Instruction Path Coprocessors

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

Abstract

This paper presents the concept of an Instruction Path Coprocessor (I-COP), which is a programmable on-chip coprocessor, with its own mini-instruction set, that operates on the core processor's instructions to transform them into an internal format that can be more efficiently executed. It is located off the critical path of the core processor to ensure that it does not negatively impact the core processor's cycle time or pipeline depth. An I-COP is highly versatile and can be used to implement different types of instruction transformations to enhance the IPC of the core processor. We study four potential applications of the I-COP to demonstrate the feasibility of this concept and investigate the design issues of such a coprocessor. A prototype instruction set for the I-COP is presented along with an implementation framework that facilitates achieving high I-COP performance. Initial results indicate that the I-COP is able to efficiently implement the trace cache fill unit as well as the register move, stride data prefetching and linked data structure prefetching trace optimizations.

1 Introduction

1.1 Dynamic Code Modification

A key lesson learned from RISC is that high performance can be achieved with fast and efficient hardware when and if the executable code is highly tuned to the machine implementation. The existence of a large installed base of legacy code and the extremely time-consuming efforts of developing new compilers and recompiling existing code strongly impede the change of software. On the other hand, spurred by the relentless progress of chip technology, hardware design is evolving at a breakneck pace and radically new microarchitectures are being proposed and implemented. Consequently the incompatibility between executable code and machine implementations is rapidly increasing.

In order to achieve high performance, this incompatibility gap must be bridged. One way to bridge this widening gap is the introduction of advanced optimization techniques in the compiler backend in the form of machine-dependent optimizations. However this approach must ensure significant performance boost in order to overcome the inertia against recompiling existing code. An alternative is to use a runtime binary to binary translator such as FX!32 [7] or a dynamic compiler such as a Java JIT compiler to perform the optimizations transparently. However, the execution of the translator or dynamic compiler adds overhead and thus impacts overall performance. Moreover, a new translator or dynamic compiler may be needed for every new machine implementation, thus requiring the user to obtain or purchase the translator or compiler upgrade. A third alternative is to add hardware in the microarchitecture to dynamically modify the object code into another format that can be more efficiently processed by fast execution cores. We refer to this informally as *dynamic code modification* and contrast it with traditional direct execution of compiler-generated object code in Figure 1.

There are a number of examples of such dynamic code modification mechanisms. For example, predecode logic is used to generate predecode bits at I-cache refill time [1]. The predecode bits help reduce the complexity and the latency of the decode stage. The instructions appended with predecode bits can be viewed as a modified form of

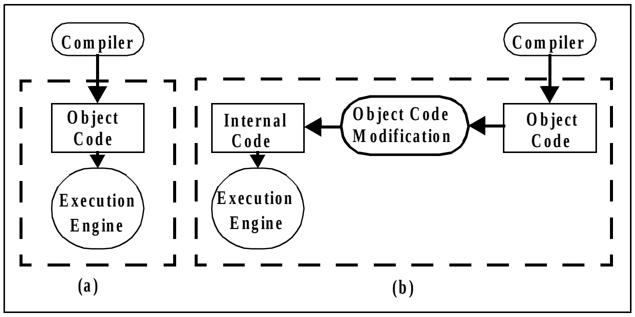


Figure 1. (a) Traditional direct execution; (b) Dynamic code modification.

the original object code. Another example is the use of sophisticated decoders that take the original instructions and translate them into another internal format. For example, the Intel P6 [2] decoders translate the x86 instructions into an internal format called uops. The uops are then executed by the fast execution core. A third example is a recently proposed idea called the *trace cache* [3][4][5]. The trace cache buffers a dynamic sequence of instructions that can span multiple basic blocks, and stores them as a trace. The fetch unit can fetch from the trace cache the entire trace in one cycle without requiring a multiported I-cache nor instruction alignment and collapsing logic..

These examples of dynamic code modifications share some common attributes. The modifications are all performed by *hardware* on the compiled object code and done at *runtime*, concurrent with code execution. The modified code can take on different formats, but all with the same goal of facilitating faster execution on more efficient hardware. Furthermore, the modified code can be buffered for repeated execution. With such buffering and repeated use, the overhead spent in dynamically modifying the object code can be effectively amortized. If the overhead does not seriously impact the critical execution path and there is significant amount of reuse, substantial performance gain can be expected.

1.2 Instruction Path Coprocessor Proposal

We believe that in the quest for higher performance, more and more complex dynamic code modification techniques will be needed in the future. In this paper, we propose the concept of the *instruction path coprocessor* (I-COP), a term we borrowed from [6], for implementing these dynamic code modifications. I-COP is an *on-chip programmable coprocessor* with its own mini-instruction set. The dynamic code modification techniques are implemented as I-COP code. In the past we have witnessed the incorporation of datapath coprocessors for operating on specialized data types, e.g. floating-point coprocessors and multimedia coprocessors. The instruction path coprocessor is analogous to the datapath coprocessors, except that it operates on the core processor's *instructions* themselves.

I-COP replaces hardwired logic implementations with a programmable engine. Using a programmable engine has the following advantages. First, complex transformations that are difficult to implement directly in hardwired logic can be implemented in I-COP code. Second, it allows many different optimizations 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 incorporation of different sets of optimizations or even different I-COP implementations. Fourth, it makes it possible to modify and upgrade the machine without changing the hardware. For example, the IPC of a machine can be increased without making any hardware changes. The down side is the potential slow down that can occur whenever hardwired logic is replaced by executing code on a programmable engine. However, in this paper, we show that this is not a serious problem.

1.3 Potential I-COP applications

To highlight the potential of the I-COP concept, we now briefly describe some possible I-COP applications.

1.3.1 Trace Construction and Optimization

The trace cache [3][4][5] stores frequently executed sequences of instructions into physically contiguous storage locations, thus allowing high bandwidth instruction fetch without significantly increasing fetch stage complexity and latency. This dynamic regrouping of instructions is performed by a hardware structure called the fill unit. The fill unit receives basic blocks of instructions as they are retired by the processor pipeline, groups them into traces, and writes these traces into the trace cache.

An I-COP can be used to implement the fill unit. It permits flexibility in the construction of traces, allowing different heuristics for trace construction and selection to be adaptively invoked depending on application characteristics. For example, the I-COP can be used to implement the algorithm proposed by Rotenberg et al. [16] to ensure that traces end in such a way that fine grained control independence can be exploited in a trace processor. Recently, there have been proposals to use the fill unit to perform optimizations on the traces [8][9]. The proposed trace optimizations include reassociation, constant propagation, instruction collapsing and instruction scheduling. The I-COP can be used to implement such optimizations. Friendly et al. [8] and Jacobson and Smith [9] showed that each of these optimizations typically only benefit a subset of applications. Instead of hardwiring all the optimizations, the I-COP can selectively execute the I-COP code to invoke only the appropriate optimizations.

1.3.2 Dynamic Instruction Formatting

Nair and Hopkins [14] proposed dynamically collecting, scheduling and caching instructions executed and retired by a scalar processor for repeated execution on a fast VLIW engine. An I-COP can be used to implement the instruction scheduling engine. Similarly, Franklin and Smotherman [15] also proposed using a fill unit to dynamically compact the stream of retired scalar instructions into VLIW instructions. An I-COP can likewise be used to implement their fill unit. An I-COP can potentially also be used to perform dynamic binary translation (e.g. of IA-32 instructions to IA-64 instructions) in hardware.

1.3.3 Concurrent Error Detection

Other than increasing performance, an I-COP can potentially be used to increase dependability. In traditional concurrent error detection, a hardware monitor runs concurrently with the core processor to detect control flow and data errors. Schuette [21] proposed using the compiler to integrate error detection code with the application code to take advantage of the core processor's idle resources and thereby remove the need for the separate hardware monitor. This requires recompilation and the resultant object code may be inefficient when different versions of the core processor are used. An I-COP can be used in place of the compiler to dynamically insert the error detection code.

In the rest of this paper, we will discuss the key I-COP design issues such as its instruction set, implementation and how it interfaces with the core processor. We will also use a subset of the above potential I-COP applications to demonstrate the feasibility of the I-COP concept. The paper is organized as follows: Section 2 describes the key I-COP design issues. Section 3 describes the four selected I-COP applications we have studied in detail and Section 4 presents their experimental results. Section 5 describes related work. Finally, Section 6 presents our conclusion and future work.

2 Architecture and Implementation

There are three key issues related to the design of an I-COP: its interface with the core processor, its instruction set design, and its efficient machine implementation.

2.1 Interface with Core Processor

The I-COP is an on-chip coprocessor. The most crucial consideration is to keep it off the critical path of the core processor to ensure that it does not negatively impact the core processor's cycle time. Therefore, the I-COP primarily interacts with the core processor's completion/retirement stage, and is expected to run concurrently alongside the core processor. It is important that the I-COP requires minimal explicit and direct control by the core processor and rarely (or preferably never) stalls the core processor.

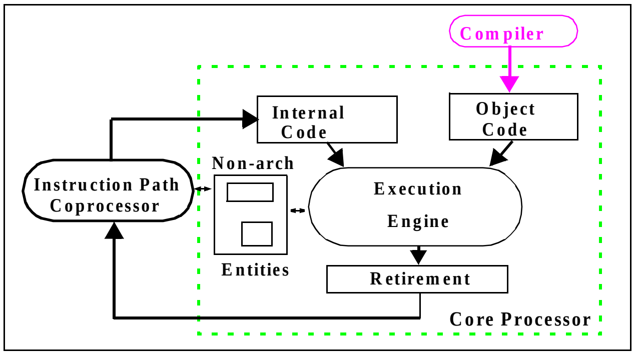


Figure 2. I-COP and core processor.

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 have to made to ensure that they do not affect the critical timing paths of the core processor.

In order for the I-COP to intelligently invoke the appropriate I-COP routines based on application program behavior, the core processor should have built-in monitors and performance counters to track its currently executing application's behavior. The I-COP can either poll these monitors and counters or the I-COP can be interrupt-driven. In the latter case, when the monitors or counters exceed or dip below threshold levels, they interrupt the I-COP and cause it to vector to specific I-COP routines. The detailed design of the interface between the I-COP and the core processor is part of our ongoing research, and is beyond the scope of this paper. We are investigating both the physical interface as well as the handshaking protocols. A high-level illustration of the proposed interface is shown in Figure 2.

2.2 I-COP Instruction Set

In this paper we define a prototype instruction set for the I-COP based on detailed analysis of potential I-COP applications and their possible hardwired implementations. We determine that other than the basic RISC type instructions, the following special instruction types are useful (Detailed descriptions of the ten specialized I-COP instructions are shown in Appendix A.):

1. Pattern matching instructions to enable regular expression recognition to be performed quickly. These instructions are useful because many of the optimizations require recognizing whether instructions fit a particular pattern.
2. Instructions to extract and insert bits from/into an integer register, and to determine if the contents of one integer register is a subset of the contents in another.

3. Instructions to access the core processor’s non-architected entities and to move data between them and the I-COP data memory.

Other than the ten special instructions, the I-COP instruction set is a simple integer-based load/store architecture with a very small instruction set size for efficient hardware implementation. Predication support and branch delay slots are provided to eliminate the need for branch prediction. The use of architected branch delay slots is acceptable since the I-COP is an internal coprocessor and user code compatibility is not an issue.

Our prototype ISA for the I-COP has an architected set of 32 integer registers (32-bit), a set of 8 predicate registers (1-bit), and a local memory called the I-COP data memory. The total instruction set consists of 22 instructions. The I-COP has a separate instruction memory, and as described in Section 2.1, is able to access non-architected structures of the core processor.

2.3 I-COP Implementation

Another design issue is the organization of the I-COP. We envision that the I-COP can be a tool for specializing the core processor and that there can be different I-COP implementations targeting different workloads. Hence, it is desirable to have a scalable machine organization that permits the implementation of different I-COPs. To minimize hardware complexity and leverage program parallelism, we believe that I-COPs should be *statically scheduled* with explicit parallelism encoded in the executable, ala VLIW or wide microcoded engines. I-COP programs are expected to be small and can be hand optimized and scheduled.

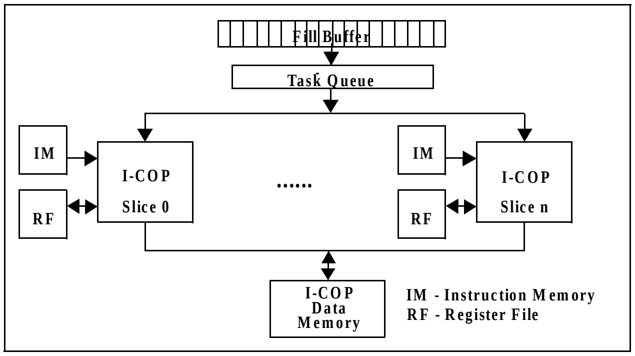


Figure 3. Scalable I-COP implementation.

A closely related issue is the performance of the I-COP. When hardwired designs are replaced by programmable engines, slow down can be expected. In order to ensure adequate performance by the I-COP, we must exploit parallelism of instruction execution. In order to exploit both instruction-level parallelism (ILP) as well as higher-level parallelism (e.g. task-level parallelism), we propose a scalable implementation like the one shown in Figure 3. In this implementation, ILP is exploited within each VLIW or wide microcoded engine we called the *I-COP slice* and higher-level parallelism is exploited across multiple such slices. All the slices share the same data memory, but to simplify the instruction memory design, each slice has its own instruction memory. For the I-COP applications we studied, an I-COP data memory of 64KB is sufficient. As more complex I-COP code is used and/or more I-COP programs need to be run concurrently, the number of slices and the instruction and data memory sizes can be scaled accordingly to ensure adequate performance. The fill buffer and task queue are used as the interface between the core processor’s retirement stage and the I-COP, and their functions are described in Section 4.2.

3 Application Examples

In order to demonstrate the feasibility and usefulness of the I-COP concept, we study four specific techniques that can be implemented using an I-COP, namely trace cache fill unit, register move trace optimization, stride data prefetch trace optimization, and linked data structure prefetch trace optimization. This section describes these four techniques and how they are implemented as I-COP programs. In Section 4, we present the performance of the I-COP implementation of these techniques.

3.1 Trace Cache Fill Unit

The trace cache fill unit receives basic blocks of instructions as they are retired by the processor pipeline, groups them into traces, and writes these traces into the trace cache. An important characteristic of the fill unit is that it is located in the backend of the machine and not in the critical path of the fetch stage. The trace cache and its fill unit are illustrated in Figure 4.

Trace construction comprises of the following steps: 1) determining where the trace starts and ends, 2) extracting the path information of the trace, 3) determining whether the trace should replace an existing trace in the trace cache, and 4) writing the trace into the trace cache.

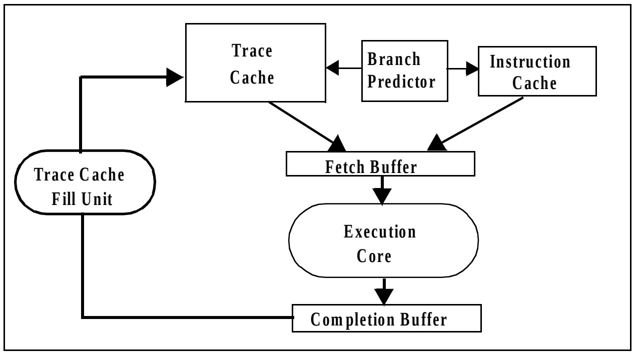


Figure 4. Trace cache mechanism.

Although the I-COP can implement many different trace construction heuristics, we use the one described by Patel et al. [4].

As instructions are retired from the reorder buffer, they are copied into a buffer called the fill buffer, which is organized as a FIFO queue. If there are insufficient entries, instructions are dropped at basic block boundaries. The fill unit examines the first 16 instructions in the fill buffer (assuming the maximum trace length is 16 instructions) and determines where the current trace being constructed should be terminated. A trace is terminated when 1) it already contains 16 instructions, or 2) it already contains three conditional branches, or 3) it contains an indirect jump, return, or trap instruction, or 4) adding the next basic block of instructions causes its length to exceed 16 instructions. The processing of the next trace can only begin when the length of the current trace has been determined.

After the length of the trace has been determined, the path information of the trace is extracted. This path information consists of the number of conditional branches in the trace and their directions, as well as the instruction type of the last instruction in the trace. Along with this path information, the four (or less) possible exit addresses of the trace are also extracted in order to facilitate partial matching [4]. The trace cache is then read to check if a trace with the same starting instruction already exists. If it doesn't, the instructions in the constructed trace, along with the path information and exit addresses are written into the trace cache. If it does, the path information of the old trace is checked against the path information of the trace being constructed. The new trace is only written into the trace cache if it is not a subset of the old trace.

In our I-COP implementation, the identification of trace end points is done by hardware associated with the fill buffer. The I-COP code receives this information as part of its input data and performs the rest of the trace construction tasks. Based on our prototype I-COP instruction set, the resultant I-COP code contains 50 instructions and is shown in Appendix B.

For comparison, we optimistically assume that an aggressive hardwired implementation takes three cycles to process a trace: one cycle to determine the trace length, one cycle to determine if the new trace should be written into the trace cache, and one cycle to write the new trace into the trace cache. The processing of a new trace can begin every cycle. In Section 4, we show that the I-COP implementation achieves almost the same performance as the hardwired implementation, and actually exceeds it when further trace optimizations are added to the I-COP code.

3.2 Register Move Trace Optimization

Beyond basic trace construction, trace optimizations can be added to the I-COP code to achieve additional performance. Recently, there have been proposals to use the fill unit to perform optimizations on the traces before writing them into the trace cache [8][9]. The register move optimization [8] is one such optimization. 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 register (or operand tag 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 a hardwired implementation of this trace optimization, comparators are needed to determine which instructions within the trace qualify as explicit move instructions. Additional comparators are also needed to substitute the input registers of dependent instructions, resulting in a significant hardware investment over the basic hardwired fill unit.

In contrast, in our I-COP implementation of this optimization, we simply insert additional lines of I-COP code to the basic fill unit code. In Section 4.3, we show that we can achieve additional performance over the basic fill unit code while using the same I-COP. The additional I-COP code comprises 423 instructions [31].

3.3 Data Prefetch Trace Optimizations

While the trace cache delivers good instruction fetch bandwidth, the performance of the core processor may still be limited by long latency operations, especially loads that miss the first-level (L1) data cache. To alleviate this problem, data prefetching techniques have been proposed to reduce the number of data cache misses. Numerous data prefetching techniques have been studied, e.g. stride prefetching [22], Markov prefetching [23] and linked data structure (LDS) prefetching [26][27]. Most of these techniques work well for only specific classes of applications. For example, stride prefetching works well for numeric applications that have striding array references, while LDS prefetching targets pointer intensive applications. It is desirable to improve data cache performance for all types of applications. Unfortunately, it is expensive to implement every data prefetching technique in hardware.

Instead, these techniques can be implemented as I-COP programs that selectively insert prefetches into traces before they are written to the trace cache. Based on an application's characteristics, the I-COP can adaptively invoke prefetch techniques that actually benefit that application. In the following two subsections, we describe how stride prefetching and LDS prefetching can be implemented as trace optimizations in the form of I-COP programs. Stride and LDS prefetching can be implemented in the compiler [24][25] but the drawbacks are that this increases the instruction cache footprint of the application and more seriously, prefetch instructions that are inserted for a certain processor generation may be ineffective for a future processor generation and become pure overhead. Also, legacy code cannot benefit from prefetching unless they are recompiled. In addition, some compiler implementations of data prefetching require accurate profile information which may not be possible or convenient to obtain.

3.3.1 Stride Prefetching

The stride prefetching scheme we implement in I-COP code is based closely on the hardware scheme proposed by Chen and Baer [22]. Their basic mechanism 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), illustrated in Figure 5.

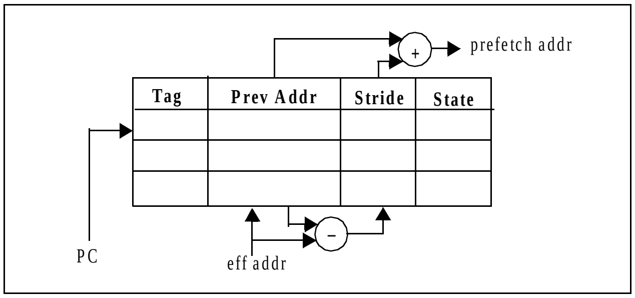


Figure 5. Reference Prediction Table.

In the I-COP implementation of this scheme, the 512 entry RPT is stored in the I-COP data memory. Whenever a load is encountered during the construction of a trace, the RPT is consulted to determine if this load is one with a consistent stride[1]. If so, a prefetch instruction is inserted in the trace before it is written to the trace cache. This prefetch instruction's base register is set to the same base register of the striding load and its offset is set to the offset of the striding load plus the stride recorded in the RPT. The prefetch instruction is only inserted in the trace if there is an empty slot in that particular trace cache line. This ensures that the prefetch instructions do not consume extra fetch bandwidth. The I-COP code for implementing stride prefetching contains 130 instructions [31].

3.3.2 Linked Data Structure Prefetching

Linked data structures (LDS), also called recursive data structures, 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.

Several prefetching techniques for LDS have been proposed, e.g. greedy prefetching, history pointer prefetching (also known as jump pointer prefetching) and data linearization [25][26][27]. The LDS prefetching we implement is based on that described by Roth et al. [26]. 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, the Potential Producer Window (PPW) and the Correlation Table (CT), illustrated in Figure 6.

Whenever a load commits, its PC and result are entered in the PPW since it is a potential producer. Its base address is also checked against the other entries in the PPW, and if there is a match, the correlation (in the form of the producer load's PC and the offset between the consumer's effective address and the consumer's base address) is recorded in the CT. Prefetches are generated when a load executes and upon consulting the CT, discovers that it is a

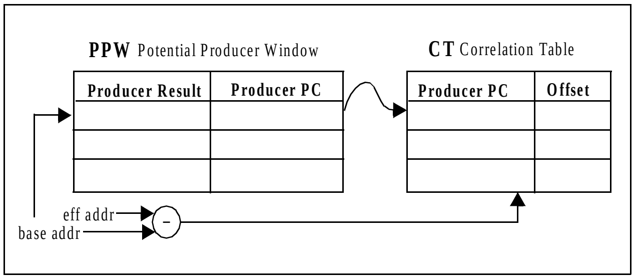


Figure 6. Establishing a correlation when consumer load executes.

[1] We assume its effective address is available to the I-COP.

producer. A prefetch is generated with an effective address equal to the sum of the load's result and the offset recorded at that CT entry.

In our adaptation for the I-COP, the 128 entry direct-mapped PPW and the 256 entry direct-mapped CT reside in the I-COP data memory. Whenever a load is encountered during the construction of a trace, it updates the PPW and CT in the same manner described above[1]. 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 into the trace cache. As in stride prefetching, the prefetch instruction is only inserted if there is an empty slot in that particular trace cache line. The prefetch instruction's base address register is set to that of the producer's result register and the offset to that of the offset recorded in the matching CT entry. The I-COP code for LDS prefetching contains 86 instructions [31].

4 Performance and Analysis

We now present the performance of our I-COP implementations of the trace cache fill unit and the three trace optimizations described in Section 3.

4.1 Experimental Methodology

Our performance simulator is built around Digital's ATOM tool [13] and uses the Alpha ISA [10]. 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 that described in [4]. It is an adaptation of the *gshare* predictor that can predict three branches per cycle. (In the future we will use the TMP completion time multiple branch predictor of [32].) We assume a perfect return address stack which is used to predict subroutine returns. The instruction cache is 16KB and direct-mapped, with a 14 cycle miss latency. A perfect L2 instruction cache is assumed.

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 1: 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. Instructions are allowed to issue out-of-order. Perfect memory disambiguation is assumed. The functional unit mix and their execution latencies are shown in Table 1; 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.

Instruction Type	Latency
Simple Integer	1
Load/Store	2/1
Branch, Predicate	1
Pattern Match	2
Search Replace	3
Extract, Insert, Subset	1

Table 2: I-COP instruction latencies.

Unless otherwise noted, the I-COP model used in our experiments consists of two I-COP slices, each with four symmetric functional units. The I-COP instruction latencies are shown in Table 2. The I-COP model is integrated with the core processor's simulator and is simulated in detail at the machine cycle level. We also implemented a scheduler to automatically schedule our (sequential) I-COP code for the 4-wide I-COP slice.

Seven SPECint95 benchmarks [28] and three pointer-intensive Olden benchmarks [29] 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 Trace Cache Fill Unit Results

In the prototype I-COP implementation, whenever the

[1] We assume its effective address and base address are available to the I-COP.

fill buffer identifies the end of a trace, it copies the instructions of this trace from the fill buffer to the I-COP memory and inserts a task into the task queue. Whenever an I-COP slice is free, it picks up a task from the front of the task queue and runs the I-COP fill unit code until it finishes constructing the trace and writing it to the trace cache. When the task queue is full, the fill buffer logic stops examining the fill buffer and only resumes when a task queue entry becomes available. When the fill buffer is full, instructions are dropped at basic block boundaries. We first determine the impact of the number of I-COP slices on the performance. Note that using a 4-wide I-COP slice, the I-COP fill unit code (50 instructions) takes only 18 cycles to execute.

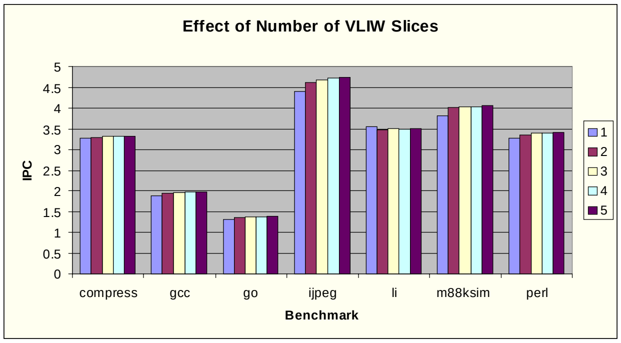
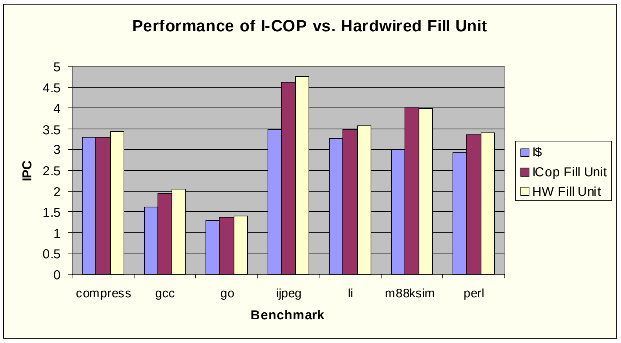


Figure 7. Performance vs. number of I-COP slices.

Figure 7 shows that performance generally increases with the number of I-COP slices. This is because the trace cache is updated more frequently and this generally results in it holding more useful traces. However, in the case of *li*, the frequent trace cache updates may actually be displacing useful traces with less useful ones. Overall, using two slices appears to be a good cost-performance trade-off.

We then compare the performance of the I-COP implementation (using 2 slices) with the hardwired implementation (as described at the end of Section 3.1). We also compare it to the configuration without a trace cache (i.e. instructions are supplied only from the instruction cache). Figure 8 shows that the fill unit implemented as I-COP code approaches within 2.9% (average) of the performance of the hardwired fill unit. This is so even though the I-COP only sees a small fraction of the retiring instructions because two I-COP slices are unable to keep up with the rate of instruction completion. For *m88ksim*, the I-COP actually performs better than the hardwired implementation possibly because the dropped instructions and/or delayed construction of traces prevent useful traces in the trace cache from being displaced by less useful ones.



igure 8. Performance of I-COP vs. hardwired fill unit.

4.3 Register Move Results

Using a 4-wide I-COP slice, the I-COP code for the register move optimization takes 106 cycles to execute (this excludes the 18 cycles required for basic trace construction). Since this is an expensive optimization, the I-COP code uses two heuristics to selectively apply this optimization to only certain traces. First, a trace is not optimized when it is first written into the trace cache. A trace is only optimized if it is already in the trace cache and has been accessed x number of times. We found $x = 5$ to be a good choice. Second, 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.

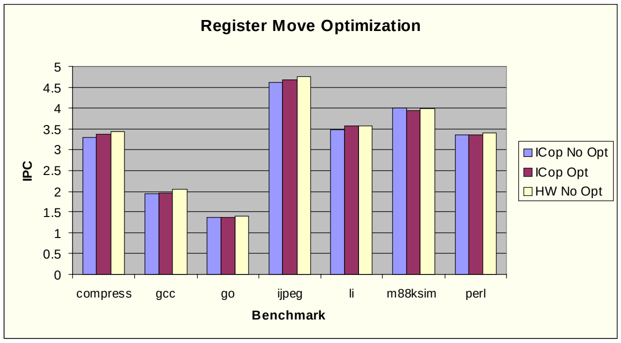


Figure 9. Performance of I-COP Register Move trace optimization.

Figure 9 shows that this optimization benefits three benchmarks, *compress*, *ijpeg* and *li*, by 2.1%, 1.5% and 2.6% respectively. In this experiment, the number of I-COP slices remains at two. Although these performance improvements are modest, no additional I-COP hardware is required. In the case of *li*, this trace optimization enables the I-COP to exceed the performance of the hardwired basic fill unit.

4.4 Data Prefetch Results

In these data prefetch experiments, the number of I-COP slices remains at two. In order not to pollute the data cache, prefetches are brought into a 64 entry prefetch buffer rather than to the cache. The SPECint95 benchmarks are used for the stride prefetching experiments while the pointer-intensive Olden benchmarks are used for the

LDS prefetching experiments.

4.4.1 Stride Prefetching

The I-COP code (130 instructions) for the stride prefetch optimization takes 33 cycles to execute on the 4-wide I-COP slice. This trace optimization benefits every benchmark except *gcc* and *ijpeg*. Figure 10 shows that it improves the performance of *compress*, *go*, *li*, *m88ksim* and *perl* by 6.1%, 2.9%, 5.5%, 8.5% and 2.1% respectively. In fact, the performance of *compress*, *go*, *li*, *m88ksim* and *perl* exceed the performance of the hardwired basic fill unit.

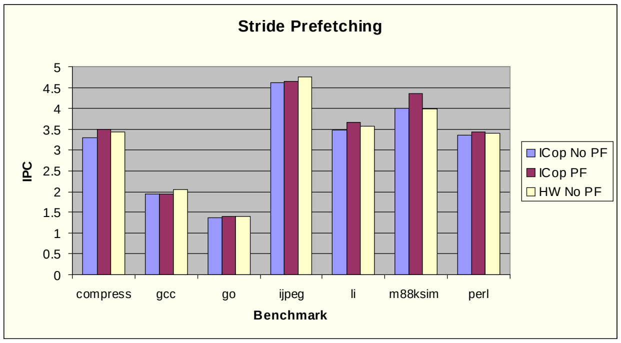


Figure 10. Performance of I-COP stride prefetching.

4.4.2 Linked Data Structure Prefetching

The I-COP code (86 instructions) for the LDS prefetch optimization takes 22 cycles to execute on the 4-wide I-COP slice. Figure 11 shows that this trace optimization benefits all three Olden benchmarks substantially. In this figure, results are shown for prefetching one line as well as two lines. In the case of the two lines prefetch, the second prefetch instruction is only inserted when there is enough space left in that trace cache line. Its base register is set to the same as that of the first prefetch instruction but its offset is equal to the offset of the first prefetch instruction plus the cache line size (32 bytes). The results show that prefetching two lines is superior for all three benchmarks. This two-line LDS prefetching improves the performance of *health*, *perimeter* and *treeadd* by 15.3%, 10.1% and 33.5% respectively. Figure 11 shows that this is significantly better than the performance of the hardwired basic fill unit.

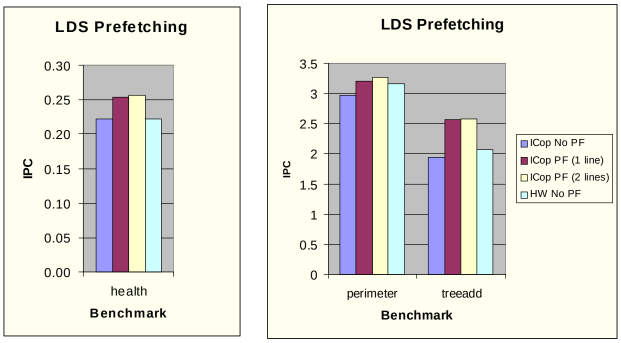


Figure 11. Performance of I-COP LDS prefetching.

These data prefetching results demonstrate the potential of the I-COP to invoke optimizations selectively to improve different types of applications. In our example, we invoked the stride prefetching I-COP code for the SPECint95 benchmarks and the LDS prefetching I-COP code for the pointer-intensive Olden benchmarks. As mentioned in Section 2.1, the mechanisms for this selective invoking of I-COP code based on application behavior are part of our ongoing research.

4.5 Observations

From our experimental results, we make several observations:

1. Because the I-COP is situated at the back-end of the core processor and due to the frequent reuse of traces, the relatively long latency of executing I-COP code (as compared to hardwired implementation) does not have significant impact on overall performance.
2. Because of the frequent reuse of traces, the large number of instructions dropped by the fill buffer (when the I-COP cannot keep up with the core processor) has little impact on performance.
3. Other trace optimizations can be incorporated as additional I-COP routines to achieve further performance gains without requiring any additional I-COP hardware.
4. There is abundant ILP in the I-COP code for trace construction and the three trace optimizations we studied. Table 4 shows their I-COP code sizes and their execution latencies on a 4-wide I-COP slice. We are also able to exploit the task-level parallelism of constructing and optimizing multiple (two in our case since we have two I-COP slices) traces in parallel.

I-COP Application	I-COP Code Size	Execution Latency
Trace Cache Fill Unit	50 lines	18 cycles
Register Move Opt	423 lines	106 cycles
Stride Prefetching	130 lines	33 cycles
LDS Prefetching	86 lines	22 cycles

Table 4: I-COP code characteristics.

5 Related Work

The I-COP transforms the core processor's instructions into a form that can be more efficiently executed. Some static transformations can also be performed by software such as a binary to binary translator [7], although the execution of the translator incurs significant overhead. Dynamic transformations can be performed by a runtime optimizing compiler [17][30]. However, since it requires context switching and is expensive to invoke, the runtime optimizations can only be performed infrequently and only on the most frequently executed instructions. In contrast, the I-COP is running continuously and can therefore be much more adaptive to changes in the behavior of the application running on the core processor. In any case, we view the transformations performed by the I-COP and dynamic compiler as potentially synergistic.

DAISY [20] uses software to dynamically transform instructions from a base architecture (e.g. x86, PowerPC) to their VLIW target architecture. This software is invoked by the runtime system whenever a new fragment of untransformed code is encountered. Their primary aim is to enable the target architecture to achieve full architectural compatibility with the base architecture. Like a dynamic compiler, the instruction transformations incur overhead and may negate the performance gain from executing the VLIW code in place of the base architecture code.

Recently, Chappell et al. [18] and Song and Dubois [19] proposed using subordinate threads to assist the execution of an application thread. This assistance may or may not be related to instruction transformations. Chappell et al. used branch prediction as an example to illustrate the use of their *microthreads* while Song and Dubois used data prefetching as an example to illustrate the use of their *nanothreads*. Unlike an I-COP which is separate from the core processor, these subordinate threads compete with the application thread for the same execution resources. Moreover, the logic required to support these subordinate threads and the arbitration between them and the application thread will complicate the design of the processor and may increase its cycle time. The core processor's ISA may also be inefficient for implementing the desired functionality of the subordinate threads. Although Chappell et al. suggest the addition of special instructions to the core processor's ISA, the increased complexity of decoding these new instructions and the new functional units required to execute them may impact the cycle time of the core processor. Also, unlike an I-COP, Chappell et al's scheme require strong compiler and operating system support.

The term *instruction path coprocessor* was originally coined by Debaere and Campenhout [6]. However, they refer to an instruction translation engine that sits between the instruction memory and the fetch stage of the target processor. In contrast, the I-COP interfaces with the back-end of the core processor and is thus able to exploit code reuse as well as avoid being in the critical path of the core processor.

6 Conclusions and Future Work

In this paper, we propose the concept of an instruction path coprocessor, which is an on-chip programmable coprocessor that operates on instructions of the core processor to transform them into an internal format that can be more efficiently executed. We show how this coprocessor should interface with the core processor and highlight its instruction set and implementation issues. Using four example instruction transformation techniques, we demonstrate the versatility and flexibility of the I-COP to improve the performance of the core processor by implementing these four techniques as I-COP programs that can be selectively invoked at run time.

Our experimental results show that an I-COP implementation of the trace cache fill unit approaches within 2.9% of the performance of a hardwired implementation while permitting great flexibility in implementing different trace construction heuristics and additional trace optimizations. In fact, when the register move and data prefetch trace optimizations are added, the I-COP implementation actually exceeds the performance of the basic hardwired fill unit. These optimizations are implemented by changing the I-COP code and do not require changes to the I-COP itself. In addition, our data prefetching results highlight the potential of the I-COP to selectively invoke different optimizations based on application characteristics.

We view this study as an early exploration of the I-COP concept. The initial findings are quite promising and indicate the potential feasibility and usefulness of the I-COP concept. It can potentially become a valuable technique for achieving further performance gains in future high performance microprocessors. Clearly more follow-on research efforts are needed. Our future research will focus on more hardware-efficient implementations of I-COPs, detailed specification and design of the I-COP/core processor interface, and other promising applications for the I-COP concept.

Acknowledgment

We would like to thank C. K. Luk as well as the members of our Carnegie Mellon Microarchitecture Research Team (CMμART) for their contributions to this work. This work also benefited from machines donated by Intel, and was supported in part by ONR (N00014-97-1-0701, N00014-96-1-0928) and in part by Intel Corp.

References

[1] Michael Slater, "AMD's K5 Designed to Outrun Pentium," in Microprocessor Report, Vol. 8, Issue 14, Oct 1994.

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

[3] E. Rotenberg, S. Bennett and J. Smith, “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching,” in Proc. of 29th Int. Symp. on Microarchitecture, 1996.

[4] 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.

[5] B. Black, B. Rychlik and J. Shen, “The Block-based Trace Cache,” in Proc. of 26th Int. Symp. on Computer Architecture, May1999.

[6] E. Debaere and J. Campenhout, “Interpretation and Instruction Path Coprocessing,” MIT Press, 1990.

[7] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli, J. Yates, “FX!32 - A profile-directed binary translator,” IEEE MICRO, 18(2), March-April 1998.

[8] D. Friendly, S. Patel and Y. Patt, “Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors,” in Proc. of 31st Int. Symp. on Microarchitecture, December 1998.

[9] Q. Jacobson and J. Smith, “Instruction Pre-Processing in Trace Processors,” in Proc. of 5th Int. Symp, on High Performance Computer Architecture, 1999.

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

[11] Microprocessor Report, 5/11/98.

[12] Keith Dieffendorf, “Katmai Enhances MMX,” Microprocessor Report, 10/5/98.

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

[14] R. Nair and M. Hopkins, “Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups,” in Proc. of 24th Int. Symp. on Computer Architecture, June 1997.

[15] M. Franklin and M. Smotherman, “A Fill-Unit Approach to Multiple Instruction Issue,” in Proc. of 27th Int. Symp. on Microarchitecture, December 1994.

[16] E Rotenberg and J. Smith, “Control Independence in Trace Processors,” in Proc. of 32nd Int. Symp. on Microarchitecture, December 1999.

[17] T. Kistler, “Dynamic Runtime Optimization,” in Proc. of the Joint Modular Languages Conference, 1997.

[18] R. Chappell, J. Stark, S. Kim and Y. Patt, “Simultaneous Subordinate Microthreading (SSMT),” in Proc. of 26th Int. Symp. on Computer Architecture, May 1999.

[19] Y. Song and M. Dubois, “Assisted Execution,” Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California, October 1998.

[20] K. Ebcioglu and E. Altman, “DAISY: Dynamic Compilation for 100% Architectural Compatibility,” in Proc. of 24th Int. Symp. on Computer Architecture, June 1997.

[21] M. Schuette, “Exploitation of Instruction-Level Parallelism for Detection of Processor Execution Errors,” Ph.D. Thesis, ECE Department, Carnegie Mellon University, 1991.

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

[23] D. Joseph and D. Grunwald, “Prefetching Using Markov Predictors,” in Proc. of 24th Int. Symp. on Computer Architecture, June 1997.

[24] T. Mowry, “Tolerating Latency Through Software-Controlled Data Prefetching,” Ph.D. Thesis, Stanford University, 1994.

[25] C. Luk and T. Mowry, “Compiler-Based Prefetching for Recursive Data Structures,” in Proc. of 7th ASPLOS, 1996.

[26] A. Roth, A. Moshovos and G. Sohi, “Dependence Based Prefetching for Linked Data Structures,” in Proc. of 8th ASPLOS, October 1998.

[27] A. Roth and G. Sohi, “Effective Jump-Pointer Prefetching for Linked Data Structures,” in Proc. of 26th Int. Symp. on Computer Architecture, May 1999.

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

[29] 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.

[30] R. Hank, W. Hwu and B. Rau, “Region-based Compilation: An Introduction and Motivation,” in Proc. of 28th Int. Symp. on Microarchitecture, December 1995.

[31] Y. Chou and J. Shen, “Instruction Path Coprocessors”, CMuART Tech. Report, Carnegie Mellon Univ., March 2000.

[32] R. Rakvic, B. Black, and J. Shen, “Completion Time Multiple Branch Prediction for Enhancing Trace Cache Performance,” in Proc. of 27th Int. Symp. on Computer Architecture, June 2000.

Appendix A: Special I-COP Instructions

In the following descriptions, the destination register is always listed on the left side of the expression. R indicates that the operand/result is in an integer register and P indicates that it is in a predicate register.

A.1 Pattern Matching Instructions

(1) R_{loc} = **PatternMatch**(nth match, source, $R_{\text{start location}}$, $R_{\text{number of entries to match}}$, $R_{\text{match bit mask}}$, $R_{\text{match pattern}}$, $R_{\text{no match return value}}$)

where nth match = {FIRST, SECOND...}
source = {FILL BUFFER, I-COP MEMORY}

This instruction searches a series of storage locations for a pattern and returns the location of the nth match. If there is no match, the no match return value is returned instead.

(2) $R_{\text{num replaced}}$ = **SearchReplace**(source, $R_{\text{start location}}$, $R_{\text{number of entries to match}}$, $R_{\text{match bit mask}}$, $R_{\text{match pattern}}$, $R_{\text{replace bit mask}}$, $R_{\text{replace\_pattern}}$)

This instruction is similar to the PatternMatch instruction but instead of returning the location of a match, it replaces matching locations with a replacement pattern.

(3) $P_{\text{predicate}}$ = **PatternMatchReg**(R_{source}, $R_{\text{bit mask}}$, $R_{\text{match pattern}}$)

This instruction is a simple form of the PatternMatch instruction. It only examines a single integer register for a pattern match.

A.2 Data Movement Instructions

These instructions are provided to move data between the main processor's microarchitecture structures (e.g. fill buffer, trace cache) and the I-COP data memory.

(4) **xStore**(source, $R_{\text{source location}}$, $R_{\text{memory location}}$, $R_{\text{number of source entries to move}}$)

(5) **xLoad**($R_{\text{memory location}}$, destination, $R_{\text{destination location}}$, $R_{\text{num of memory words to move}}$)

(6) P_{hit} = **Read**(source, $R_{\text{source location}}$, $R_{\text{source tag}}$, $R_{\text{memory location}}$)

xStore moves data into the I-COP memory while xLoad moves data out of it. Read is similar to xStore except that it also returns whether the access was successful. It is used for reading tagged structures like the trace cache.

A.3 Other Special Instructions

Other special instructions are provided to extract bits from an integer register, insert bits into an integer register, and to determine if the contents of one integer register is a subset of the contents in another.

(7) $R_{\text{extracted data}}$ = **Extract**(R_{data}, $R_{\text{bit mask}}$)

(8) R_{result} = **Insert**(R_{src}, $R_{\text{insert data}}$, $R_{\text{insert mask}}$)

(9) P_{true} = **Subset**(R_{value1}, R_{value2})

Appendix B: I-COP Code

To improve readability, instructions are shown in a pseudo C format rather than in the assembly instruction format. Also, variable names are used in place of explicit register numbers.

```
# II. determine whether to write line into T$
temp_loc1 = Load(HW_QUEUE)
closing_loc = Load(temp_loc1 + 64)
temp_loc2 = temp_loc1 + 1024
num_branches = 0
tempA = PatternMatch(FIRST_MATCH, MEMORY, temp_loc1, closing_loc, TYPE_MASK, COND_BR, 16)
p4 = SetPred(tempA != 16)
temp1 = Load(temp_loc2[tempA])
branch_takenA = Extract(temp1, BRANCH_TAKEN_MASK)
p4: num_branches++
p4: new_path = branch_takenA
tempB = PatternMatch(SECOND_MATCH, MEMORY, temp_loc1, closing_loc, TYPE_MASK, COND_BR, 16)
p5 = AndPred(tempB != 16, p4)
temp2 = Load(temp_loc2[tempB])
branch_takenB = Extract(temp2, BRANCH_TAKEN_MASK)
p5: new_path = new_path << 1
p5: num_branches++
p5: new_path += branch_takenB
temp3 = Load(temp_loc2[0])
index = Extract(temp3, INDEX_MASK)
tag = Extract(temp3, TAG_MASK)
p6 = Read(T_CACHE, index, tag, temp_loc3)
temp4 = Load(temp_loc3[97])
old_path = Extract(temp4, PATH_MASK)
p7 = Subset(new_path, old_path)
p7 = OrPred(p6 == 0, !p7)
# III. write T$
p8 = SetPred(branch_takenA == 1)
p8: exit_addressA = Extract(temp1, PC_MASK)
p8: exit_addressA += 4
!p8: exit_addressA = Extract(temp1, BRANCH_TGT_MASK)
# append to end of temp_loc1 after insts
p4: Store(exit_addressA, temp_loc1+64)
p4: Store(branch_takenA, temp_loc1+100)
p9 = SetPred(branch_takenB == 1)
p9: exit_addressB = Extract(temp2, PC_MASK)
p9: exit_addressB += 4
!p9: exit_addressB = Extract(temp2, BRANCH_TGT_MASK)
p5: Store(REG, exit_addressB, MEMORY, temp_loc1+72)
p5: Store(REG, branch_takenB, MEMORY, temp_loc1+101)
temp9 = Load(temp_loc2[closing_loc])
closing_inst_type = Extract(temp9, TYPE_MASK)
not_taken_address = Extract(temp9, PC_MASK)
not_taken_address += 4
taken_address = Extract(temp9, BRANCH_TGT_MASK)
Store(closing_inst_type, temp_loc1+96)
Store(not_taken_address, temp_loc1+88)
Store(taken_address, temp_loc1+80)
Store(new_path, temp_loc1+97)
Store(closing_loc, temp_loc1+98)
Store(num_branches, temp_loc1+96)
num_words = closing_loc + 25
# insert optimization code here
...
WRITE_TRACE:
p7: xLoad(temp_loc1, T_CACHE, index, num_words)
```