the clock speed of the PipeRench I-COP matches the clock speed of the core processor. When the clock speed is half that of the core processor, the design point [3 pipelines, 33 total physical stripes] is the best organization.

4.2.2 Register Move Trace Optimization

Figure 9 shows the performance of different PipeRench organizations when the register move optimization is applied in addition to basic trace construction.

The graphs in Figure 9 are organized in a similar fashion to those in Figure 8. Because the PipeRench I-COP is efficient in implementing this optimization (22 virtual stripes in addition to 11 virtual stripes for basic trace construction; in contrast, the VLIW requires 423 instructions in addition to 50 instructions for basic trace construction), fewer total physical stripes are needed to match the VLIW implementation. When the PipeRench I-COP runs at the same speed as the core processor, the [4 pipelines, 12 total physical stripes] organization matches the performance of the VLIW I-COP. When it runs at half the speed, the [5 pipelines, 25 total physical stripes] organization accomplishes the same goal.

From the results, we also observe that when compared to basic trace construction, given the same PipeRench I-COP organization, applying this optimization improves performance. For example, the harmonic mean IPC of the I-COP organization [3 pipelines, 15 total physical stripes] (assuming I-COP and core processor run at the same clock speed) increases from 2.69 to 2.72. Although these performance improvements are modest, no additional I-COP hardware was required; only the I-COP code, i.e. the PipeRench *configuration bits*, are changed.

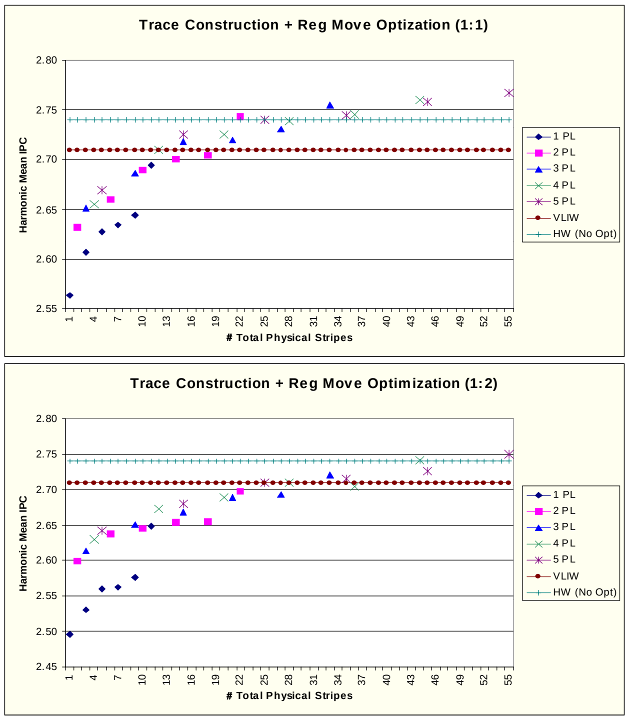


Figure 9. Register move optimization performance.

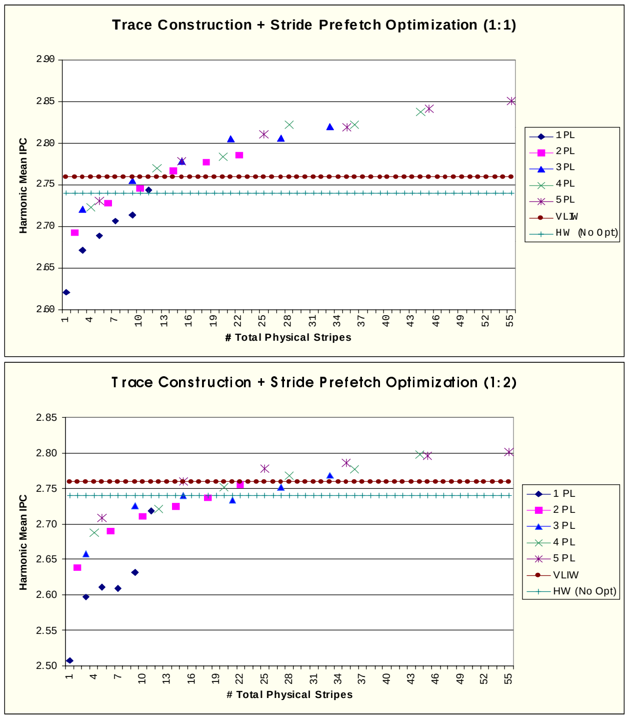


Figure 10. Stride prefetch performance.

4.2.3 Stride Prefetch Trace Optimization

Figure 10 shows the performance of different PipeRench organizations when the stride data prefetch optimization is applied in addition to basic trace construction. This optimization is applied to all traces. The results are clearly superior to those in Figure 8, demonstrating the advantage of an I-COP in being able to improve core processor performance by modifying I-COP code and without changing the I-COP hardware. The PipeRench I-COP is efficient in implementing this optimization, requiring only a [3 pipelines, 9 total physical stripes] organization to match the VLIW I-COP when it is running at the same clock speed as the core processor. When it is running at half the speed, a [5 pipelines, 15 total physical stripes] organization is required. Note also that these I-COP organizations also handily exceed the performance of the hardwired trace cache fill unit performing trace construction with no trace optimization (labeled HW (No Opt) in Figure 10).

4.2.4 LDS Prefetch Trace Optimization

Figure 11 shows the performance of a PipeRench I-COP running at the same clock speed as the core processor when the LDS data prefetch optimization is applied. Because the IPC performance of the *health* benchmark is an order of magnitude lower than those of the other two Olden benchmarks, we avoid using the harmonic mean of their IPCs. Instead, the performance of each benchmark is shown separately. The I-COP organization shown is the same one that matches the performance of the VLIW I-COP for the stride data prefetch optimization, i.e. 3 pipelines, 9 total physical stripes. We observe that a small PipeRench I-COP is able to match the performance of the VLIW I-COP. The performance of this PipeRench I-COP also exceeds that of the hardwired trace cache fill unit with no trace optimization by a considerable margin.

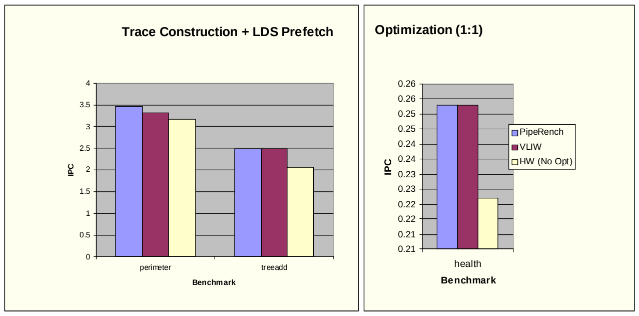


Figure 11. LDS prefetch performance (1:1 clock speed).

Figure 12 is similar to Figure 11 except that the results shown are for an I-COP that runs at half the clock speed of the core processor. The I-COP organization shown is the same one that matches the performance of the VLIW I-COP for the stride data prefetch optimization, i.e. 5 pipelines, 15 total physical stripes. Again, we observe that a small PipeRench I-COP is able to match the performance of the VLIW I-COP.

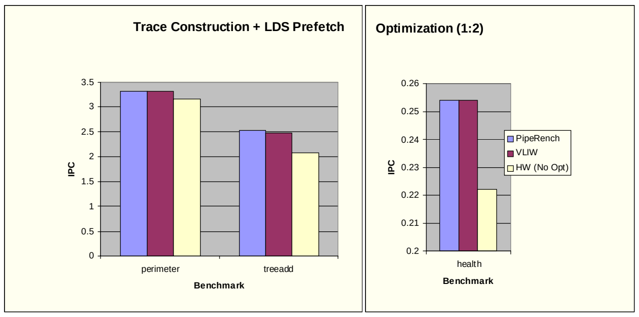


Figure 12. LDS prefetch performance (1:2 clock speed).

4.2.5 Estimated Area of PipeRench I-COP

To match or exceed the VLIW I-COP performance for trace construction and all three trace optimizations, and assuming that the PipeRench I-COP is only able to run at half the speed of the core processor, the estimated die area of the PipeRench I-COP (33 total physical stripes) fabricated in a 0.18 micron process is 33 x 1.03 = 34 sq mm. To put this in perspective, Table 4 shows the estimated die

Component		% of die	Process (u)	Area (mm^2)	Area scaled for 0.18 u
IBM G6 [17]	FPU	7.1	0.22	15.3	10.2
Transmeta 3120 [18]	FPU	12.3	0.22	9.5	6.3
UltraSparc-2i [19]	FPU	12.0	0.29	18.0	6.9
AMD K6 [20]	FPU	14.3	0.35	23.1	6.1
NEC MP98 [21]	64KB cache	8.0	0.15	9.1	13.1
NEC Cache SRAM [22]	512K cache	100.0	0.25	132.0	68.4

Table 4: Die areas of microarchitecture structures.

areas of other microarchitecture structures in the same process. The PipeRench I-COP is roughly equivalent in area to 256KB of fast SRAM, or about 11% of the die area of a current high-end microprocessor. If the I-COP is able to run at the same speed as the core processor, the die area required drops to approximately 15 x 1.03 = 15 sq mm, or roughly equivalent to 128KB of fast SRAM, or about 5% of the die area.

As noted in Section 2.2, these area estimates are likely to be conservative due to the conservative circuit design and fabrication process assumed. Also, the DIL compiler achieves relatively low utilization of PipeRench resources because of its fast and greedy approach to placement and routing, as illustrated in Table 5. The percentage resource utilization numbers are obtained by dividing the total number of native PipeRench operations in the application by the total number of ALUs available in a design with the resultant number of physical stripes. Significant performance improvements can easily be obtained by optimizing the PipeRench architecture, circuit design and compiler for I-COP applications.

I-COP Application	Resources Utilized Per Stripe
Trace construction	54%
Register move trace optimization	55%
Stride prefetch trace optimization	70%
LDS prefetch trace optimization	56%

Table 5: Utilization of PipeRench fabric resources.

If one is willing to trade off a little performance (2.6% lower for trace construction, 1.8% for the register move trace optimization, 1.1% lower for stride prefetch trace optimization), and assuming the PipeRench I-COP is only able to run at half the speed of the core processor, one can implement the [3 pipelines, 9 total physical stripes] I-COP in 9 x 1.03 = 9.27 sq mm, which is roughly equivalent in area to 64KB of fast SRAM, or 3% of the die area of a current high-end microprocessor. In future fabrication processes (0.13 micron and beyond), the I-COP will occupy an even smaller fraction of the available die area.

5 Conclusions and Future Work

In this paper, we have described an efficient means of implementing an I-COP by using the PipeRench reconfigurable architecture. We also show how hardware code modifications can be mapped to the PipeRench pipelined computation model. In our experimental evaluation, we found that a PipeRench I-COP used to perform trace construction and trace optimizations for a trace cache fill unit not only achieves good performance but can be implemented in less than 11% of the area of a current high-end microprocessor. If one is willing to trade off only a little performance, this figure can be reduced to 3% or lower. We believe that this demonstrates that an I-COP can be implemented in a reasonable amount of chip area.

In addition to being area-efficient, the PipeRench I-COP implementation also allows the designer to easily trade off the size of the reconfigurable fabric with other parts of the microarchitecture to maximize overall performance. As the PipeRench configuration bits do not need to be modified, this trade-off can be changed very late in the design cycle. The PipeRench I-COP implementation is also highly scalable. As I-COP programs become more complex or more I-COP programs need to be run concurrently, the number of physical stripes in the reconfigurable fabric can be increased with minimal design effort.

In conclusion, we believe that we have demonstrated the I-COP concept to be useful and feasible. With the need for increasingly sophisticated hardware code modification techniques, we believe that an I-COP is a potentially powerful tool in the microarchitect's arsenal. We also believe that hardware code modification techniques enabled by the I-COP can be synergistically combined with software runtime code optimization techniques to further improve the performance of future high performance microprocessors.

Our current research focuses on studying other I-COP applications like using an I-COP to perform run-time trace scheduling [23] and completion-time branch prediction in the context of a trace cache [24]. We also plan to study the interface between the I-COP and the core processor in greater detail, and in particular how the core processor can selectively and adaptively invoke the appropriate I-COP programs based on application behavior. Finally, we hope that the demonstrated feasibility of the I-COP concept will serve to stimulate further research into advanced hardware code modification techniques.

Acknowledgment

PipeRench development was primarily sponsored by DARPA, under contract DABT 63-96-C-0083. This work benefited from machines donated by Intel, and was also supported in part by ONR (N00014-97-1-0701, N00014-96-1-0928) and in part by Intel Corp.

References

[1] Linley Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," in Microprocessor Report, Vol. 9, Issue 2, February 1995.

[2] E. Rotenberg, S. Bennett and J. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," in Proc. of 29th International Symposium on Microarchitecture, 1996.

[3] Y. Chou and J. Shen, "Instruction Path Coprocessors," in Proc. of 27th International Symposium on Computer Architecture, June 2000.

[4] S. Goldstein et al., "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," in Proc. of 26th International Symposium on Computer Architecture, May 1999.

[5] H. Schmit, "Incremental Reconfiguration for Pipelined Applications," in Proc. of Workshop on FPGAs for Custom Computing Machines, April 1997.

[6] M. Budiu and S. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics," in Proc. of 7th International Symposium on Field Programmable Gate Arrays, February 1999.

[7] S. Patel, D. Friendly and Y. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism," Technical Report CSE-TR-335-97, University of Michigan, May 1997.

[8] B. Black, B. Rychlik and J. Shen, "The Block-based Trace Cache," in Proc. of 26th International Symposium on Computer Architecture, May 1999.

[9] D. Friendly, S. Patel and Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", in Proc. of 31st International Symposium on Microarchitecture, 1998.

[10] Q. Jacobson and J. Smith, "Instruction Pre-Processing in Trace Processors", in Proc. of 5th International Symposium on High Performance Computer Architecture, 1999.

[11] T. Chen and J. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors", IEEE Transactions on Computers, Vol. 44, No. 5, 1995.

[12] A. Roth and G. Sohi, "Effective Jump-Pointer Prefetching for Linked Data Structures", in Proc. of 26th International Symposium on Computer Architecture, 1999.

[13] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in Proc. of SIGPLAN Conference on Programming Language Design and Implementation, 1994.

[14] Alpha Architecture Handbook, Digital Equipment Corporation, 1992.

[15] http://www.spec.org

[16] A. Rogers, M. Carlisle, J. Reppy and L. Hendren, "Supporting Dynamic Data Structures on Distributed Memory Machines", ACM Transactions on Programming Languages and Systems, 17(2), March 1995.

[17] K. Diefendorff, "Processors Penetrate Gigahertz Territory," Microprocessor Report, Vol. 14, Archive 2, February 2000.

[18] T. Halfhill, "Transmeta Breaks x86 Low-Power Barrier," Microprocessor Report, Vol. 14, Archive 2, February 2000.

[19] "Low-Cost UltraSPARC-2i Appears," Microprocessor Report, Vol. 12, No. 1, January 26, 1998.

[20] D. Draper et al., "Circuit Techniques in a 266 MHz MMX-enabled Processor," IEEE Journal of Solid State Circuits, Vol. 32, No. 11, November 1997.

[21] P. Glaskowsky, "NEC Decants Merlot," Microprocessor Report, Vol. 14, Archive 3, March 2000.

[22] H. Nambu et al., "1.8-ns Access, 550-MHz, 4.5-Mb CMOS SRAM," Vol. 33, No. 11, IEEE Journal of Solid State Circuits, Vol. 33, No. 11, November 1998.

[23] R. Nair and M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," in Proc. of 24th International Symposium on Computer Architecture, June 1997.

[24] R. Rakvic, B. Black and J. Shen, "Completion Time Multiple Branch Prediction for Enhancing Trace Cache Performance," in Proc. of 27th International Symposium on Computer Architecture, June 2000.

[25] M. Merton et. al, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots," in Proc. of 27th International Symposium on Computer Architecture, June 2000.

[26] P. Pillai, "The Instruction Path Coprocessor Implemented on the PipeRench Fabric," CMuART Tech. Report, Carnegie Mellon Univ., 2000.