2008

Hardware-assisted path profiling in microprocessor systems implemented on top of an FPGA

Brett A. Boren

HARDWARE-ASSISTED PATH PROFILING IN MICROPROCESSOR SYSTEMS IMPLEMENTED ON TOP OF AN FPGA

by

BRETT A. BOREN

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in
The Department of Electrical and Computer Engineering
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2008

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

(student signature)

21 Dec 2007

(date)

THESIS APPROVAL FORM

Submitted by Brett Boren in partial fulfillment of the requirements for the degree of Master of Science in Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of the University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements of the degree of Master of Science in Engineering.

Committee Chair (date)

Rhonda Kay Slade 12/18/07

B. Earl Wells 12/18/07

1/2/08 Department Chair

J. Cannon 1/3/08 College Dean

Allen M. Moriarty 4/15/08 Graduate Dean

ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science in Engineering

College/Dept. Engineering/Electrical and Computer Engineering

Name of Candidate Brett Boren
Title Hardware-Assisted Path Profiling in Microprocessor Systems Implemented on Top of an FPGA

This thesis focuses on a low overhead technique for collecting low-level execution profiles from programs. These profiles are very useful to processor designers working on future microprocessors and to software designers seeking to exploit the maximum performance of current hardware. As part of this technique, we develop an algorithm for collecting basic blocks into paths using program loop structure and then record the execution frequency of these paths. We then implement that algorithm in both pure software and in a mixed hardware/software environment and perform experiments to verify functionality and evaluate performance. Our results show that a software approach to path profiling is viable and that a dedicated hardware collector and compressor approach improves performance considerably.

Abstract Approval:

Committee Chair

Department Chair

Graduate Dean Debra M. Moriarity

ACKNOWLEDGMENTS

The work described in this thesis would not have been possible without the assistance of a number of people who deserve special mention. First, my adviser, Dr. Kulick, was patient and persistent from beginning to end and a constant source of good ideas and perspective. To my committee, thank you for good feedback and corrections. To Casey Pogue, Jay Olive, Shane Basham, and Dr. Will Cohen, for laying the groundwork that much of this work was built upon. To Mr. Richard Ciliax and the US Army Research, Development, and Engineering Command, for funding my initial research assistantship.

To Ken Lesueur, Leah Cunningham, and Mike Krause, thank you for believing that I could finish this thesis and creating opportunities for me to make that happen. Without your support I would likely just be another person with an unfinished master's degree.

To my family, thank you for instilling within me a desire to achieve and excel and for continuing to encourage me through the years. Lastly but most importantly, to my wife, Rebecca, I cannot begin to express my thanks for holding on these several years while we lived in limbo between school and the rest of our lives. Thank you for enduring my mistakes and motivating me when I could not motivate myself.

Soli Deo Gloria

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Program trace analysis is a tool frequently used to understand actual program execution. Most software analysis tools are concerned with static analysis of a program and operate by analyzing a program as written. Few tools exist for determining actual program execution. While the output or proper functioning of the program is proof of correct execution, it provides little information about the detailed internal operations of the program execution. For example, simple static analysis or verification of correct outputs cannot reveal whether a program is just meeting deadlines or has significant slack time for future improvements. Practically all optimization questions hinge on being able to collect data about the internal workings of an executing program. The primary approach to answering these questions requires detailed recording of the program trace history. The collection and compression of this information in real time with minimal impact to the original application is the focus of this work.

1.1 Profiling and Tracing

Profiling is a loosely used term that generally refers to collecting data based on the execution of a program. This data is usually statistical and is best described as a set of counts for particular events during the program's execution. Tracing, a complement to profiling, is a technique used to collect sequential data is collected about a program's

execution. This technique allows information about combinations of events to be recorded. Profiling can be done to various degrees of invasiveness to the executing program, with the more invasive profiling incurring a higher performance penalty while also providing more accurate data. This work attempts to find some method to incorporate the strengths of both techniques.

1.2 Performance Monitoring Hardware

In the last 10 years, innovations in performance monitoring hardware have greatly aided data collection by providing facilities for producing very fine grain processor performance statistics gathered by the processor itself. Examples include counting retired instructions, cache misses, and branch prediction rates. Using these statistics, software developers can optimize software for particular processors. Since these statistics are continuously accumulated, the counters themselves have to be sampled by instrumentation code that introduces extremely high overheads during the sampling of the performance monitoring hardware.

High overhead is not the only consideration when using performance monitoring hardware. There is a tradeoff between accuracy, overhead, and the ability to correlate the measurements back to specific instructions and the code that generated those instructions. For example, the more the performance hardware is sampled, the less accurate the results will be since the sampling instructions will change the processor performance in a manner proportional to the frequency of the sampling. However, more samples mean that the numbers can be more exactly correlated back to an instruction or range of instructions within the program. Also, with more sampling comes more execution overhead since the ratio of real program instructions to instrumentation instructions goes down. On the other

hand, when the counters are sampled less frequently the data is more accurate but also more aggregated meaning the correlation of the measurements to specific instructions is difficult.

1.3 Problem Statement

Overhead is the primary limiting factor on performing tracing analyses. The sampling of performance monitoring hardware can be a very inexpensive way to acquire execution data when used at a system level, but it is not suitable for fine-grained data collection because the data collection significantly affects execution characteristics. For instance, sampling performance monitoring hardware at each basic block traversal can incur a 50% execution penalty because, according to our research, the average basic block length is about 6 instructions and it takes anywhere from two to three instructions to sample a performance register. While more advanced techniques [4] can usually cut the overhead in half, even 20% is a significant overhead in cases such as batch processing or in timing-critical applications.

1.4 Proposed Solution

Tracing programs generate an enormous amount of redundant information, because the majority of the execution time for most programs is spent in a loop or several loops that can generate redundant information with each traversal of the loop. If this redundant sequential information, from here on called paths, can be reduced or compressed the savings in trace file size and Input/Output (I/O) traffic would be appreciable.

This new technique is called "loop-oriented path-profiling" because the frequencies of specific paths within the context of loops are recorded. The key premise of

this thesis is that the additional overhead introduced by matching paths in software could be moved to a co-processor. The goal of this research is to show that hardware-assisted path-profiling greatly reduces the overhead of pure software path-profiling.

CHAPTER 2

LITERATURE REVIEW

Several techniques exist for gathering data about the execution of a program. These techniques are a mixture of hardware and software approaches and most suffer from the same basic problems of temporal overhead, data volume, and other undesired effects on the executed program such as cache pollution. These techniques can be categorized under either profiling or tracing, and both types of techniques can utilize performance monitoring hardware. However, before discussing these approaches some review of Control-Flow Graphs is necessary.

2.1 Control-Flow Graphs

Programs can be mapped into a Control Flow Graph (CFG), a graph whose vertices are basic blocks and whose edges represent transfers of control within the program [4]. A basic block is a contiguous sequence of code in which all instructions execute if the first instruction executes.

Figure 2.1 demonstrates how a simple program can be represented in a CFG. In modern processor architectures, where high degrees of parallelism are extracted via massively pipelined execution units, the control of flow and the correct predictions of control flow are critical to performance. The CFG is therefore a useful tool when

performing static code analysis since its edges represent the places where control flow can negatively affect execution by changing direction.

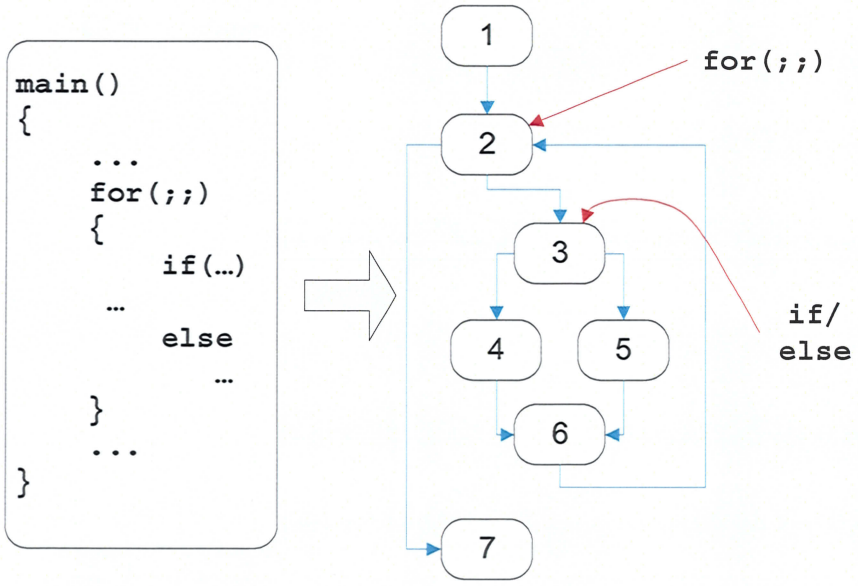

Figure 2.1 Mapping a program into a CFG

Compilers use a CFG representation of a program in order to optimize its execution on a certain platform. The example in Figure 2.2 compares unoptimized and optimized versions of assembly generated from a loop. In the unoptimized version, branch instructions are used for both outcomes of the loop test component causing processor pipeline stalls on each loop iteration. In the optimized version, only one outcome results in a branch. Further optimization would arrange for the least likely outcome to get the branch (in the case of a for-loop, the failing case should be the least likely outcome).

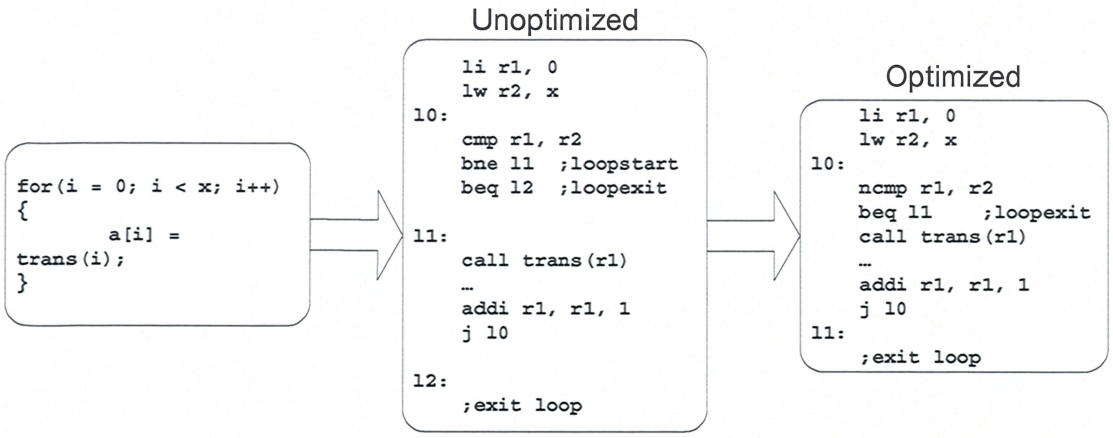

Figure 2.2 Optimized vs. unoptimized assembly

2.2 Profiling

Profiling is a technique that involves gathering statistical data about an executing program that may or may not involve changing the compiled program. Profiling is generally a low overhead technique to gather data since it usually involves incrementing counters [4]. For example, a profile could be made of all the branch instructions in a program and whether or not the branch was taken each time that branch is executed. Returning to the example in Figure 2.2, this kind of profile could provide real statistics to help the compiler's optimizer choose the branch outcome least likely to occur. Another of the more common uses of profiling is to determine code coverage, or the fraction of the code written that was actually executed for a particular execution or some set of executions. Profiling can be conducted at different levels within the execution of a program: from system-level to function or subroutine level, all the way down to a basic block or instruction level [11].

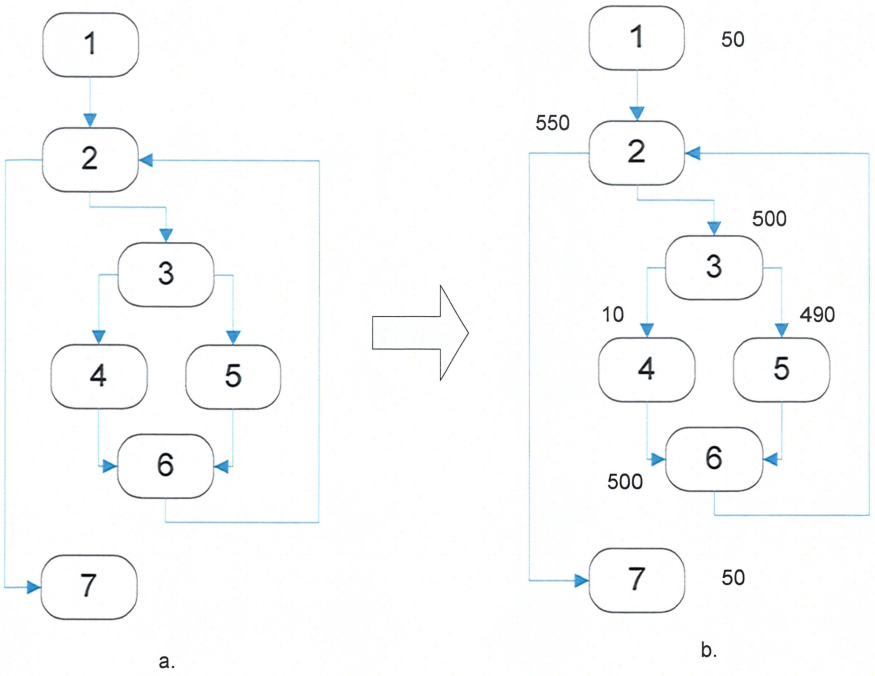

Figure 2.3 A CFG (a) annotated weights representing execution frequency (b)

Basic block profiling can be used to gather fine-grained profile data about an executing program by instrumenting the program at the CFG vertices, or basic blocks, to increment counters unique to each block. By analyzing all of the counters from a program execution, a weighted CFG can be constructed. As an example, consider the CFG of the hypothetical procedure presented in Figure 2.1 and repeated in Figure 2.3(a). The weighted CFG shown in Figure 2.3(b) is generated by back-annotating an arbitrary execution profile onto the original CFG. Recall from Figure 2.1 that the function represents a loop containing an if-else clause. For this execution, the function was executed 50 times, the loop iterated 500 times (10 times per function call on average), and the inner if-else clause branched to block "5" 490 of 500 times, or 98% of the time. This information is enough for a compiler to reorient the branch instructions for the

instrumented function to assume that the "if" at block three will go to block five and that the loop test at block "2" should succeed.

A similar technique called edge profiling [4] places instrumentation code on edges. This can be tricky because edges connect blocks of code but contain no code themselves. Figure 2.4(b) shows what the same execution profile used in Figure 2.3(b) would look like as an edge profile. The edge profiling algorithm is implemented in the tool *qpt* [4].

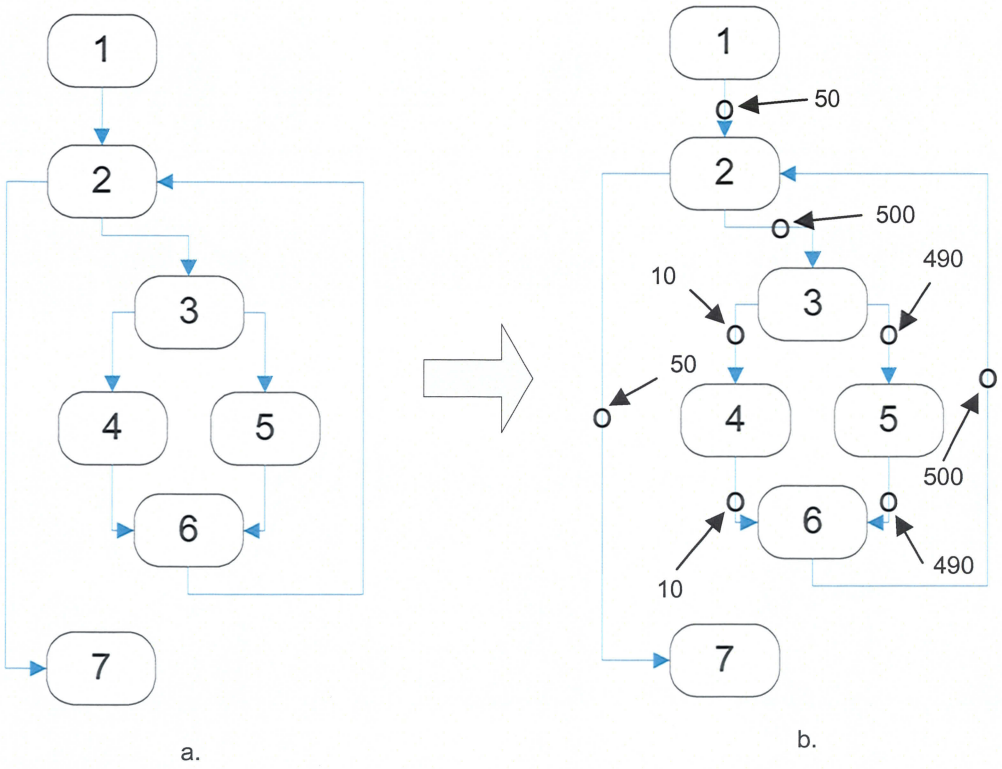

Figure 2.4 A CFG (a) with edge profiling instrumentation points and example execution frequencies (b)

Methods exist for minimizing the number of instrumentation points required to accurately profile a program [4]. These approaches break the program into its functions,

determine an initial counter placement based on a spanning tree of the CFG for that particular function, and then execute the program to get a profile. This profile is then used to get a minimal spanning tree of each function's CFG. Figure 2.5(b) shows the minimum spanning tree of the edge profile annotated CFG from Figure 2.4(b). The edge profile collected can be processed back into a block profile. According to Ball and Larus [4], the minimal edge profiling approach is at least as efficient in terms of instrumentation overhead as the minimal basic block profiling approach and usually better.

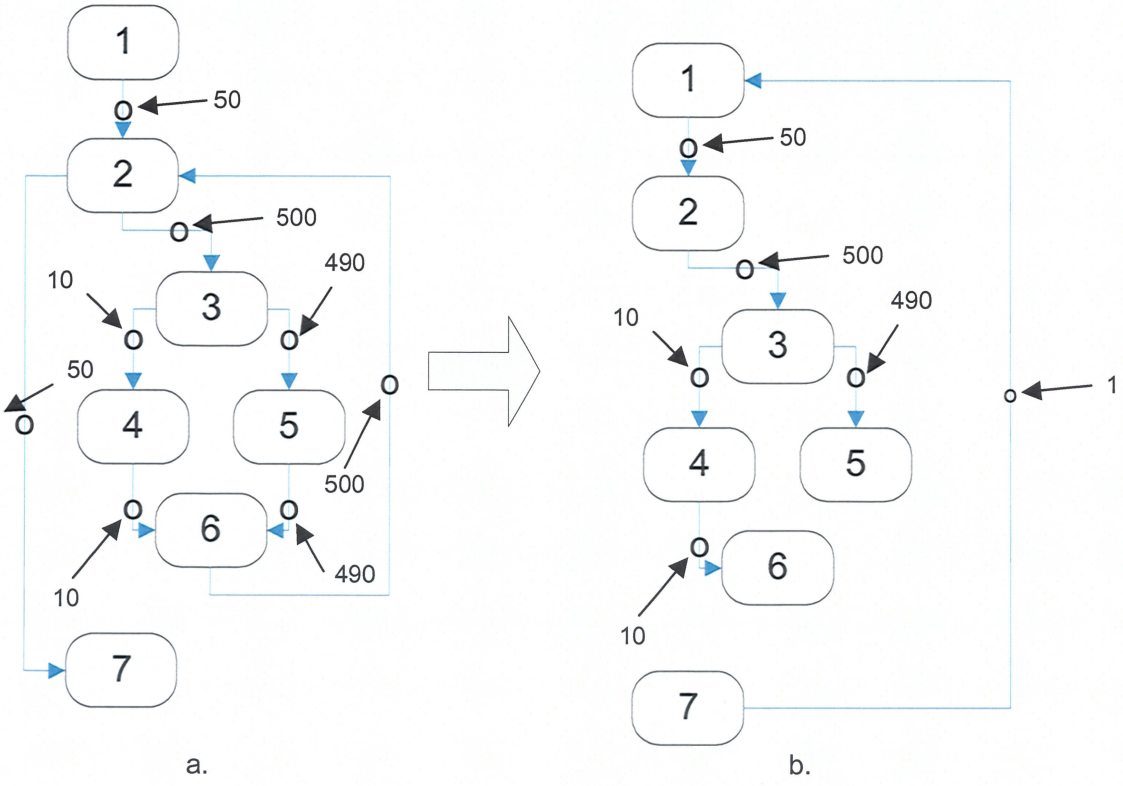

Figure 2.5 A minimal spanning tree (b) derived from a weighted CFG generated by an edge profile

2.3 Tracing

Profiling has limitations on the amount of data it can provide about program behavior changes during execution. A block or edge profile only contains a set of block execution frequencies for the program, but more advanced studies of a program execution require sequential information as to the order in which different code sequences were executed as well as information about variations in the execution of code. The lack of sequential and temporal information in profiling gives rise to another technique called tracing. Basic block tracing places instrumentation code at the beginning of each basic block just as in basic block profiling, but it writes out a trace event called a *witness* instead of incrementing a counter. Figure 2.6 shows the edge profiled CFG from Figure 2.4(b) and what a trace from that same simulated execution might look like. The profile can just provide weights, but the trace provides execution order via the sequence of witnesses. It is worth noting that a trace can be processed back into a profile. Because each witness is unique and sequential, a complete picture of a program's execution through its CFG can be derived. Other tracing methods can be employed for recording sequences of memory accesses during program execution, but they typically involve simulations, such as SimpleScalar [7] rather than direct execution of the instrumented programs due to complexity of extracting all of the addresses from a running program. These simulations are very time consuming and as such are not the focus of this work.

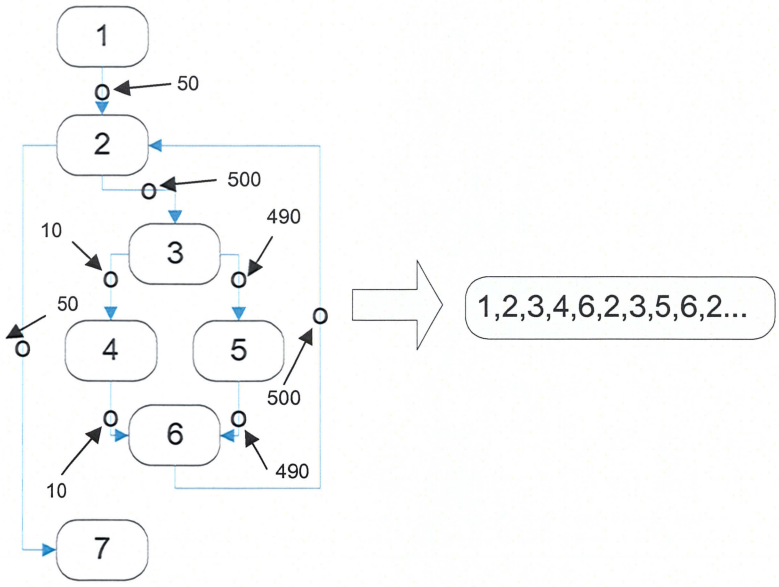

Figure 2.6: An instance of an execution trace generated by an instrumented program

2.4 Path Profiling

Another type of profiling exists that can be characterized as a combination of the other profiling methods and tracing. This technique has roughly twice the overhead of the profiling techniques but far less than the pure tracing techniques while still providing sequential data localized to instances of functions. This method will be referred to as Ball-Larus path-profiling and involves modifying the edge-profiling algorithm to change state bits instead of incrementing counters [3]. By predetermining all of the unique paths through a function at compile time, a state machine can be constructed representing the function's CFG. At the end of the function, the ending state uniquely identifies the path taken and can be used as an index into an array of counters. This method is advantageous because it stores sequential data beyond the block and edge frequencies, thus providing some idea of (beyond-near-term) consequences for compiler optimizers or cache pre-loaders. The Ball-Larus Path profiling algorithm was implemented in the tool *PP*; the

example given in [3] of its application to a CFG can be found in Figure 2.7. Figure 2.7(a) depicts a CFG whose edges have been weighted in such a way that each path from block A to block F will have a unique value if the weights on the traveled edges were added together. The chart in Figure 2.7(b) is created by the algorithm as it walks backwards from block F up to block A in topographical order with the Numpaths(v) representing the number of paths from a block v to the last block (block F). The second part of the algorithm uses an edge profile to determine the least frequently traversed edges to place a minimal set of incrementers that will guarantee each path has a unique sum of edge weights as seen in Figure 2.7(c).

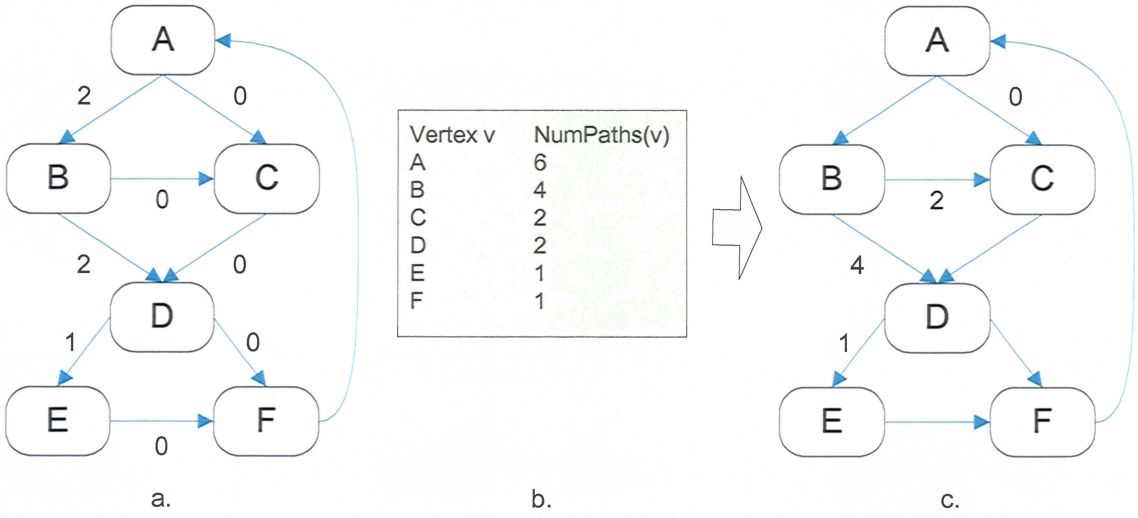

Figure 2.7 An example application of the Ball-Larus path profiling algorithm

A further refinement of the Ball-Larus path-profile is the Whole Program Path (WPP) [10]. While not changing path collection, the WPP approach deals with the compression of the generated path profiles and analyzing that compact form to locate

frequently executed paths. Further extensions to the Ball-Larus path have enabled better inter-procedural path profiling [13].

2.5 Performance Monitoring Hardware

Performance monitoring hardware is a fairly young technology in the computing world [17]. This technology can provide many profiling and tracing techniques with low-level program execution statistics such as cache miss rates. Those statistics can be analyzed by post-processing so that, for instance, a compiler might restructure code to minimize the overall cache miss rate. Performance monitoring hardware usually takes the form of a set of registers that can be configured to count several isolated events within the processor. Recently, processors have introduced counters that can count correlated events such as pipeline stalls due to cache misses. These counters can either be sampled regularly, or they can be set to generate an interrupt on overflow. Currently, the performance monitoring hardware does not involve very complex logic, relying on just some registers, multiplexers, and adders. This simplicity explains why processor manufacturers do not mind building this hardware into their chip designs.

Early in the 1990s, specialized peripheral hardware such as the National Institute of Standards and Technology (NIST) MultiKron system [12] was developed that could be attached to a Versa Module Europa (VME) bus, Peripheral Component Interconnect (PCI) bus, or Sun SBus in order to gather performance data. Counters existed on the MultiKron boards that could be accessed from a running program. Additionally, for Symmetric Multi-Processor (SMP) systems, the MultiKron boards had the advantage of not being biased towards any particular processor so good timing measurements of code executing on multiple processors was possible. Experiments had to be carefully timed

since the MultiKron boards only had a limited amount of memory and bandwidth. This technology, however, suffers further from the severe slowdown associated with the distance from the actual processor and the disparity between the host bus speed and the processor speed. In context, when the MultiKron systems were first being developed, PCI busses were 32-bits wide and ran at 33 MHz while the processors and memory were only running at 66MHz [16]. However, modern processors run anywhere from 2-4 GHz with memory clock-rates up to 1.3GHz, but the PCI bus has only grown to 133MHz at best with 33-66MHz being the most common [2]. PCI Express addresses some of this shortcoming but still must run much slower (250 MHz) than the processor due to physical distance limitations [5]. Also, since the performance hardware didn't actually exist on the processor, additional bus traffic was required to read and set counters and many of the statistics most sought after, such as cache performance, were very difficult to accurately generate.

ProfileMe [8] is an approach conceived by DEC engineers to allow specific individual instructions to be profiled. This method reveals no program structure data, but does track all events occurring due to a particular instruction (cycles/pipeline stage, cache misses, memory access, instruction retirement or abortion) as it is executed and reports back via an interrupt. In practice, profiling an entire program requires several executions with random sampling of instructions on each run and sufficient distance in between samples so as to minimize overhead. As expected, this technique can provide very good feedback for optimization because highly accurate instruction-level data can be gathered and aggregated to identify performance bottlenecks, and the cost in hardware is related only to the maximum number of individual instructions that are to be monitored

simultaneously. This low cost contrasts against typical profiling which gathers aggregate data and where performance is inferred from the frequency of a particular event. This approach is not useful in non-repeatable situations such as real-time intensive applications.

More recently [19], a simulated hardware implementation of a path profiler has been developed using the SimpleScalar processor simulator [6]. In this approach no software has to be changed: instead, the hardware uses branch addresses to detect paths. The fact that no software changes are required is a key strength to this approach given the "intractable" problem of attributing certain processor events back to real program instructions intermixed with instrumentation instructions [1]. To handle the uncertainty surrounding the number of paths an arbitrary program might have, this implementation only tracks hot-paths, frequently executed paths, and discards the infrequently executed paths. The overhead measured for such a hardware-based tracer was 6%, a significant improvement over the 30-45% typical overhead of software-only approaches. Additionally, the same platform was used to collect power usage profiles annotated back to particular paths and sequences of paths.

Probably the most popular way to make use of performance monitoring hardware is a technique called system-wide profiling, which collects data without changing the executing program [11]. This technique involves taking advantage of the operating system's preemptive multitasking mechanism by inserting instrumentation code into the time-slice logic that runs as processes are being moved in and out of the run state. These transitions can be a result of a timer interrupt, I/O processing, or the interrupts generated by performance monitoring hardware because of a counter overflow. The counters can

also be initialized with pre-computed values to force them to overflow and cause an interrupt at a user-tunable frequency. By saving the Program Counter (PC) when overflows occur, a post processing tool can identify the code executing when the samples were taken and use this information to back-annotate the original source. This approach can also be used to profile the kernel itself since the processor's performance monitoring configuration can be set to persist across context switches.

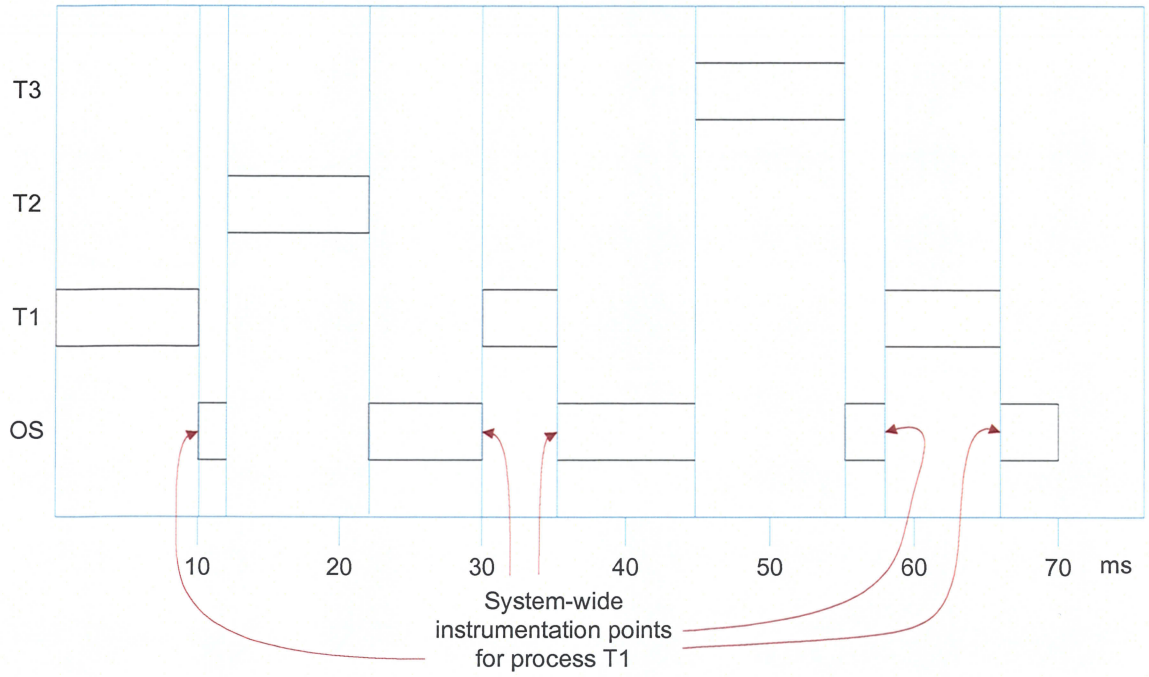

Figure 2.8 Timeline showing how system-wide profiling can instrument a running program using the pre-emptive multitasking OS mechanism

In Figure 2.8, tasks T1, T2, and T3 are executing and being swapped in and out of execution by the Operating System (OS). In this case the process of interest is T1, so the OS generates a profile as T1 moves in and out of execution. Nothing about T1's execution has changed other than a small net runtime delay since the OS portions of the

profile generation use up a few more cycles. The open-source tool OProfile for Linux [11] and Intel's vTune tool for Linux and Microsoft Windows [20] are examples of tools that are based around a system-wide profiling approach.

While system-wide profiling can provide very accurate data with little to no overhead, it is fundamentally limited in the types of data it can provide. Only aggregate performance can be measured because program structure cannot be obtained without gathering more specific data.

2.6 Origins of Current Work

The research reported in this thesis originated within the University of Alabama in Huntsville and the Software Engineering Directorate of the US Army's Aviation and Missile Command. The work was started by Will Cohen and first implemented in a tool called Dynamic Software Thread Execution Profiling (DSTEP). DSTEP was used for analyzing the real-time behavior of simulated processors inside missile system simulators. This analysis was based on the start and end times of functions executing on parallel simulated processors, allowing a developer to quickly identify the critical path of the missile software at a function level. DSTEP used a low-level instrumentation library called Logit along with changes to the GNU Compiler Collection (GCC) [18] to insert instrumentation calls at function entry and exit. Additionally, Logit used the *perfctr* framework [15] to access hardware performance counters so that the amount of time spent in execution for a particular function could be known. The resulting execution trace described which functions ran on which virtual processor and for how long. With this data, DSTEP could display which functions were running on the virtual processors and where the critical path lay, accounting for all concurrently executing tasks.

This thesis reports on an extension of DSTEP to refine that granularity from the function level to the basic block level. The resulting avalanche of data prompted the loop-oriented path-profiling approach described in the next chapter.

CHAPTER 3

THEORY AND ENABLING TECHNOLOGIES

This research introduces the idea of a loop-oriented path profile and demonstrates its utility in both a pure software implementation and in a mixed hardware/software implementation.

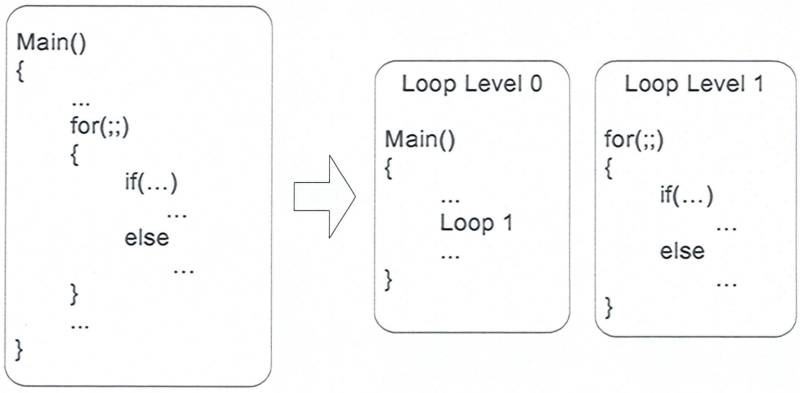

Figure 3.1 A program partitioned according to its loop structure

The loop-oriented path profile is a variant of the Ball-Larus Path Profile with the focus on loops instead of functions. In this technique, a path is defined as a unique sequence of basic blocks from the start of a loop iteration to the end of that iteration, including any nested loops or function calls. From this broad definition, we can express

the entire program execution in terms of paths within nested loops, much like the code itself appears when properly formatted. Even the entrance to a program can fit into this model as a loop of zero depth that executes only once. By classifying every block of code into one of these paths, a picture can be made of a program's execution in terms of it stepping into and out of loops. The outermost loop iteration, referred to as Loop Level 0, is executed only once.

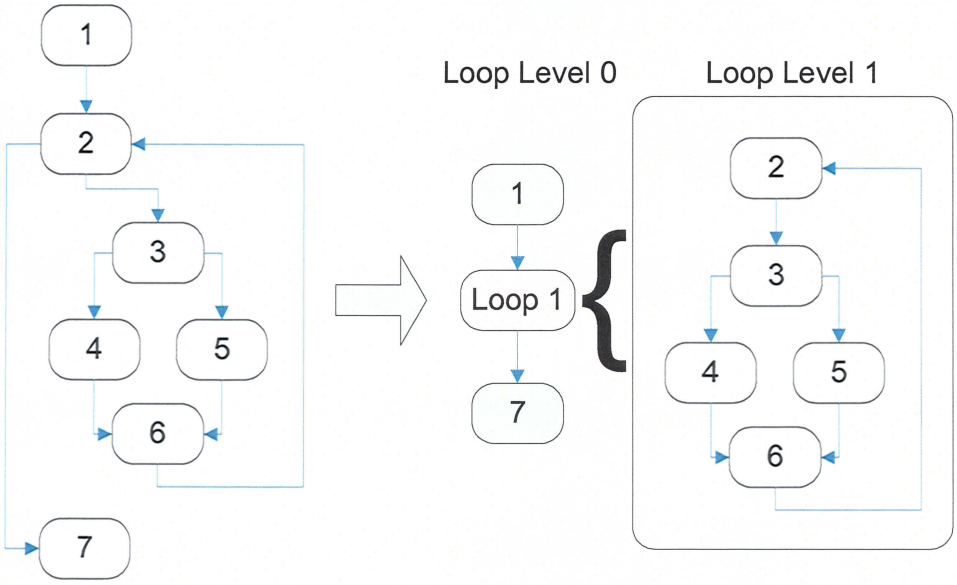

Figure 3.2 A CFG partitioned by loop-oriented Path profile algorithm

Figure 3.1 shows how a function with a single loop fits into the loop-oriented path model. Any nested loops would appear at deeper loop levels. This contrasts with the Ball-Larus approach in that the function is cut cleanly along the lines of loop entry and exit, rather than loop back-edges. Figure 3.2 displays the partition of the CFG of a function containing a loop into two CFGs using the loop-oriented path profile technique. With this

partitioning, it is easy to see how the outermost "loop" has one path executed once and the innermost loop has two paths.

The loop-oriented path-profile approach seeks primarily to preserve data at a basic block level while minimizing overhead in terms of instrumentation execution time and in I/O traffic resulting from collected data. The Ball-Larus path profiling method [3] and its derivatives are only able to count or log events [1] at the path level. The ProfileMe method [8] could randomly acquire events down to the instruction level. There are other applications which may still require granularity between the path and instruction level and need continuous, not random data gathering. Logit and DSTEP were developed to help identify critical paths at the granularity of functions calls; this work seeks to extend that approach from function blocks to basic blocks.

To reach our goal, we take a step back and return to basic block profiling. In addition to placing instrumentation code on basic block boundaries, we also place instrumentation on loop entrances and exits as well as the loop's back-edge to separate loop iterations as in Figure 3.3(a). Instead of recording each basic block event, we can aggregate multiple basic block events into a loop-oriented path. For a particular instance of a loop, some finite number of paths will be encountered and each distinct one is collected and the corresponding frequencies recorded as in Figure 3.3(b). A path is complete when either the loop exits or a new iteration begins. A checksum is maintained as each path is collected so that on completion, each path can be checked against the stored paths first for length, then the checksum, and lastly on a block-by-block basis. At loop exit, the list of collected paths is output along with the frequencies of each path. Additionally, Figure 3.3(c) shows a "superblock" representation derived from the sum of

the path checksums. The superblock is recorded into the parent loop so that the child loop's individual behavior can be differentiated through different iterations of the parent loop, because the superblock number will change based on the paths seen in the loop execution. In Figure 3.2, the CFG is broken into two loop levels with the inner, or child loop being represented as a superblock in the parent, or outer loop.

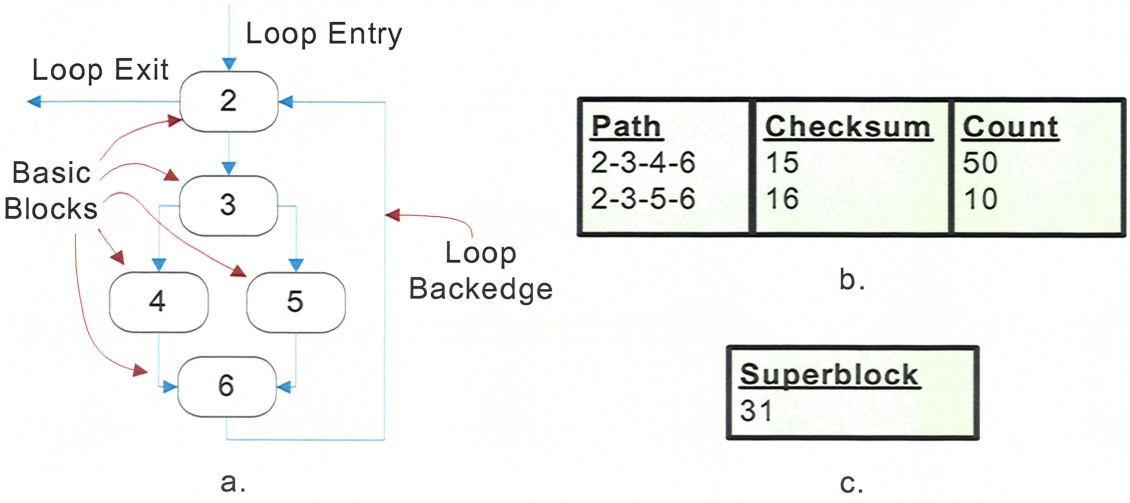

Figure 3.3: A CFG with instrumentation points used for the loop oriented path-profiling technique (a) and the paths, checksums and representative frequency for that CFG (b) along with the superblock computed for that loop execution (c)

It should be noted that this approach also works for inter-procedural analysis since function entry and exit do not affect the loop tracing code. In a way, this is as if all functions are "inlined" since the called function's path or paths will appear embedded in the path from the calling function.

Unlike the Ball-Larus algorithm, this approach allows instrumentation of a program without requiring pre-processing of the CFG. As was described in Chapter 2, this method begins by dividing the program into its functions and then inserting

instrumentation along an optimized subset of each individual function's control-flow edges. Because the Ball-Larus algorithm only collects data at the granularity of a path, it is limited in gathering data at a basic block level. Also, since an in-depth analysis of the program's CFG is required a hardware-only implementation would be very difficult. However, a simulated hardware implementation of the runtime portion has already been demonstrated which only uses branch addresses in infer CFG structure [19].

3.1 Algorithm Description and Example

The severe slowdown introduced by the block tracing code inspired research to compress the block traces in real time. The path-tracing approach was introduced with a focus on collecting performance monitoring data along with the path traces. Wanting to leverage the GCC work already performed, it was discovered that GCC left markers for the loop entry, exit, and continue events in its intermediate representation. Macros were written that added Logit library calls at these locations, and corresponding calls were implemented in the Logit instrumentation library along with the necessary shared data structures. The loop-oriented path profiling algorithm was implemented with the basic block instrumentation collecting the retired instruction count numbers to be embedded in the paths. When a particular path reached the end of the loop and matched an existing path, its block instruction counts were averaged into the recorded block instruction counts and path repetition count.

Figure 3.5 shows the CFG of a program containing a loop nested within a loop such as would be found in a function that traversed a 2D matrix. Any time a basic block (BB) is encountered, a basic block event occurs that adds that block number to a list of up to 10 blocks representing the current path and adds that block number into the checksum representing the current path. As the program progresses from block 1 to block 2 a loop is entered, prompting a loop start (LS) event. Since the application always executes in the context of a loop, an LS event generates a save of that context and prepares to profile a new loop. The loop start event is described more thoroughly in Figure 3.6.

```
Push paths, pathlengths, repetition counts, and
    checksums onto stack
Push "others" counter onto stack
Push "number of paths observed at this loop
    level" onto stack
Push "current path in progress" onto stack

Reset all path registers
```

Figure 3.6 Loop start event pseudocode

Another LS event is generated as execution moves from block 2 to block 3. This is a very simple loop, so the real work of this program is done in block 4 (for instance a matrix operation such as 'a[i][j] = b[i][j];'). As execution moves from block 4 along the loop back-edge to block 3, the loop iteration is complete, so a loop continue (LC) event is generated. The LC does two things. It compares the path just collected to any other paths previously observed by first comparing their length, then their checksum, then their blocks and either increments the count of a matched path or adds the path to the list. The loop continue event is described more thoroughly in Figure 3.7.

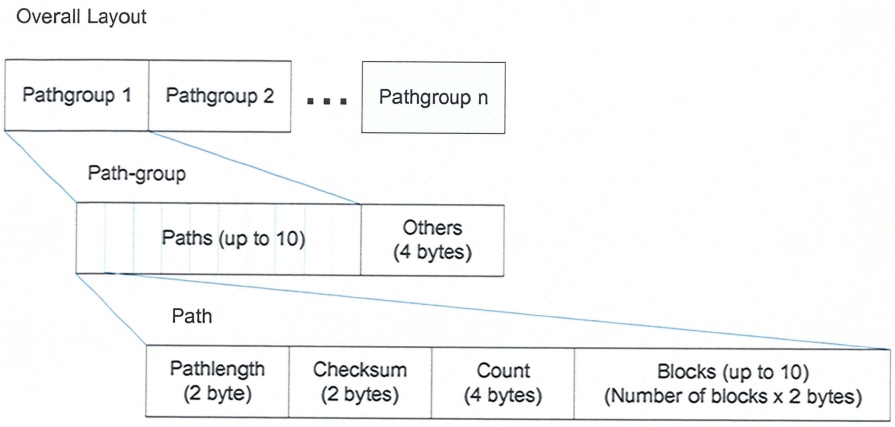

Figure 3.9 Data format for loop-oriented path profile

3.2 Initial Implementation

The loop-oriented path-profile technique has been implemented in the Logit software tool. As previously noted, Logit uses a combination of compiler modifications and an instrumentation library to accomplish its tracing in pure software without any dedicated hardware. We extended it to trace basic blocks and loop entries, exits, and loop continues (events that occur when control passes along the loop back-edge – see Figure 3.3). First, basic block tracing was introduced. The GCC compiler version 3.0 block profiling infrastructure was modified to insert a call to the Logit library each time a new basic block was entered. This approach allowed the optimizers to run and gave us good correlation back to the original source code without having to hand-modify source-code as in Figure 3.4. However, this drastically slowed down program execution since the instrumentation library not only did a large amount of I/O, but also used the perfctr library [15] to record the basic block's retired instruction count. The perfctr library was discovered to rely upon the notoriously slow Linux ioctl system call which, combined with the overhead generated by the Logit library, resulted in a three order of magnitude

slowdown of the instrumented program compared to the uninstrumented software. However, even after removing the perfctr calls, at least an order of magnitude overhead was still observed.

It was noted that the comparison logic, path collection and storage were ideal candidates for hardware implementation to spare the processor from doing much of the instrumentation workload. Table 3.1 shows how widely the instrumentation overhead can vary based upon the control-flow characteristics of the application. The first program is a standard DES encryption implementation. The second is a Bubblesort implementation that sorts a 50 element array that has already been sorted backwards which is the worst-case scenario for Bubblesort. The third is a Fast-Fourier Transform (FFT) implementation. The last is a test program used to stress the loop-oriented path profiling algorithm.

Table 3.1 Software-only logger times versus the original application execution times

Application	original time (cycles)	Original With Software Logit time (cycles)	Ratio
DES	34928	422826	12.11
Bubblesort	20047	565513	28.21
FFT	438580	688140	1.57
Test	15471	579063	37.43

A separate implementation was done to move the collection and comparison logic and memory into hardware in order to minimize the software instrumentation overhead. The required software writes code-words composed of an event type and an optional data argument such as a basic block number to the hardware tracing unit at event boundaries, allowing the hardware to work in parallel to the processor. The hardware has

two tasks: reducing the data being recorded using the path profile algorithm, and recording the reduced traces to memory without involving the processor. The traces in memory can be recovered after main processing provided that the amount of trace data does not exceed the size of the memory buffer.

It was decided to implement this path logger in FPGA fabric instead of in a processor system simulator like SimpleScalar [7]. The strength of this approach is that it can deliver a working system that takes into account the complexities of a real microprocessor system that may not necessarily be modeled in a processor simulator [19]. The most obvious downside of this work was that it was a much more lossy profile than a pure Ball-Larus path profile or a loop-oriented path profile. Due to the finite resources of a processor (even a simulated one) and the difficulty of quickly moving a large amount of data off the processor without affecting processor performance, the author decided to focus only on Hot Paths, or paths most frequently observed for the entire program, and thus collect only a small amount of path information that reflected the majority of the runtime of the program. Also, the problem of getting this information off the processor is not addressed because it is a small amount of data. This method cannot be used for analyzing code coverage or for doing a true critical path analysis since infrequently executed paths may occur along the critical path.

3.3 Enabling Products and Technologies

We used a Xilinx ml401 evaluation board along with the Xilinx Embedded Developer's Kit (EDK) version 8.2 and Integrated Software Environment (ISE) 8.2 [21]. The ml401 Virtex4 evaluation board shown in Figures 3.10 and 3.11 contains, among

other things, an onboard Virtex4-LX25 FPGA, 64MB DDR-RAM, a 1MB SRAM, and the pin-outs necessary to connect a DB-9 serial cable.

We used a unique mechanism of some of the Xilinx Spartan and Virtex FPGA models, the Fast Simplex Link (FSL). The FSL is a "uni-directional point-to-point communication channel bus used to perform fast communication between any two design elements on the FPGA when implementing an interface to the FSL bus" [22]. It provides the ideal low-overhead, simple communication route from the processor where the instrumented program is running to the path processing logic. In plain language, the FSL connection inside the Microblaze processor requires a single instruction to write a single 32bit word compared to as many as three instructions when using the standard On-Chip Peripheral Bus (OPB) interface. Additionally, the FSL interface, having only a sender and a receiver, will only block when its first-in, first-out buffer (FIFO) is full whereas, a bus interface like the OPB must deal with contention and a negotiation protocol due to the multiple end-points of a general-purpose bus. The performance difference is substantial, ranging from 2 to 10 times faster [9].

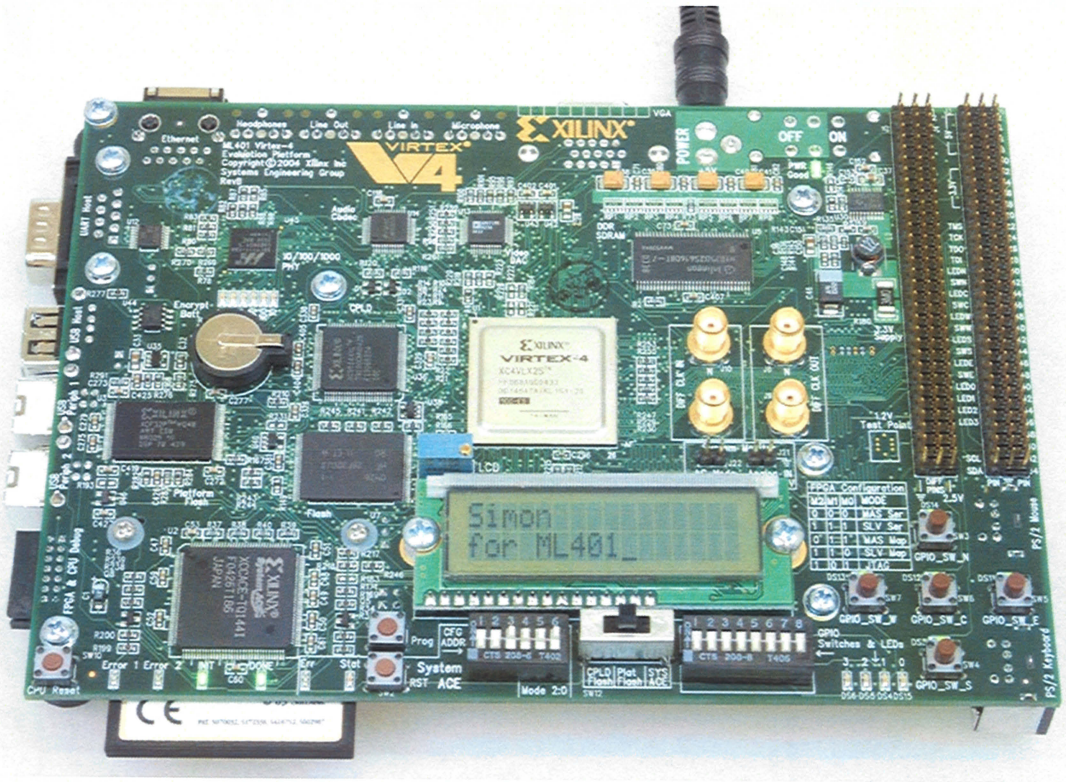

Figure 3.10: The Xilinx ml401 Virtex4 Evaluation Platform

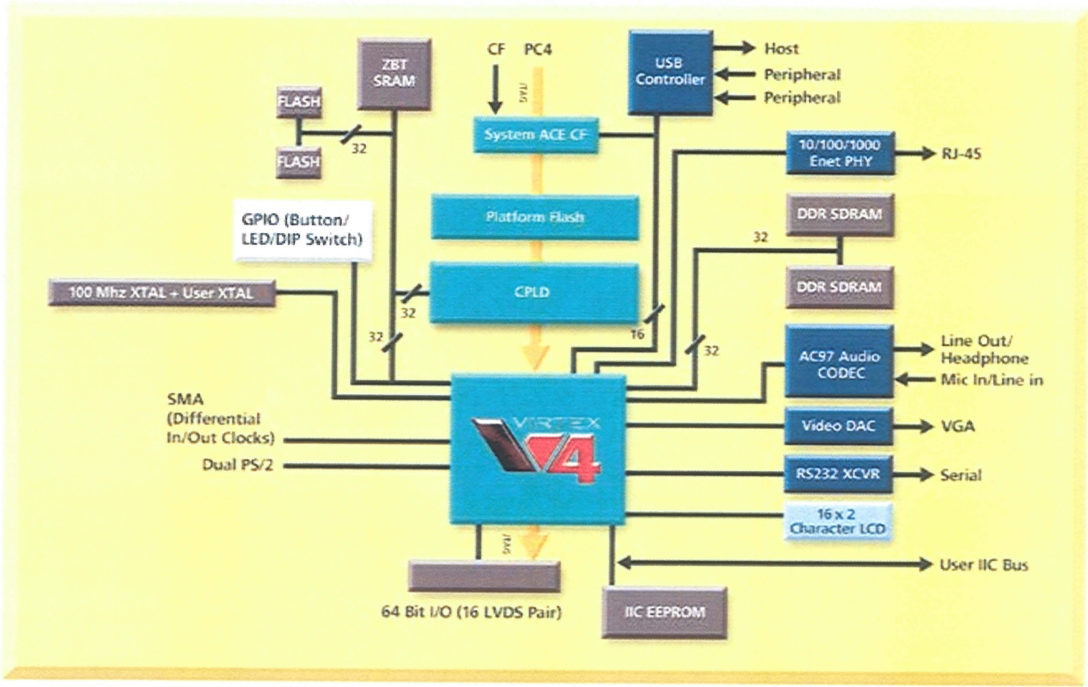

Figure 3.11: Block diagram of the Xilinx ml401 Virtex4 Evaluation Platform

ImpulseC was used to speed hardware and software development [14]. ImpulseC allows the use of a modified subset of C without the encumbrance of timing directives to describe hardware and provides interfaces to link that hardware to related software processes. Functional units are divided into modules and a set of high-level mechanisms are provided via an Application Programmer's Interface (API). At compile time these modules can be assigned to either hardware or software, and the ImpulseC compiler creates the high-level communications within the context of your target platform via a vendor-provided support package tailored to that platform.

With the ImpulseC and Xilinx hardware and software tools, a testbed was built to evaluate both the effectiveness and cost of the loop oriented path profiling algorithm. The next chapter deals with the pure software and hardware-assisted implementations of that algorithm and the observed performance of each.

CHAPTER 4

EXPERIMENTAL SETUP AND RESULTS

To assess the price of performing this tracing in pure software, a version of Logit was ported to the Microblaze platform followed by two hardware-assisted implementations. The first hardware-assisted implementation was built with pure VHDL but was abandoned due to the difficulty of transcoding a high-level algorithm needing significant data structure manipulations into hardware. The second hardware-assisted implementation and the pure software implementation used the ImpulseC hardware/software co-design environment [14] to speed development. In all, this algorithm has been implemented four times in whole or in part.

The Xilinx Microblaze soft-core processor [23], the IBM On-chip Peripheral Bus (OPB) [24] and various Xilinx IP such as memory controllers, a UART, and a timer were implemented on top of the ml401 board (See Figure 4.1). The 64kB on-chip block RAM (BRAM) was used along with the OPB-connected 1MB ZBT SRAM to run the instrumented program. The heap and stack segments were relocated to the SRAM, while the text and other segments ran out of the block RAM. This relocation was helpful since the exact state of the block RAM is programmed along with the FPGA configuration making the FPGA configuration and program execution occur in the same step.

Unfortunately, the SRAM could not be connected to the same bus as the BRAM (the Xilinx Local Memory Bus – LMB) because no LMB-based SRAM controller exists. Lastly, the 64MB DDR-RAM was reserved exclusively for the path profiles generated by the software and hardware loggers.

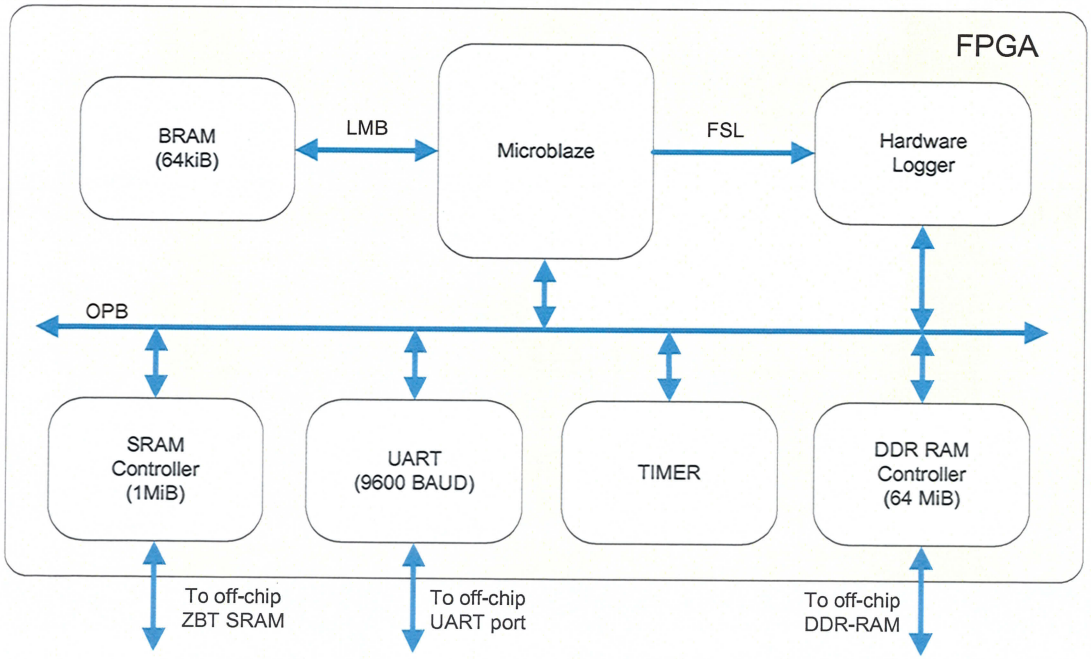

Figure 4.1: Block diagram of the Microblaze system with the Hardware logger included

To accomplish the Logit instrumentation, an instrumented program was encapsulated within the necessary ImpulseC syntax and each point inside of a loop was instrumented with macros (See Figure 4.2) to perform the path profiling. When using software logging, the macros reference a function that accepts event codes and payload data in a manner identical to the hardware: therefore, the software logging happened in sequence with the executing process. The software runs the loop-oriented path profile

algorithm and logs the profiles to the DDR-RAM via the Microblaze's OPB bus interface.

```
#ifdef SOFTWARE_LOGGER
#define BB(x) logit( (BASICBLOCK<<28) |x )
#define LS() logit(LOOPSTART<<28)
#define LC() logit(LOOPCONT<<28)
#define LE() logit(LOOPEND<<28)
#else
#define BB(x) { \
         uint32 data; \
         data = (BASICBLOCK<<28) | x; \
         WRITE_STREAM(producer_proc,trace_input_stream,data,sizeof(co_uint32));}

#define LS() {\
         uint32 data; \
         data = LOOPSTART<<28; \
         WRITE_STREAM(producer_proc,trace_input_stream,data,sizeof(co_uint32));}

#define LC() {\
         uint32 data; \
         data = LOOPCONT<<28; \
         WRITE_STREAM(producer_proc,trace_input_stream,data,sizeof(co_uint32));}

#define LE() {\
         uint32 data; \
         data = LOOPEND<<28; \
         WRITE_STREAM(producer_proc,trace_input_stream,data,sizeof(co_uint32));}
#endif //SOFTWARE_LOGGER
```

Figure 4.2: Macros used to implement the Microblaze Logit instrumentation

When using hardware-assisted logging, the macros resolve to three Microblaze instructions (with GCC level 2 optimization, 5 instructions with no optimization) that hand off code-words to the hardware via the FSL link. The hardware element, also written in ImpulseC, collected and compressed the trace events and wrote them out to the DDR-RAM via the OPB bus interface. After execution of the instrumented program in both hardware-assisted and software-only logging, the memory element was flushed out of a serial line by the Microblaze processor to a recording program on a PC for analysis. Then a comparison was made between the uninstrumented program, the software

instrumented program, and the hardware instrumented program in terms of total execution time according to the time returned by the OPB timer peripheral.

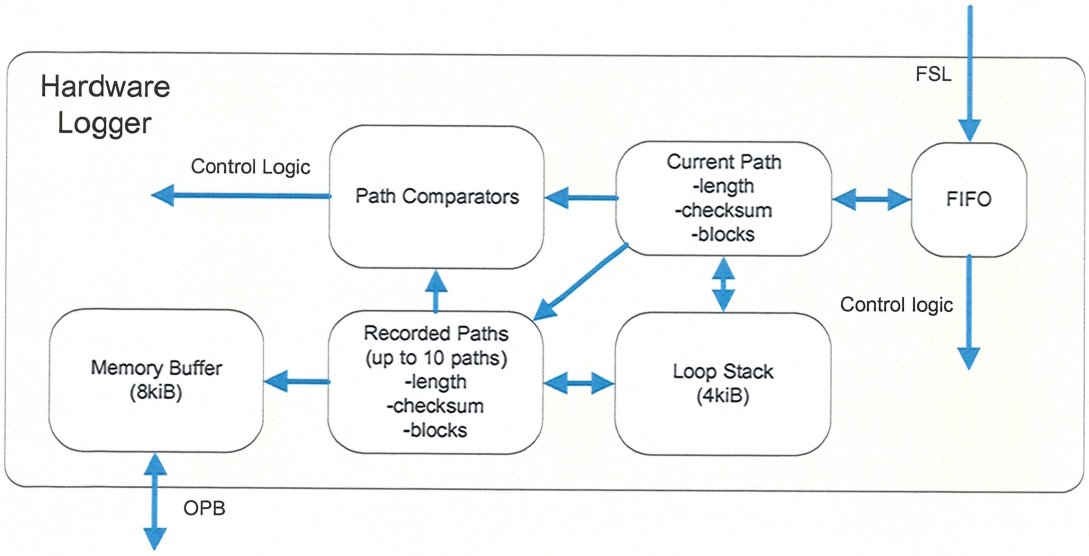

Figure 4.3: A notional data-path for the Impulsec-implemented witness processing hardware

Figure 4.3 shows a notional data-path for the hardware logger. Since ImpulseC has the ability to vary its pipeline design for a given C description the data-path is not exact and can change based on compile-time variables and pragmas inserted into the C description. The ImpulseC "stream" mechanism is really just a programmable-depth FIFO that can use the FSL for connectivity. When any loop event (LS, LC, or LE) is received, it affects control logic while any basic blocks are immediately added to the current path and added to the checksum if the maximum path length has not been exceeded. At a LOOP START event, the current and recorded paths are pushed onto the loop stack and the Current Path data structure is re-initialized. At a LOOP CONTINUE event, the Current Path is fed into the Path Comparators with all of the Recorded Paths and either a recorded path is matched and incremented or the Current Path is stored in the

Recorded Paths. At a LOOP END event, the Recorded Paths are sent to the Memory Buffer for asynchronous offloading to the DDR-RAM memory.

Some representative programs were hand-instrumented (using the above referenced macros) just for this experiment. These were drawn from ImpulseC examples in addition to a test program meant to verify correct operation of the logger. The first is a DES implementation extracted from the software portion of a 3DES hardware/software comparison example. The second is a simple Bubblesort example meant to show how even an inefficient algorithm can perform well in hardware. The third is a Fast Fourier Transform hardware implementation meant to show how well the FFT can be parallelized in hardware. These last two examples were extracted from their "hardware" implementations with no changes except for merging the setup code from their corresponding software-only source files. The last program tested was a framework CFG program that does no real work, but serves as a way to generate instrumentation events to verify the functional envelope of the logger.

4.1 Experimentation

Transforming the ImpulseC hardware and software sources into a functioning microprocessor system on an FPGA requires two toolsets, Impulse Accelerated Technologies ImpulseC CoDeveloper and Xilinx Platform Studio. The workflow for using these two tools is shown in Figure 4.4. CoDeveloper allows you to simulate, compile, and export ImpulseC hardware/software systems. The simulator uses a native compiler to treat the hardware elements as separate processing threads and compiles both the simulated hardware and pure software into a single application that can be run and monitored through a separate application giving you the ability to include debug

statements in the hardware modules. The compilation turns the hardware modules into an HDL-description tailored to a particular synthesis platform (we use the Xilinx VHDL platform). The final step integrates the hardware and software with the synthesis tool.

After CoDeveloper exports that ImpulseC software and the corresponding hardware elements we use Xilinx Platform Studio to synthesize and download to the board. First, we start with a reference platform specific to the ml401 board and add in the software application and then connect in the hardware component. Next, XPS compiles the hardware descriptions for the full system from VHDL down into netlists, then places and routes them into the Virtex4 LX25 FPGA. After that, XPS compiles the platform support libraries auto-generated to match the hardware system and then compiles the software application and uses the resulting object code to initialize the block RAM areas of the hardware programming file. Lastly, XPS downloads the programming file to the ml401 board where it begins execution immediately after the programming completes. A terminal program capable of talking to a serial port is used to view and record the output from the program. This same process was repeated for each test case for each program.

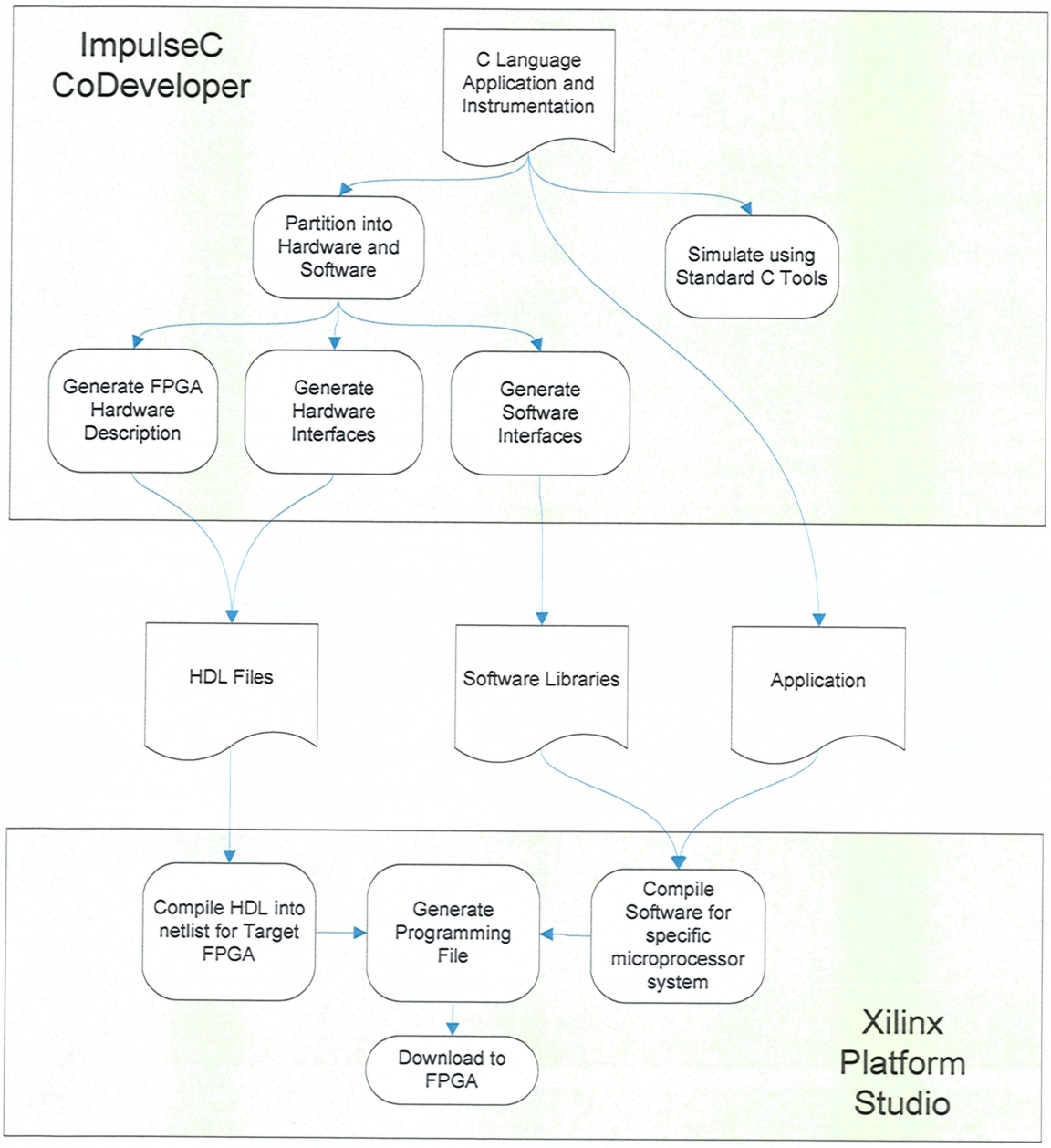

Figure 4.4: Complete Workflow for implementing an ImpulseC hardware/software system on a Xilinx FPGA (derived from [14])

4.2 Results

In order to satisfy the principal thesis that a hardware-assisted path profiler could be implemented with a lower overhead than a pure software path profiler, the hardware-assisted and pure software path profilers were run against the four programs

under test. The Xilinx OPB Timer peripheral [25] and supporting library was used to measure clock ticks and thus, determine elapsed times. The original program was also run without any instrumentation to establish a baseline performance for comparison. The results are shown in Table 4.1.

Table 4.1: Results of hardware-assisted and software-only instrumentation

Program	logger	total time (clock ticks)	extra cycles (clock ticks)	%additional	Ratio
DES	original	34928			
	software	422826	387898	1110.56%	12.11
	hardware	63424	28496	81.58%	1.82
Bubblesort	original	20047			
	software	565513	545466	2720.94%	28.21
	hardware	81603	61556	307.06%	4.07
FFT	original	438580			
	software	688140	249560	56.90%	1.57
	hardware	448380	9800	2.23%	1.02
Test program	original	15471			
	software	579063	563592	3642.89%	37.43
	hardware	139149	123678	799.42%	8.99

For each of the four programs under test there were three runs: one with the NO_LOGGER defined causing the instrumentation macros to be empty, one with the SOFTWARE_LOGGER macro defined causing the instrumentation macros to call the software logging function, or with neither of these defined causing the macro to pipe the witnesses to the hardware logger. As shown above, there is always some overhead for performing the instrumentation, though the amount varies widely depending on the program.

The worst performance is seen from the test program which is understandable since no "real" work is performed by that program (see Figure 4.5). Since our instrumentation adds instructions to the program, our overall performance is tied to the proportion of instrumentation instructions to real instructions. Since instrumentation placement is dependent on the control-flow instructions within the program, programs with complex control-flow graphs but very small proportional amounts of arithmetic or memory accesses will experience the highest overheads. In other words, programs with very small basic blocks will have the highest overheads from this kind of instrumentation.

```
LS();

for(i = 0; i <5; i++)
{
        BB(i);
}

LS();

for(j = 0; j < 60; j++)
{
        for(i=64+j%12; i <73+j%12; i++)
        {
                if(!(i%9))
                {
                        LS();
                        BB(128);
                        BB(129);
                        LC();
                        LE();
                }
                BB(i);
        }
        LC();
}
LE();

BB(i);

LC();
LE();
```

Figure 4.5: Source code for Test Program

Still, the hardware-assisted overhead numbers for the Test Program case are extremely high compared with the other three (37:1 and 9:1 for the pure software and hardware-assisted runs respectively) and a root cause analysis of the Test Program's results is needed. It was determined that all of the software had been compiled with GCC optimization level 2. To explore the effect of GCC optimization on the pure software and hardware-assisted path profilers, another set of runs was performed with all optimization turned off (GCC option '-O0'. Those results are shown in Table 4.2.

Table 4.2: Unoptimized timings (all software compiled with GCC optimization level 0)

Program	logger	total time (clock ticks)	extra cycles (clock ticks)	%additional	Ratio
DES	original	93341			
	software	1180735	1087394	1164.97%	12.65
	hardware	121821	28480	30.51%	1.31
Bubblesort	original	101585			
	software	1016001	914416	900.15%	10.00
	hardware	144001	42416	41.75%	1.42
FFT	original	999076			
	software	1781405	782329	78.31%	1.78
	hardware	1049077	50001	5.00%	1.05
Test Program	original	143344			
	software	1180735	1037391	723.71%	8.24
	hardware	273104	129760	90.52%	1.91

Compared to the O2 optimized case, these ratios look more consistent. Generally, the software logger is about 10 times slower than the original program and the hardware-assisted logger is up to twice as slow. Specifically, the overheads seen in the Test Program are now roughly in line with the DES and Bubblesort times. This would seem to indicate that the optimizer did indeed cull out at least some of the control-flow

logic in the uninstrumented program which means the comparison between the optimized instrumented and uninstrumented Test Program are invalid. Figure 4.6 shows the optimized assembly and the equivalent C representation for the Test Program which shows that most of the program has been optimized away. Apparently, the only thing that kept this code from being wiped out was the complex loop control logic in the inner loop.

```
addk    r22, r0, r0
addik   r22, r22, 1
addk    r5, r22, r0
brlid   r15, 144

addik   r6, r0, 12
addik   r4, r3, 73
addik   r3, r3, 64
cmp     r18, r4, r3
bltid   r18, -4
addik   r3, r3, 1
addik   r18, r0, 59
cmp     r18, r22, r18
bgeid   r18, -44

addik   r3, r3, -1
```

```
for(j = 0; j < 60; j++)
        for(i=64+j%12; I < 73+j%12; i++);
```

Figure 4.6: The actual assembly and equivalent C-language remains of the Test Program after running the O2 optimizer

The FFT results seem out of line in comparison with the other 3 programs and require a separate root cause analysis. The logger performance for both pure software and hardware-assisted logging improved between the unoptimized and optimized cases. How did the optimizer affect the instrumentation instructions for the hardware-assisted case since the hardware logger underneath did not change? Table 4.3 show the difference between the actual instrumentation instructions for each type of loop and basic-block event in both the unoptimized and O2 optimized cases. The instructions below are similar for each event and are as follows [23]:

- **imm IMM** – loads the 32-bit immediate value IMM into register r0
- **addik rD, rA, IMM** – Add a 16-bit immediate (IMM) to register rA and store in register rD without changing the Carry flag
- **put rA, FSLx** – write value from register rA into the Fast Simplex Link interface FSLx

as well as two instructions found only in the unoptimized case:

- **swi rD, rA, IMM** – store a word from register rA into the memory location computed by adding the contents of register rD and the immediate IMM.
- **lwi rD, rA, IMM** – load a word into register rD from the memory location computed by adding register rA with the immediate IMM.

The optimizer has therefore removed the load and store instructions that appear to be just saving the computed witness to the stack frame and then reloading it before writing the witness to the FSL destined for the hardware logger.

Table 4.3: Comparison of hardware-assisted instrumentation instructions in Microblaze assembly

	Unoptimized	Optimized

BB	imm 12288
	addik r3, r0, 1 // 0x30000001
	swi r3, r19, 28
	lwi r3, r19, 28
	put r3, rfsl0
		imm 12288
		addik r3, r0, 1 // 0x30000001
		put r3, rfsl0

LS	imm 16384
	addik r3, r0, 0 // 0x40000000
	swi r3, r19, 28
	lwi r3, r19, 28
	put r3, rfsl0
		imm 16384
		addik r4, r0, 0 // 0x40000000
		put r4, rfsl0

LC	imm 20480
	addik r3, r0, 0 // 0x50000000
	swi r3, r19, 1916
	lwi r3, r19, 1916
	put r3, rfsl0
		imm 20480
		addik r3, r0, 0 // 0x50000000
		put r3, rfsl0

LE	imm 24576
	addik r3, r0, 0 // 0x60000000
	swi r3, r19, 1916
	lwi r3, r19, 1916
	put r3, rfsl0
		imm 24576
		addik r3, r0, 0 // 0x60000000
		put r3, rfsl0

So the optimizer also affects the hardware-assisted instrumentation code making it run faster in addition to improving the program being instrumented. A look at this FFT implementation shows two non-nested loops with long paths and large basic blocks (see Appendix A). This code does not have many loop events and they don't happen very often, which explains the low overhead of instrumenting this program.

The last area to consider is whether or not the hardware caused the additional delay seen by the other programs. To test this idea a separate set of runs was performed where the hardware had been changed to only read and discard the witnesses sent to it. As shown in Table 4.4, the results were identical to the real hardware instrumentation

down to the cycle; therefore, all of the overhead observed is due to the instrumentation instructions inserted into the program.

Table 4.4: The unoptimized comparison of the timings of the original programs, the hardware-assisted path profilers, and the dummy hardware-assisted profilers

Program	logger	total time (clock ticks)	extra cycles (clock ticks)	%additional	Ratio
DES	original	93341			
	real hw	121821	28480	30.51%	1.31
	dummy hw	121821	28480	30.51%	1.31
Bubblesort	original	101585			
	real hw	144001	42416	41.75%	1.42
	dummy hw	144001	42416	41.75%	1.42
FFT	original	999076			
	real hw	1049077	50001	5.00%	1.05
	dummy hw	1049077	50001	5.00%	1.05
Test Program	original	143344			
	real hw	273104	129760	90.52%	1.91
	dummy hw	273104	129760	90.52%	1.91

CHAPTER 5

SUMMARY, CONCLUSIONS, AND FUTURE WORK

This thesis proposed a new approach to generating path profiles based around loop structure and theorized that a hardware implementation was both possible and would have lower overhead than a software-only version. Based on previous work under DSTEP, we implemented this algorithm in both hardware and software and performed experimentation against some sample programs to verify functionality and timing. We found that the observed performance of the software instrumentation depends significantly on the optimization stages within the compiler. We also observed that the overhead depended upon the program structure. The best case for our profiling involved long basic blocks while the worst case occurred using our Test Program, which had very short basic blocks. This experimentation showed that the hardware-assisted path profiler was lower overhead than the pure software path profiler. Moreover, we were able to show that the hardware never blocked during profiling, meaning that all of the overhead was due to the additional software instructions needed for instrumentation.

Our conclusion is that loop oriented path profiling is a good alternative to basic block profiling and that hardware-assisted profiling has much lower overhead than a pure software approach. We also saw that ImpulseC provided a good environment for developing such a hardware-assisted profiling engine.

In the future, a study of the impact of this kind of hardware logger on bus traffic should be made. Also, the impact of loop-oriented instrumentation instructions on cache and pipeline performance would be helpful. This research allows a view into the future of hardware performance monitoring. Further research into techniques for performing this instrumentation with fewer or without any software changes is needed. A method similar to Vaswani et al. [19] could be used to modify a soft processor such as the Microblaze to emit a branch-taken bit and address to a dedicated path-profiler. Such a system would not require the instrumentation of any programs, but would have to find some novel mechanism to offload the profiles generated (most likely a non-maskable interrupt and interrupt-handler combination).

The original loop oriented path profiling approach sought to attach current performance monitoring hardware metrics to basic blocks or paths. Further research would first have to either modify the Microblaze to have performance monitoring hardware or make use of another soft processor already containing such features. Subsequently, the witnesses generated to the hardware path profiler could also contain metrics that could be recorded at the path level. The availability of high accuracy processor metrics localized to specific paths without the perturbation introduced by software-only loggers would be very interesting to hardware and software designers alike.

APPENDICES

APPENDIX A

Sources for Programs Under Test

A.1 DES

```
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include "co.h"
#include "pathlogger.h"

#if defined(MICROBLAZE)
#include "xparameters.h"
#define printf xil_printf
#else
// include sim file when not inside of xps
#include "cosim_log.h"
#endif
/* Portable C code to create DES key schedules from user-provided keys
 * This doesn't have to be fast unless you're cracking keys or UNIX
 * passwords
 */

/* Key schedule-related tables from FIPS-46 */

/* permuted choice table (key) */
static unsigned char pc1[] = {
    57, 49, 41, 33, 25, 17,  9,
     1, 58, 50, 42, 34, 26, 18,
    10,  2, 59, 51, 43, 35, 27,
    19, 11,  3, 60, 52, 44, 36,

    63, 55, 47, 39, 31, 23, 15,
     7, 62, 54, 46, 38, 30, 22,
    14,  6, 61, 53, 45, 37, 29,
    21, 13,  5, 28, 20, 12,  4
};

/* number left rotations of pc1 */
static unsigned char totrot[] = {
    1,2,4,6,8,10,12,14,15,17,19,21,23,25,27,28
};

/* permuted choice key (table) */
```

```
static unsigned char pc2[] = {
    14, 17, 11, 24,  1,  5,
     3, 28, 15,  6, 21, 10,
    23, 19, 12,  4, 26,  8,
    16,  7, 27, 20, 13,  2,
    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32
};

/* End of DES-defined tables */


/* bit 0 is left-most in byte */
static int bytebit[] = {
    0200,0100,040,020,010,04,02,01
};

// The producer process generates some data and initializes memory.
//
void producer(co_stream trace_input_stream)
{
        int32 i, j, data;
        IF_SIM(cosim_logwindow debug;)
        IF_SIM(debug = cosim_logwindow_create("producer");)
        TIME_EVENT("Producer: start");
#ifdef IMPULSE_C_TARGET
    HW_STREAM_OPEN(producer_proc,trace_input_stream, O_WRONLY, UINT_TYPE(32));
#else
    co_stream_open(trace_input_stream, O_WRONLY, UINT_TYPE(32));
#endif
// Insert test case here
        {
                LS();
                BB(1);
                unsigned char key[8];
                int decrypt = 0;
                unsigned long k[16][2];
                static int Asmversion = 0;
                unsigned char pc1m[56];     /* place to modify pc1 into */
        unsigned char pcr[56];      /* place to rotate pc1 into */
        register int i,j,l;
            int m;
                unsigned char ks[8];
                LS();
        for (j=0; j<56; j++) {      /* convert pc1 to bits of key */
                        BB(2);
                l=pc1[j]-1;     /* integer bit location  */
                m = l & 07;     /* find bit      */
                pc1m[j]=(key[l>>3] &    /* find which key byte l is in */
                bytebit[m]) /* and which bit of that byte */
                ? 1 : 0;    /* and store 1-bit result */
                        LC();
        }
                LE();
                LS();
        for (i=0; i<16; i++) {      /* key chunk for each iteration */
                        BB(3);
                memset(ks,0,sizeof(ks));    /* Clear key schedule */
                        LS();
                for (j=0; j<56; j++)    /* rotate pc1 the right amount */
                        {
```

```
                                BB(4);
                    pcr[j] = pc1m[(l=j+totrot[decrypt? 15-i : i])<(j<28? 28 : 56) ?
l: l-28];
                                LC();
                            }
                            LE();
                            LS();
                /* rotate left and right halves independently */
                    for (j=0; j<48; j++){   /* select bits individually */
                                BB(5);
                    /* check bit that goes to ks[j] */
                    if (pcr[pc2[j]-1]){
                                        BB(6);
                            /* mask it in if it's there */
                            l= j % 6;
                            ks[j/6] |= bytebit[l] >> 2;
                    }
                                LC();
                    }
                            LE();
                            BB(7);
                    /* Now convert to packed odd/even interleaved form */
                    k[i][0] = ((long)ks[0] << 24)
                            | ((long)ks[2] << 16)
                     | ((long)ks[4] << 8)
                     | ((long)ks[6]);
                    k[i][1] = ((long)ks[1] << 24)
                     | ((long)ks[3] << 16)
                     | ((long)ks[5] << 8)
                    | ((long)ks[7]);
                    if(Asmversion){
                                BB(8);
                        /* The assembler versions pre-shift each subkey 2 bits
                         * so the Spbox indexes are already computed
                         */
                        k[i][0] <<= 2;
                        k[i][1] <<= 2;
                    }
                            LC();
                }
                    LE();
                    LC();
                    LE();
        }

#ifdef IMPULSE_C_TARGET
    HW_STREAM_CLOSE(producer_proc,trace_input_stream);
#else
    co_stream_close(trace_input_stream);
#endif

        TIME_EVENT("Producer: end");
}
```

A.2 Bubblesort

```
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include "co.h"
#include "pathlogger.h"

#if defined(MICROBLAZE)
#include "xparameters.h"
#define printf xil_printf
#else
// include sim file when not inside of xps
#include "cosim_log.h"
#endif

#define N 50

// The producer process generates some data
// and initializes memory.
//
void producer(co_stream trace_input_stream)
{
        int32 i, j, data;
        IF_SIM(cosim_logwindow debug;)
        IF_SIM(debug = cosim_logwindow_create("producer");)
        TIME_EVENT("Producer: start");
#ifdef IMPULSE_C_TARGET
    HW_STREAM_OPEN(producer_proc,trace_input_stream, O_WRONLY, UINT_TYPE(32));
#else
     co_stream_open(trace_input_stream, O_WRONLY, UINT_TYPE(32));
#endif

        LS();
        {
                BB(0);
                int streamValue;
                int tmp,k,i,top;
                int data[N];

                printf("Unsorted path: ");
                LS();
                for(i = 0; i < N; i++)
                {
                        BB(1);
                        data[i] = 100-i;
                        printf("%d,", data[i]);
                        LC();
                }
                LE();
                BB(2);
                printf("\n");

                //sort data here
                top = i;
                k = 0;
                tmp = 0;

                LS();
                while ( top >= 1 ) {
                        BB(3);
```

```
                        k=0;
                        LS();
                        while ( k < top ) {
                                BB(4);
                                if ( data[k] > data[k+1] ) {
                                        BB(5);
                                        tmp = data[k];
                                        data[k] = data[k+1];
                                        data[k+1] = tmp;
                                }
                                BB(6);
                                k=k+1;
                                LC();
                        }
                        LE();
                        BB(7);

//current highest number is known and sent to consumer
                        top=top-1;
                        LC();
                }
                LE();

                BB(8);
                printf("Sorted path: ");
                LS();
                for(i = 0; i < N; i++)
                {
                        BB(9);
                        printf("%d,", data[i]);
                        LC();
                }
                LE();
                BB(10);
                printf("\n");
        }
        LC();
        LE();

#ifdef IMPULSE_C_TARGET
    HW_STREAM_CLOSE(producer_proc,trace_input_stream);
#else
    co_stream_close(trace_input_stream);
#endif

        TIME_EVENT("Producer: end");
}
```

A.3 FFT

```
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include "co.h"
#include "pathlogger.h"

#if defined(MICROBLAZE)
#include "xparameters.h"
#define printf xil_printf
#else
// include sim file when not inside of xps
#include "cosim_log.h"
#endif

#define NUM_PTS 256

// define a complex number type
typedef struct cpx_s {
        co_int16 r,i;
} cpx_t;

// macros for packed complex numbers (32 bit unsigned ints).
#define C_PACK(c) ((((co_uint32)((c).r))<<16)|((co_uint32)((c).i)&0xffff))
#define C_REAL(u) ((co_int16)((u)>>16))
#define C_IMAG(u) ((co_int16)(u))

// number of fractional bits in fixed-point representation
#define FRACBITS 15

// scale down a by 2^DW with rounding
#define FXROUND32(a,DW) (((int32)(a)+((int32)1<<((DW)-1)))>>(DW))
#define FXROUND16(a,DW) (((int16)(a)+((int16)1<<((DW)-1)))>>(DW))

#define getBank(i) ((i^(i>>2)^(i>>4)^(i>>6))&0x3)

//
// select is used to assign res the correct value v0,..,v3 for the selected
bank
//
#define select(res,mem,bank0,v0,bank1,v1,bank2,v2,bank3,v3)\
                        if (bank0==mem)\
                                res=v0;\
                        else if (bank1==mem)\
                                res=v1;\
                        else if (bank2==mem)\
                                res=v2;\
                        else\
                                res=v3

// The producer process generates some data and initializes memory.
//
void producer(co_stream trace_input_stream)
{
        IF_SIM(cosim_logwindow debug;)
        IF_SIM(debug = cosim_logwindow_create("producer");)
        TIME_EVENT("Producer: start");
#ifdef IMPULSE_C_TARGET
    HW_STREAM_OPEN(producer_proc,trace_input_stream, O_WRONLY, UINT_TYPE(32));
#else
```

```
    co_stream_open(trace_input_stream, O_WRONLY, UINT_TYPE(32));
#endif

        LS();

        //Insert program here
        BB(1);
        // Each array is implemented in its own memory bank.  By splitting the
data into multiple
        // arrays, we are able to access many values in one cycle.
        co_uint32
twiddles1[64]={0x7fff0000,0x7ff5fcdc,0x7fd8f9b8,0x7fa6f696,0x7f61f374,0x7f09f05
5,0x7e9ced38,0x7e1dea1e,0x7d89e707,0x7ce3e3f5,0x7c29e0e6,0x7b5cdddd,0x7a7cdad8,
0x7989d7da,0x7884d4e1,0x776bd1ef,0x7641cf05,0x7504cc21,0x73b5c946,0x7254c674,0x
70e2c3aa,0x6f5ec0e9,0x6dc9be32,0x6c23bb86,0x6a6db8e4,0x68a6b64c,0x66cfb3c1,0x64
e8b141,0x62f1aecd,0x60ebac65,0x5ed7aa0b,0x5cb3a7be,0x5a82a57e,0x5842a34d,0x55f5
a129,0x539b9f15,0x51339d0f,0x4ebf9b18,0x4c3f9931,0x49b4975a,0x471c9593,0x447a93
dd,0x41ce9237,0x3f1790a2,0x3c568f1e,0x398c8dac,0x36ba8c4b,0x33df8afc,0x30fb89bf
,0x2e118895,0x2b1f877c,0x28268677,0x25288584,0x222384a4,0x1f1a83d7,0x1c0b831d,0
x18f98277,0x15e281e3,0x12c88164,0x0fab80f7,0x0c8c809f,0x096a805a,0x06488028,0x0
324800b};
        co_uint32
twiddles2[64]={0x7fff0000,0x7fd8f9b8,0x7f61f374,0x7e9ced38,0x7d89e707,0x7c29e0e
6,0x7a7cdad8,0x7884d4e1,0x7641cf05,0x73b5c946,0x70e2c3aa,0x6dc9be32,0x6a6db8e4,
0x66cfb3c1,0x62f1aecd,0x5ed7aa0b,0x5a82a57e,0x55f5a129,0x51339d0f,0x4c3f9931,0x
471c9593,0x41ce9237,0x3c568f1e,0x36ba8c4b,0x30fb89bf,0x2b1f877c,0x25288584,0x1f
1a83d7,0x18f98277,0x12c88164,0x0c8c809f,0x06488028,0x00008001,0xf9b88028,0xf374
809f,0xed388164,0xe7078277,0xe0e683d7,0xdad88584,0xd4e1877c,0xcf0589bf,0xc9468c
4b,0xc3aa8f1e,0xbe329237,0xb8e49593,0xb3c19931,0xaecd9d0f,0xaa0ba129,0xa57ea57e
,0xa129aa0b,0x9d0faecd,0x9931b3c1,0x9593b8e4,0x9237be32,0x8f1ec3aa,0x8c4bc946,0
x89bfcf05,0x877cd4e1,0x8584dad8,0x83d7e0e6,0x8277e707,0x8164ed38,0x809ff374,0x8
028f9b8};
        co_uint32
twiddles3[64]={0x7fff0000,0x7fa6f696,0x7e9ced38,0x7ce3e3f5,0x7a7cdad8,0x776bd1e
f,0x73b5c946,0x6f5ec0e9,0x6a6db8e4,0x64e8b141,0x5ed7aa0b,0x5842a34d,0x51339d0f,
0x49b4975a,0x41ce9237,0x398c8dac,0x30fb89bf,0x28268677,0x1f1a83d7,0x15e281e3,0x
0c8c809f,0x0324800b,0xf9b88028,0xf05580f7,0xe7078277,0xdddd84a4,0xd4e1877c,0xcc
218afc,0xc3aa8f1e,0xbb8693dd,0xb3c19931,0xac659f15,0xa57ea57e,0x9f15ac65,0x9931
b3c1,0x93ddbb86,0x8f1ec3aa,0x8afccc21,0x877cd4e1,0x84a4dddd,0x8277e707,0x80f7f0
55,0x8028f9b8,0x800b0324,0x809f0c8c,0x81e315e2,0x83d71f1a,0x86772826,0x89bf30fb
,0x8dac398c,0x923741ce,0x975a49b4,0x9d0f5133,0xa34d5842,0xaa0b5ed7,0xb14164e8,0
xb8e46a6d,0xc0e96f5e,0xc94673b5,0xd1ef776b,0xdad87a7c,0xe3f57ce3,0xed387e9c,0xf
6967fa6};
        co_uint32 data0[NUM_PTS/4];
        co_uint32 data1[NUM_PTS/4];
        co_uint32 data2[NUM_PTS/4];
        co_uint32 data3[NUM_PTS/4];
        cpx_t mem0,mem1,mem2,mem3;

        co_uint8 i,m,j,j0,j1,j2,j3,inc,t,t0,tinc;
        co_uint16 axis;

        cpx_t d;
        co_uint32 u,ii;
        cpx_t p1,p2,p3;
        cpx_t c0,c1,c2,c3;
        cpx_t d0,d1,d2,d3;

        co_uint2 bank0,bank1,bank2,bank3;
        co_uint8 loc0,loc1,loc2,loc3;

        {
                /* reorder data as it comes in for inplace fft computation */
```

```
    //co_stream_open(result_stream, O_WRONLY, INT_TYPE(32));
    //co_stream_open(sample_stream, O_RDONLY, INT_TYPE(32));
    LS();
    for(ii=0; ii<NUM_PTS; ii++) {
            BB(2);
            d.r=ii&0x1f;
            d.i=d.r;
            u=C_PACK(d);

            // this does the reordering
            j0=((i>>6)|((i>>2)&0x0c)|((i<<2)&0x30)|((i<<6)&0xc0));
            // store it in the assigned bank
            bank0=getBank(j0);
            if (bank0==0)
            {
                    BB(3);
                    data0[j0>>2]=u;
            }
            if (bank0==1)
            {
                    BB(4);
                    data1[j0>>2]=u;
            }
            if (bank0==2)
            {
                    BB(5);
                    data2[j0>>2]=u;
            }
            if (bank0==3)
            {
                    BB(6);
                    data3[j0>>2]=u;
            }
            LC();
    }
    LE();
    //
    // compute 4 stages of 64 butterflies in one 4x64 loop
    //
// By flattening the FFT algorithm into a single loop, we are able
to use the pipeline
    // option enabling us to compute 1 butterfly/cycle
    //
    BB(7);
// these parameters are used to compute the appropriate indexing
for the current stage
    m=1;
    j=0;
    axis=0x00ff;
    inc=4;
    t=0;
    tinc=0x40;
    i=0;
    LS();
    do {
            BB(8);
        // compute the addresses of the four samples required for
        this butterfly
            j0=j;
            j1=j+m;
            j2=j+(m<<1);
            j3=j2+m;
            t0=t;
```

```
                    // prep next butterfly
                    t=(t+tinc)&0x3f;
                    if ((i&(axis>>8))==(axis>>8))
                    {
                            BB(9);
                            j+=inc;
                    }
                    else
                    {
                            BB(10);
                            j++;
                    }
                    BB(11);
                    i++;
                    if ((i&0x3f)==0) {
                            BB(12);
                            // compute parameters for the next stage
                            tinc=tinc>>2;
                            m=m<<2;
                            axis=axis<<2;
                            inc=(m<<2)-(axis>>8);
                    }

                    //
                    // retrieve the four samples from memory
                    //
                    BB(13);
                    bank0=getBank(j0);
                    bank1=getBank(j1);
                    bank2=getBank(j2);
                    bank3=getBank(j3);

                // select the address for each array according to the bank
                assignments
                    select(loc0,0,bank0,j0,bank1,j1,bank2,j2,bank3,j3);
                    select(loc1,1,bank0,j0,bank1,j1,bank2,j2,bank3,j3);
                    select(loc2,2,bank0,j0,bank1,j1,bank2,j2,bank3,j3);
                    select(loc3,3,bank0,j0,bank1,j1,bank2,j2,bank3,j3);

                    // get the data
                    u=data0[loc0>>2];
                    mem0.r=C_REAL(u); mem0.i=C_IMAG(u);
                    u=data1[loc1>>2];
                    mem1.r=C_REAL(u); mem1.i=C_IMAG(u);
                    u=data2[loc2>>2];
                    mem2.r=C_REAL(u); mem2.i=C_IMAG(u);
                    u=data3[loc3>>2];
                    mem3.r=C_REAL(u); mem3.i=C_IMAG(u);

                // select the sample for each d0,..,d3 according to the
                bank assignments

select(d0,0,bank0,mem0,bank1,mem1,bank2,mem2,bank3,mem3);

select(d1,1,bank0,mem0,bank1,mem1,bank2,mem2,bank3,mem3);

select(d2,2,bank0,mem0,bank1,mem1,bank2,mem2,bank3,mem3);

select(d3,3,bank0,mem0,bank1,mem1,bank2,mem2,bank3,mem3);

                    // scale down data by dividing by 4
                    d0.r=FXROUND16(d0.r,2);
```

```
    d0.i=FXROUND16(d0.i,2);
    d1.r=FXROUND16(d1.r,2);
    d1.i=FXROUND16(d1.i,2);
    d2.r=FXROUND16(d2.r,2);
    d2.i=FXROUND16(d2.i,2);
    d3.r=FXROUND16(d3.r,2);
    d3.i=FXROUND16(d3.i,2);

    //
    // radix-4 butterfly
    //
    u=twiddles1[t0];
    c1.r=C_REAL(u); c1.i=C_IMAG(u);
    u=twiddles2[t0];
    c2.r=C_REAL(u); c2.i=C_IMAG(u);
    u=twiddles3[t0];
    c3.r=C_REAL(u); c3.i=C_IMAG(u);

p1.r=FXROUND32(IMUL32(c1.r,d1.r)-
               IMUL32(c1.i,d1.i),FRACBITS);
p1.i=FXROUND32(IMUL32(c1.r,d1.i)+
               IMUL32(c1.i,d1.r),FRACBITS);
p2.r=FXROUND32(IMUL32(c2.r,d2.r)-
               IMUL32(c2.i,d2.i),FRACBITS);
p2.i=FXROUND32(IMUL32(c2.r,d2.i)+
               IMUL32(c2.i,d2.r),FRACBITS);
p3.r=FXROUND32(IMUL32(c3.r,d3.r)-
               IMUL32(c3.i,d3.i),FRACBITS);
p3.i=FXROUND32(IMUL32(c3.r,d3.i)+IMUL32(c3.i,d3.r),FRACBITS
);

    c0.r=d0.r+p2.r; // c0=d0+twiddle2*d2
    c0.i=d0.i+p2.i;
    c1.r=p1.r+p3.r; // c1=twiddle1*d1+twiddle3*d3
    c1.i=p1.i+p3.i;
    c2.r=d0.r-p2.r; // c2=d0-twiddle2*d2
    c2.i=d0.i-p2.i;
    c3.r=p3.i-p1.i; // c3=j*(tiddle1*d1-twiddle3*d3)
    c3.i=p1.r-p3.r;

d0.r=c0.r+c1.r; //
d0=d0+twiddle2*d2+twiddle1*d1+twiddle3*d3
    d0.i=c0.i+c1.i;
    d1.r=c2.r-c3.r;
    d1.i=c2.i-c3.i;
    d2.r=c0.r-c1.r;
    d2.i=c0.i-c1.i;
    d3.r=c2.r+c3.r;
    d3.i=c2.i+c3.i;

// select the output data for each memory according to the
bank assignments
    select(mem0,0,bank0,d0,bank1,d1,bank2,d2,bank3,d3);
    select(mem1,1,bank0,d0,bank1,d1,bank2,d2,bank3,d3);
    select(mem2,2,bank0,d0,bank1,d1,bank2,d2,bank3,d3);
    select(mem3,3,bank0,d0,bank1,d1,bank2,d2,bank3,d3);

    // store the data
    data0[loc0>>2]=C_PACK(mem0);
    data1[loc1>>2]=C_PACK(mem1);
    data2[loc2>>2]=C_PACK(mem2);
    data3[loc3>>2]=C_PACK(mem3);
    LC();
```

```
                } while (i!=0);
                LE();
                // stream out the data
                BB(14);
                j0=0;
                /*do {
                        bank0=getBank(j0);
                        if (bank0==0)
                                u=data0[j0>>2];
                        if (bank0==1)
                                u=data1[j0>>2];
                        if (bank0==2)
                                u=data2[j0>>2];
                        if (bank0==3)
                                u=data3[j0>>2];
                        //co_stream_write(result_stream,&u,sizeof(co_uint32));
                        j0++;
                } while (j0!=0);*/

                //co_stream_close(sample_stream);
                //co_stream_close(result_stream);
                //IF_SIM(break;)

        }

        //end of program
        LC();
        LE();

#ifdef IMPULSE_C_TARGET
    HW_STREAM_CLOSE(producer_proc,trace_input_stream);
#else
    co_stream_close(trace_input_stream);
#endif

        TIME_EVENT("Producer: end");
}
```

A.4 Test Program

```
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include "co.h"
#include "pathlogger.h"

#if defined(MICROBLAZE)
#include "xparameters.h"
#define printf xil_printf
#else
// include sim file when not inside of xps
#include "cosim_log.h"
#endif

// The producer process generates some data and initializes memory.
//
void producer(co_stream trace_input_stream)
{
        int32 i, j, data;
        IF_SIM(cosim_logwindow debug;)
        IF_SIM(debug = cosim_logwindow_create("producer");)
        TIME_EVENT("Producer: start");
#ifdef IMPULSE_C_TARGET
    HW_STREAM_OPEN(producer_proc,trace_input_stream, O_WRONLY, UINT_TYPE(32));
#else
    co_stream_open(trace_input_stream, O_WRONLY, UINT_TYPE(32));
#endif

        LS();

        for(i = 0; i <5; i++)
        {
                BB(i);
        }

        LS();

        for(j = 0; j < 60; j++)
        {
                for(i=64+j%12; i <73+j%12; i++)
                {
// add some variance by skipping a block every-other run
                        if(!(i%9))
                        {
                                printf("subloop->");
                                LS();
                                BB(128);
                                BB(129);
                                LC();
                                LE();
                        }
                        printf("%d->", i);
                        BB(i);
                }
                printf("\n");
                LC();
        }
        LE();
```

```
        BB(i);

        LC();
        LE();

#ifdef IMPULSE_C_TARGET
    HW_STREAM_CLOSE(producer_proc,trace_input_stream);
#else
    co_stream_close(trace_input_stream);
#endif

        TIME_EVENT("Producer: end");
}
// vim:ts=4:cin!
```

APPENDIX B

Sources for Hardware and Software Instrumentation

The ImpulseC-based instrumentation is composed of 5 source files. The first is the configuration file, "config.c", which contains the overall setup necessary for every ImpulseC project.

B.1 config.c

```
#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "pathlogger.h"

// See pathlogger_sw.c
extern void producer(co_stream trace_input_stream);
extern void consumer(co_signal done, co_memory memblk);
// See pathlogger_hw.c
extern void memio_run(co_stream trace_output_stream, co_memory memblk,
co_signal done);
extern void logger_run(co_stream trace_input_stream, co_stream
trace_output_stream);

void config_pathlogger(void *arg)
{
        co_signal done_sig;
        co_memory external_memory;
        co_process producer_proc, consumer_proc;
        co_process logger_proc;
        co_process memio_proc;
        co_stream trace_input_stream;
        co_stream trace_output_stream;

        //start_sig = co_signal_create(NULL);
        IF_SIM(cosim_logwindow_init();)
        done_sig = co_signal_create("DoneSignal");
        external_memory = co_memory_create("external_mem", "ext0",
64*1024*1024);

      trace_input_stream = co_stream_create("trace_input_stream",
                                            UINT_TYPE(32), 16);
```

```
        trace_output_stream = co_stream_create("trace_output_stream",
                                               UINT_TYPE(32), 16);

        producer_proc = co_process_create("producer_proc",
(co_function)producer,
                                          1, trace_input_stream);
        logger_proc = co_process_create("logger_proc", (co_function)logger_run,
                                          2, trace_input_stream,
trace_output_stream);
        consumer_proc = co_process_create("consumer_proc",
(co_function)consumer,
                                          2,done_sig, external_memory);
        memio_proc = co_process_create("memio_proc", (co_function)memio_run,
                                  3, trace_output_stream, external_memory,
done_sig);

        // Assign logger_proc (logger_run) to hardware element PE0
        co_process_config(logger_proc, co_loc, "PE0");
        co_process_config(memio_proc, co_loc, "PE0");
}

co_architecture co_initialize()
{

return(co_architecture_create("pathlogger","xilinx_vhdl",config_pathlogger,NULL
));
}
```

B.2 pathlogger.h

The next file is the header file that the program under test references, "pathlogger.h". This file contains all of the macros used for instrumentation.

```
#ifndef PATHLOGGER_H
#define PATHLOGGER_H

#define FUNCTIONSTART           0x01
#define FUNCTIONEND             0x02
#define BASICBLOCK              0x03
#define LOOPSTART               0x04
#define LOOPCONT                0x05
#define LOOPEND                 0x06

#ifdef IMPULSE_C_TARGET
#define WRITE_STREAM(w,x,y,z) HW_STREAM_WRITE(w,x,y);
#else
#define WRITE_STREAM(w,x,y,z) co_stream_write(x,&y,z);
#endif

#ifdef NO_LOGGER
#define BB(x)
#define LS()
#define LC()
#define LE()
#elif SOFTWARE_LOGGER
#define BB(x) logit( (BASICBLOCK<<28) |x )
#define LS() logit(LOOPSTART<<28)
#define LC() logit(LOOPCONT<<28)
#define LE() logit(LOOPEND<<28)
```

```
#else
#define BB(x) { \
        uint32 data; \
        data = (BASICBLOCK<<28) | x; \
        WRITE_STREAM(producer_proc,trace_input_stream,data,sizeof(co_uint32));}

#define LS() {\
        uint32 data; \
        data = LOOPSTART<<28; \
        WRITE_STREAM(producer_proc,trace_input_stream,data,sizeof(co_uint32));}

#define LC() {\
        uint32 data; \
        data = LOOPCONT<<28; \
        WRITE_STREAM(producer_proc,trace_input_stream,data,sizeof(co_uint32));}

#define LE() {\
        uint32 data; \
        data = LOOPEND<<28; \
        WRITE_STREAM(producer_proc,trace_input_stream,data,sizeof(co_uint32));}
#endif //SOFTWARE_LOGGER

#define END_OF_LOOPTRACE 0x12345678

#ifdef IMPULSE_C_TARGET
#include "xtmrctr.h"
#ifdef DECLARE_LOGIT_TIMERS
XTmrCtr TimerCounter;
uint32 timer_counts[16];
char event_names[16][64];
int num_counts = 0;
#else
extern XTmrCtr TimerCounter;
extern uint32 timer_counts[16];
extern char event_names[16][64];
extern int num_counts;
#endif

#define INIT_TIMERS \
        printf("Initializing timers...\n\r"); \
        XTmrCtr_Initialize(&TimerCounter, XPAR_OPB_TIMER_1_DEVICE_ID); \
        XTmrCtr_SetResetValue(&TimerCounter,0,0); \
        XTmrCtr_Start(&TimerCounter,0);

#define TIME_EVENT(x) \
        timer_counts[num_counts]=XTmrCtr_GetValue(&TimerCounter,0); \
        strncpy(event_names[num_counts], x, 63); \
        num_counts++;

#define PRINT_TIMERS \
        { \
                int i; \
                printf("%d Timer values recorded\n\r", num_counts); \
                for(i = 0; i < num_counts; i++) \
                        printf("%ld %s\n", timer_counts[i], event_names[i]); \
        }
#else
#define INIT_TIMERS
#define TIME_EVENT(x)
#define PRINT_TIMERS
#endif

#endif //PATHLOGGER_H
```

B.3 pathlogger_hw.c

The next file is the first part of thehardware path profiler description, "pathlogger_hw.c". This is the main part of the hardware path profiler that takes in event witnesses and outputs path profile traces.

```
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "pathlogger.h"

// MAX_PATHLENGTH must be an even number to keep from breaking LOOPEND output
below
#define MAX_PATHLENGTH 10
#define MAX_LOOPDEPTH 10
#define MAX_PATHS_PER_LOOP 10

// Hardware process
void logger_run(co_stream trace_input_stream, co_stream trace_output_stream)
{
        short current_pathlength=0;
        short current_checksum=0;
        short current_blocks[MAX_PATHLENGTH];

        uint8 loop_depth = 0;

        // recorded paths and pathlengths for this level
        // copying this array of objects to and from stack memory will
        // probably account for the majority of the slowdown in the system
        uint8 num_paths = 0;
        short pathlengths[MAX_PATHS_PER_LOOP];
        short checksums[MAX_PATHS_PER_LOOP];
        long  pathcounts[MAX_PATHS_PER_LOOP];
        short blocks[MAX_PATHS_PER_LOOP][MAX_PATHLENGTH];
        long others;

        short stack[1024];
        short stack_ptr = 0;

        // Create Logger Window
        IF_SIM(cosim_logwindow debug;)
        IF_SIM(debug = cosim_logwindow_create("logger_run");)

        DUMP_STACK;

        co_stream_open(trace_input_stream, O_RDONLY, UINT_TYPE(32));
        co_stream_open(trace_output_stream, O_WRONLY, UINT_TYPE(32));

        while(!co_stream_eos(trace_input_stream))
        {
                co_uint32 data;
                uint8   event_code;
                uint16  payload;
                int8 i;
```

```
co_stream_read(trace_input_stream, &data, sizeof(uint32));

// split out the event code
event_code = (uint8)(data>>28);
payload = (uint16) data;

switch(event_code)
{
case BASICBLOCK:
        if(current_pathlength < MAX_PATHLENGTH)
        {
           IF_SIM(cosim_logwindow_fwrite(debug, "%d: writing out
                 block %d: %d.\n", loop_depth,
                 current_pathlength, payload);)
                // grab bottom 16bits of trace
                current_blocks[current_pathlength] = payload;
           IF_SIM(cosim_logwindow_fwrite(debug, "%d: wrote %d,
                 read %d.\n", loop_depth, payload,
                 current_blocks[current_pathlength]);)
                current_pathlength = current_pathlength + 1;

        }
        else
        {
           IF_SIM(cosim_logwindow_fwrite(debug, "%d:Buffer
                 full\n", loop_depth);)
        }
        current_checksum += payload;
    IF_SIM(cosim_logwindow_fwrite(debug, "%d:cksum = %d\n",
           loop_depth, current_checksum);)
        break;
case LOOPSTART:
    IF_SIM(cosim_logwindow_fwrite(debug, "%d: Entering loop\n",
           loop_depth);)

        if(loop_depth < MAX_LOOPDEPTH)
        {

           IF_SIM(cosim_logwindow_fwrite(debug, "stack ptr =
                 %d\n", stack_ptr);)
                // Copy the stored paths into the stack
                for(i = 0; i < num_paths; i++)
                {
                        int j;
                        for(j = 0; j < pathlengths[i]; j++)
                        stack[stack_ptr++] = blocks[i][j];
                        stack[stack_ptr++] = pathlengths[i];
                        stack[stack_ptr++] = checksums[i];
                        stack[stack_ptr++] = pathcounts[i];
                }
                stack[stack_ptr++] = others;
                stack[stack_ptr++] = num_paths;

                // Copy over current path info
                for(i = 0; i < current_pathlength; i++)
                        stack[stack_ptr++] = current_blocks[i];
                stack[stack_ptr++] = current_checksum;
                stack[stack_ptr++] = current_pathlength;

           IF_SIM(cosim_logwindow_fwrite(debug, "stack ptr =
                 %d\n", stack_ptr);)
                // Reset current path info
```

```
                current_checksum = 0;
                current_pathlength = 0;
                num_paths = 0;
                others = 0;
        }

        // Increment loop_depth regardless of depth
        loop_depth++;
        break;
case LOOPCONT:
    IF_SIM(cosim_logwindow_fwrite(debug, "%d:Continuing
        loop\n",loop_depth);)
        if(loop_depth < MAX_LOOPDEPTH)
        {
                int match = 0;

          IF_SIM(cosim_logwindow_fwrite(debug, "%d:LOOPCONT -
                less than MAX_LOOPDEPTH\n",loop_depth);)

        // Compare each stored path against current path
                for(i = 0; i < num_paths; i++)
                {
                        int j;

                        match = 1;

                 // if the pathlengths don't match, neither
                 will the paths...
                if(pathlengths[i] != current_pathlength)
                                match = 0;
                 // if the checksums don't match, neither will
                        the paths...
                 // (this should catch most cases that length
                        check doesn't)
                else if(checksums[i] != current_checksum)
                                match = 0;
                        else
                        // blocks by block compare (any pragmas
                                that can parallelize this?)
                        for(j = 0; j < pathlengths[i]; j++)
                        if(blocks[i][j] != current_blocks[j])
                                {
                                        match = 0;
                                        break;
                                }
                        // if we have a match, increment its
                                count and break out
                                 if(match)
                                 {
                                        pathcounts[i]++;
                                IF_SIM(cosim_logwindow_fwrite(deb
                                      ug, "%d:LOOPCONT matched
                                      path %d - %d reps\n",i,
                                      pathcounts[i]);)
                                         break;
                                 }
                        }

                 // If not match was found and we aren't at max
                        capacity, add the path to the list
                        if(!match)
                        {
```

```
                        IF_SIM(cosim_logwindow_fwrite(debug,
                               "%d:LOOPCONT adding
                               path\n",loop_depth);)
                        if(num_paths < MAX_PATHS_PER_LOOP)
                                {
                        pathlengths[num_paths] =
                               current_pathlength;
                        IF_SIM(cosim_logwindow_fwrite(debug,
                               "%d:LOOPCONT added pathlength[%d]
                               = %d\n",loop_depth, num_paths,
                               pathlengths[num_paths]);)
                        checksums[num_paths] =
                               current_checksum;
                        IF_SIM(cosim_logwindow_fwrite(debug,
                               "%d:LOOPCONT added
                               checksum\n",loop_depth);)
                                pathcounts[num_paths] = 1;
                        IF_SIM(cosim_logwindow_fwrite(debug,
                               "%d:LOOPCONT added
                               pathcount\n",loop_depth);)
                        for(i = 0; i < current_pathlength; i++)
                               blocks[num_paths][i] =
                                      current_blocks[i];
                        IF_SIM(cosim_logwindow_fwrite(debug,
                               "%d:LOOPCONT added
                               blocks\n",loop_depth);)
                                      num_paths++;
                                }
                                else
                                {
                        // else - should add in the others bin
                                      others++;
                        IF_SIM(cosim_logwindow_fwrite(debug,
                               "%d:LOOPCONT incrementing others
                               -> %d\n", others);)
                                }

                        }
                        // reset currents
                        current_pathlength = 0;
                        current_checksum = 0;
                }
                break;
case LOOPEND:
   IF_SIM(cosim_logwindow_fwrite(debug, "%d:Exitting
          loop\n", loop_depth);)
                // decrement loop_depth regardless
                loop_depth--;

                if(loop_depth < MAX_LOOPDEPTH)
                {
                        uint32 loop_cksum = 0;
                        long dataword;
                 // First check to see if current path needs to
                        be recorded

                        // Then write out the stored paths
                        for(i = 0; i < num_paths; i++)
                        {
                                uint8 j;
                                loop_cksum += checksums[i];
                        dataword = (pathlengths[i]<<16) |
                               checksums[i];
```

```
            co_stream_write(trace_output_stream,
                  &dataword, sizeof(co_uint32));

            co_stream_write(trace_output_stream,
                  &pathcounts[i],
                  sizeof(co_uint32));
            // For odd-length paths, we write out
                  garbage for last block number
            // since reader takes into account the
                  pathlength when dumping
            // - and MAX_PATHLENGTH must be even
            for(j = 0; j < pathlengths[i]; j=j+2)
                  {
                  dataword = (blocks[i][j] <<16) |
                        blocks[i][j+1];
                  co_stream_write(trace_output_stre
                        am, &dataword,
                        sizeof(co_uint32));
                   }
            IF_SIM(cosim_logwindow_fwrite(debug,
                  "%d:LOOPEND - wrote out
                  blocks\n", loop_depth);)
            }

            dataword = END_OF_LOOPTRACE;
      co_stream_write(trace_output_stream,
            &dataword, sizeof(co_uint32));
      co_stream_write(trace_output_stream, &others,
            sizeof(co_uint32));

            // Then restore the current vars
      // Remember to make this exactly reverse the
            way loop-start stored them
      // First, we pull over the local scalars
            (pathlength, chsum, and #paths)
            // then we pull off the current

current_pathlength = stack[--stack_ptr];
current_checksum = stack[--stack_ptr];
for(i = current_pathlength-1; i >= 0; i--)
            current_blocks[i] = stack[--stack_ptr];

            // Add superblock from completed loop
            if(current_pathlength < MAX_PATHLENGTH)
            {
            current_blocks[current_pathlength++] =
                  loop_cksum;
                   current_checksum+=loop_cksum;
            }

            num_paths = stack[--stack_ptr];
            others = stack[--stack_ptr];
      // Technically, we should pop these in
            reverse, but nothing say one ordering
            of paths is any more optimal than its
            reverse
            for(i = 0; i < num_paths; i++)
            {
            int j;
            pathcounts[i] = stack[--stack_ptr];
            checksums[i] = stack[--stack_ptr];
            pathlengths[i] = stack[--stack_ptr];
            for(j = pathlengths[i]-1; j >= 0; j--)
```

```
                                        blocks[i][j] = stack[--
                                              stack_ptr];
                                    }
                                }
                                break;
                        }

                }

                co_stream_close(trace_input_stream);
                co_stream_close(trace_output_stream);
}
```

B.4 memio.c

The next file is the second part of the hardware profiler. The part just handles buffering the stream of path profile traces into memory. The filename is "memio.c".

```
#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "pathlogger.h"

#define LOCAL_BUFFER_SIZE 2048

void memio_run(co_stream trace_output_stream, co_memory memblk, co_signal done)
{
        int32 buffer[LOCAL_BUFFER_SIZE];
        int32 blockOffset = 0;
        int32 bufferOffset = 0;
        int32 res;

        co_stream_open(trace_output_stream, O_RDONLY, UINT_TYPE(32));

        // Specify synchronous RAM for the local array memory
        co_array_config(buffer, co_kind, "sync");

        // Create Logger Window
        IF_SIM(cosim_logwindow debug;)
        IF_SIM(debug = cosim_logwindow_create("memio_run");)

        while(!co_stream_eos(trace_output_stream))
        {
                co_uint32 data;

                co_stream_read(trace_output_stream, &data, sizeof(uint32));

                if(bufferOffset >= LOCAL_BUFFER_SIZE)
                {
                    IF_SIM(cosim_logwindow_fwrite(debug, "Writing out buffer to
                           memory offset %ld\n", blockOffset);)
                    co_memory_writeblock(memblk,
                           LOCAL_BUFFER_SIZE*blockOffset*sizeof(int32),
                                buffer, LOCAL_BUFFER_SIZE*sizeof(int32));
                        blockOffset++;
                        bufferOffset = 0;
                }

              IF_SIM(cosim_logwindow_fwrite(debug, "Writing %ld to buffer
                    position #%ld\n", data, bufferOffset);)
```

```
                buffer[bufferOffset++] = data;
        }

       IF_SIM(cosim_logwindow_fwrite(debug, "FLUSH out remaining buffer to
              memory offset %ld with %ld elements\n", blockOffset,
              bufferOffset);)
        co_memory_writeblock(memblk,
LOCAL_BUFFER_SIZE*blockOffset*sizeof(int32),
                buffer, bufferOffset*sizeof(int32));

        // Send a signal to consumer process (memory write complete)
        res = LOCAL_BUFFER_SIZE*blockOffset + bufferOffset;
        co_signal_post(done, -res);
}
```

B.5 pathlogger_sw.c

The last file is the software path profiler and the profile recovery and output routine. The filename is "pathlogger_sw.c".

```
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include "co.h"
#define DECLARE_LOGIT_TIMERS
#include "pathlogger.h"

#if defined(MICROBLAZE)
#include "xparameters.h"
#define printf xil_printf
#else
// include sim file when not inside of xps
#include "cosim_log.h"
#endif

void timer_int_handler(void * baseaddr_p) {}
void uart_int_handler(void *baseaddr_p) {}
// Support different memory names used by template EDK projects
#define EXT0_BASEADDR XPAR_DDR_SDRAM_64MX32_MEM0_BASEADDR
#define EXT0_HIGHADDR XPAR_DDR_SDRAM_64MX32_MEM0_HIGHADDR

// The ext_alloc() function is required to set up the shared
// external memory resource, which is assumed for this example
// to be DDR memory available on the platform target.

#if defined(MICROBLAZE)
void *ext0_alloc(size_t size)
{
        static unsigned int address = EXT0_BASEADDR;
        void *ret;
        if ((EXT0_HIGHADDR+1 - address) < size) return(NULL);
        ret=(void *)address;
        address += size;
        printf("ext0_alloc: returning %lx\n", ret);
        return(ret);
}
#endif

extern co_architecture co_initialize(void *);
```

```
#ifdef SOFTWARE_LOGGER
// MAX_PATHLENGTH must be an even number to keep from breaking LOOPEND output
      below
#define MAX_PATHLENGTH 10
#define MAX_LOOPDEPTH 10
#define MAX_PATHS_PER_LOOP 10

short current_pathlength=0;
short current_checksum=0;
short current_blocks[MAX_PATHLENGTH];

uint8 loop_depth = 0;

// recorded paths and pathlengths for this level
// copying this array of objects to and from stack memory will
// probably account for the majority of the slowdown in the system
uint8 num_paths = 0;
short pathlengths[MAX_PATHS_PER_LOOP];
short checksums[MAX_PATHS_PER_LOOP];
long  pathcounts[MAX_PATHS_PER_LOOP];
short blocks[MAX_PATHS_PER_LOOP][MAX_PATHLENGTH];
long others;

// This should be at the bootom end of the DDR
long *stack_ptr = (long *)EXT0_HIGHADDR;

// This is the trace buffer
long *trace_buffer = (long *)EXT0_BASEADDR;

void logit(co_uint32 data)
{
        uint8   event_code;
        uint16  payload;
        int8 i;

        // split out the event code
       event_code = (uint8)(data>>28);
       payload = (uint16) data;

       IF_SIM(cosim_logwindow_fwrite(debug, "Processing trace: %lx. event =
              %lx\n", data, event_code);)
       switch(event_code)
       {
                case BASICBLOCK:
                        if(current_pathlength < MAX_PATHLENGTH)
                        {
                                // grab bottom 16bits of trace
                                current_blocks[current_pathlength] = payload;
                                current_pathlength = current_pathlength + 1;
                        }

                        current_checksum += payload;

                        break;
                case LOOPSTART:
                        if(loop_depth < MAX_LOOPDEPTH)
                        {
                                // Copy the stored paths into the stack
                                for(i = 0; i < num_paths; i++)
                                {
                                        int j;
```

```
                        for(j = 0; j < pathlengths[i]; j++)
                                *(--stack_ptr) = blocks[i][j];
                        *(--stack_ptr) = pathlengths[i];
                        *(--stack_ptr) = checksums[i];
                        *(--stack_ptr) = pathcounts[i];
                }
                *(--stack_ptr) = others;
                *(--stack_ptr) = num_paths;

                // Copy over current path info
                for(i = 0; i < current_pathlength; i++)
                        *(--stack_ptr) = current_blocks[i];
                *(--stack_ptr) = current_checksum;
                *(--stack_ptr) = current_pathlength;

                // Reset current path info
                current_checksum = 0;
                current_pathlength = 0;
                num_paths = 0;
                others = 0;
        }

        // Increment loop_depth regardless of depth
        loop_depth++;
        break;
case LOOPCONT:
        if(loop_depth < MAX_LOOPDEPTH)
        {
                int match = 0;

        // Compare each stored path against current path
                for(i = 0; i < num_paths; i++)
                {
                        int j;

                        match = 1;

                 // if the pathlengths don't match, neither
                        will the paths...
                if(pathlengths[i] != current_pathlength)
                                match = 0;
                 // if the checksums don't match, neither will
                        the paths...
                 // (this should catch most  cases that length
                        check doesn't)
                else if(checksums[i] != current_checksum)
                                match = 0;
                        else
                        // blocks by block compare (any pragmas
                               that can parallelize this?)
                        for(j = 0; j < pathlengths[i]; j++)
                               if(blocks[i][j] !=
                                      current_blocks[j])
                                        {
                                                match = 0;
                                                break;
                                        }
                 // if we have a match, increment its count and
                        break out
                        if(match)
                        {
                                pathcounts[i]++;
                                break;
```

```
                        }
                    }

                // If not match was found and we aren't at max
                    capacity, add the path to the list
                if(!match)
                {
                        if(num_paths < MAX_PATHS_PER_LOOP)
                        {
                        pathlengths[num_paths] =
                                current_pathlength;
                        checksums[num_paths] =
                                current_checksum;
                                pathcounts[num_paths] = 1;
                        for(i = 0; i < current_pathlength; i++)
                                blocks[num_paths][i] =
                                        current_blocks[i];
                                num_paths++;
                        }
                        else
                        {
                        // else - should add in the others bin
                                others++;
                        }

                }
                // reset currents
                current_pathlength = 0;
                current_checksum = 0;
        }
        break;
case LOOPEND:
        // decrement loop_depth regardless
        loop_depth--;

        if(loop_depth < MAX_LOOPDEPTH)
        {
                uint32 loop_cksum = 0;
                long dataword;
          // First check to see if current path needs to be
                recorded
          // rig this so that a loop cont will always happen
                before a loopend

                // Then write out the stored paths
                for(i = 0; i < num_paths; i++)
                {
                        uint8 j;
                        loop_cksum += checksums[i];
                 dataword = (pathlengths[i]<<16) |
                        checksums[i];
                        *trace_buffer++ = dataword;
                        *trace_buffer++ = pathcounts[i];
                 // For odd-length paths, we write out garbage
                        for last block number
                 // - ok, since reader takes into account the
                        pathlength when dumping
                        // - and MAX_PATHLENGTH must be even
                        for(j = 0; j < pathlengths[i]; j=j+2)
                        {
                        dataword = (blocks[i][j] <<16) |
                                blocks[i][j+1];
                                *trace_buffer++ = dataword;
```

```
                                        }
                                }

                                *trace_buffer++ = END_OF_LOOPTRACE;
                                *trace_buffer++ = others;

                        // Then restore the current vars
                        // Remember to make this exactly reverse the way
                              loop-start stored them
                        // First, we pull over the local scalars (pathlength,
                              chsum, and #paths)
                                // then we pull off the current

                                current_pathlength = *stack_ptr++;
                                current_checksum = *stack_ptr++;
                                for(i = current_pathlength-1; i >= 0; i--)
                                        current_blocks[i] = *stack_ptr++;

                                // Add superblock from completed loop
                                if(current_pathlength < MAX_PATHLENGTH)
                                {
                                 current_blocks[current_pathlength++] =
                                        loop_cksum;
                                        current_checksum+=loop_cksum;
                                }

                                num_paths = *stack_ptr++;
                                others = *stack_ptr++;
                        // Technically, we should pop these in reverse, but
                              nothing say one ordering of paths is any more
                              optimal than its reverse
                                for(i = 0; i < num_paths; i++)
                                {
                                        int j;
                                        pathcounts[i] = *stack_ptr++;
                                        checksums[i] = *stack_ptr++;
                                        pathlengths[i] = *stack_ptr++;
                                        for(j = pathlengths[i]-1; j >= 0; j--)
                                                blocks[i][j] = *stack_ptr++;
                                }
                        }
                        break;
        }

}
#endif //SOFTWARE_LOGGER
// The consumer process accepts status signals from the hardware process
// and reads/displays the resulting data
//
void consumer(co_signal done, co_memory memblk)
{
        int32 result;
        int32 i,j;
        int32 datasize, numblocks;
        IF_SIM(cosim_logwindow debug;)
        IF_SIM(debug = cosim_logwindow_create("consumer");)
        TIME_EVENT("consumer: start");
#ifdef SOFTWARE_LOGGER
        // Need this to be negative to match hardware
        result = ((long) EXT0_BASEADDR - (long) trace_buffer)/4;
#elif NO_LOGGER
        result = 0;
#else
```

```
        printf("Waiting for signal...\r\n");
        do {
                co_signal_wait(done, &result);
                TIME_EVENT("consumer: signal received");
                printf("Received signal: state is %d\r\n", result);
        } while (result >= 0);
#endif //SOFTWARE_LOGGER

        for( i=0; i < -result; i++)
        {
                static short path = 0;
                long tmp;
                short pathlength, checksum, n, j;
                long count;
                short blocks[10];

             co_memory_readblock(memblk, i++*sizeof(co_uint32), &tmp,
                    sizeof(co_uint32));
                if(tmp == END_OF_LOOPTRACE)
                {
                    co_memory_readblock(memblk, i++*sizeof(co_uint32), &tmp,
                           sizeof(co_uint32));
                        if(tmp > 0)
                                printf("Other paths total = %d\n", tmp);
                        printf("End of loop trace\n\n");
                        path = 0;
                        i--;
                        continue;
                }
                checksum = tmp & 0xFFFF;
                pathlength = tmp>>16;
             co_memory_readblock(memblk, i++*sizeof(co_uint32), &count,
                    sizeof(co_uint32));
                n = (pathlength%2 ? pathlength+1 : pathlength)/2;
                if(i+n > -result) break;

        printf("Reading path %d: Length = %d, cksum = %d, count = %ld\n",
                                path, pathlength, checksum, count);

             printf("Reading %d 32-bit words to get a %d length path\n", n,
                    pathlength);

             co_memory_readblock(memblk, i*sizeof(co_uint32), blocks,
                    n*sizeof(co_uint32));
                i+=n-1;
                IF_SIM(// word swap these
                for(j = 0; j < n; j++)
                {
                        short tmp2;
                        tmp2 = blocks[2*j];
                        blocks[2*j] = blocks[2*j+1];
                        blocks[2*j+1] = tmp2;
                })
                for(j =0; j < pathlength-1; j++)
                        printf("%d->", blocks[j]);
                printf("%d\n", blocks[j]);
                //printf("%lx\n", tmp);
                path++;
                printf("i = %d, result = %d\n", i, -result);
        }
        TIME_EVENT("consumer: end");
}
```

```
int main(int argc, char *argv[]) {
        co_architecture my_arch;
        int c;

        printf("Memory read/write example for Xilinx external memory.\n");
        my_arch = co_initialize(NULL);
        INIT_TIMERS;
        co_execute(my_arch);
        PRINT_TIMERS;
        printf("Memory read/write example complete.\n");
        IF_SIM(printf("Press Enter key to continue...\n");)
        IF_SIM(c = getc(stdin);)

        return(0);
}
```

REFERENCES

[1] G. Ammons, T. Ball, and J.R. Larus. Exploiting Hardware Performance Counters with Context Sensitive Profiling. In Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation, pages 85-96, 1997.

[2] S. Arramreddy and D. Riley. PCI-X 2.0 White Paper. http://www.pcisig.com/. Revision 1.0, April 5, 2002.

[3] T. Ball and J.R. Larus. Efficient Path Profiling. In Proceedings of MICRO 96, pages 46-57, December 1996.

[4] T. Ball and J.R. Larus. Optimally Profiling and Tracing Programs. ACM Transactions on Programming Languages and Systems, 16(4):1319-1360, July 1994.

[5] J. Brewer and J. Sekel. PCI Express Technology. White paper. http://www.dell.com/. Feb. 2004.

[6] D. Burger, T.M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, 1996.

[7] D. Burger and T. Austin, "The SimpleScalar Tool Set Version 2.0," Computer Architecture News, 25 (3), pp. 13-25, June, 1997.

[8] J. Dean, et al. ProfileMe: Hardware Support of Instruction-Level Profiling on Out-of-Order Processors. In Proceedings of the Thirteenth Annual IEEE/ACM International Symposium on Microarchitecture. pages 292-302, December 1997.

[9] I. Gonzalez and F.J. Gomez-Arribas. Ciphering Algorithms in Microblaze-Based Embedded Systems. Computers and Digital Techniques, IEEE Proceeding, Vol 153, no. 2, pp. 87-92, 6 March 2006.

[10] J.R. Larus. Whole Program Paths. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pages 259-269, 1999.

[11] J. Levon and P. Elie. Oprofile: A System Profiler for linux. `http://oprofile.sf.net`, September 2004.

[12] A. Mink, J. Roberts, and J. Antonishek. "Operating Principles of the VME MultiKron Interface Board," National Institute of Standards and Technology, NISTIR 5233, Aug. 1993.

[13] S. Tallam, X. Zhang, and R. Gupta. Extending Path Profiling across Loop Backedges and Procedure Boundaries. In Proceedings of the International Symposium on Code Generation and Optimization (CGO'04). pages 251-262, March 2004.

[14] D. Pellerin and S. Thibault, Practical FPGA Programming in C, Upper Saddle River, NJ: Prentice Hall Professional Technical Review, 2005.

[15] M. Pettersson. PerfCtr: Linux Performance Counters Driver. http://perfctr.sf.net

[16] T. Shanley and D. Anderson. PCI System Architecture. Addison-Wesley Professional, 1999.

[17] B. Sprunt. The Basics of Performance-Monitoring Hardware. Micro, IEEE, Vol. 22, Iss. 4, Pages 64-71, Jul/Aug 2002.

[18] R. Stallman and others. Using and Porting the GNU Compiler Collection. Free Software Foundation, 1999.

[19] K. Vaswani, M.J. Thazhuthaveetil, and Y.N. Srikant. A Programmable Hardware Path Profiler. In Proceedings of the International Symposium on Code Generation and Optimization, (GCO'05). pages 217-228, March 2005.

[20] Intel VTune Performance Analyzer.

[21] Xilinx Inc., Embedded System Tools Reference Manual, Embedded Development Kit EDK 8.2i, UG111 (v6.0), June 23, 2006.

[22] Xilinx Inc., FSL Bus v. 2.00.a Product Specification, DS449 (v1.6), December 1, 2005.

[23] Xilinx Inc., MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 8.2i, UG081 (v6.3), August 29, 2006.

[24] Xilinx Inc., On-Chip Peripheral Bus V2.0 with OPB Arbiter(v1.10c), DS401, August 31, 2006.

[25] Xilinx Inc., OPB Timer/Counter (v1.00b), DS465, June 9, 2006.