BitRaker Anvil: Binary Instrumentation for Rapid Creation of Simulation and Workload Analysis Tools

Brad Calder Todd Austin Don Yang
Timothy Sherwood Suleyman Sair David Newquist Tim Cusac

BitRaker Inc.
San Diego, CA
www.BitRaker.com

Abstract

A wide range of ARM developers from architects, to compiler writers, to software developers, need tools to understand, analyze, and simulate program behavior. For developers to achieve high levels of system and program correctness, performance, reliability, and power efficiency these tools must be fast and customizable to the problems at hand.

BitRaker Anvil is a tool building framework allowing developers to rapidly build tools to achieve these goals. BitRaker Anvil uses binary instrumentation to modify ARM binaries for the purpose of analyzing program behavior. BitRaker Anvil equips the developer with an easy to use API that allows the user to specify the particular program characteristics to analyze. Using this API, the developer can create custom tools to perform simulation or workload analysis several orders of magnitude faster than using a cycle level simulator.

Prior binary instrumentation technology requires that analysis code be merged into the same binary as the code to be analyzed. A key new feature of our binary instrumentation framework is *ReHost* analysis, which allows an instrumented ARM binary to make calls to analysis code that is written in the native format of the desktop machine. Using this for cross-platform ARM development results in analysis that runs orders of magnitude faster while simultaneously reducing the size of the ARM binary images.

1 Introduction

BitRaker Anvil provides a framework (API) for building custom binary instrumentation tools for ARM to rapidly simulate and analyze any program behavior you want. Our binary instrumentation framework provides the ability to automatically add hooks to an ARM binary, which then trigger calls to analysis code. A new binary is created with the instrumentation hooks and analysis code. When a hook is encountered during execution, it invokes a corresponding analysis routine passing in parameters that provide information (e.g., effective addresses, register values) about the current state of the program's execution. These analysis routines can then perform detailed simulation or profiling.

1.1 Uses for Binary Instrumentation

Binary instrumentation has significant advantages for device and chip architects, compiler writers, and software developers.

Device and Chip Architects. Binary instrumentation provides an Architect with efficient design space exploration and workload characterization tools to make the best design trade-off decisions early in the design cycle. In addition it allows an architect the ability to rapidly build *custom* tools that target any problem encountered in the design process. In order to produce a chip that performs as efficiently as possible (in terms of time, space and power) an architect must completely understand the workload that the chip is being designed for. Without a solid understanding of the target workload, an all too common result is a chip that looks good on paper but that fails to perform at the expected level when run with real applications. Performing a thorough characterization of the workload should be done *early* in the design cycle and binary instrumentation is a fast and simple way to complete this characterization even before a detailed architecture model is completed.

As an example, an architect needs information such as the workload's instruction mix and common code sequences. This is necessary to determine what instructions on which to focus hardware optimization. In addition, since embedded processors typically focus on running a small handful of applications, knowing the exact needs of those applications can yield significant improvements by guiding the development of hardware accelerators. Another example of important workload analysis, is monitoring the memory footprint in order to implement a high performance and adequately sized memory hierarchy in the processor. All of this information can be gathered quickly and thoroughly for the full workload's execution using binary instrumentation.

Compiler Writers. Embedded software writers typically spend a significant amount of time generating highly optimized binaries. This optimization process often requires a profile to be gathered and then fed back into the compiler to guide the optimization. The profile often contains the most frequently executed control flow paths through the program's execution [2], as well as potentially identifying what parts of the program have semi-invariant values to be specialized [3]. The BitRaker Anvil binary instrumentation framework enables the building of binary profiling tools to quickly and accurately gather these feedback-directed profiles.

Software Developers. The BitRaker framework provides the ability to quickly create custom programmer productivity tools.

These are tools a software developer can use to examine all aspects about their program's execution including code coverage, where time is being spent, memory usage, memory errors, power usage, thread race conditions, and more. An example tool is a hierarchical code profiler that reveals where time is being spent in an application for the purpose of guiding program optimization efforts. Another example is a tool that efficiently uncovers memory leaks and memory access violations. We have built both of these tools using the BitRaker Anvil framework and provide them as commercial products. BitRaker Anvil provides you the ability to quickly build such tools and then map the generated information back to the source code, showing you exactly where important events occur.

1.2 Advantage of Binary Instrumentation

Gathering the above mentioned information using binary instrumentation has several advantages over simulation and source level instrumentation.

Source Level Instrumentation. Systems that are tied to source code instrumentation for gathering profile information are also tied to a specific language and often require a special compiler that can consume the inserted annotations. Using binary instrumentation means that you are not tied to using a particular compiler nor source code language to perform the analysis.

Furthermore, source code instrumentation systems require that *all* of the source code is available for instrumentation. Often times this is not feasible when, for example, third party and operating system libraries are linked into the binary. Therefore, an important advantage of binary instrumentation is that it makes the whole program available for analysis, which includes being able to instrument and analyze the impact from closed source libraries.

Cycle Level Simulation. Most of the analysis described above can be gathered by hacking a cycle level simulator. One of the main advantages of using binary instrumentation is that it can perform the analysis orders of magnitude faster than using a cycle level simulator. Because of this increased analysis speed, the whole program can be simulated and analyzed. This speed increase makes it possible to *analyze the entire execution of an application*. In comparison, due to the overhead of a cycle level simulator less than 1% of the program's execution can be analyzed, and this still takes more time than whole program analysis with binary instrumentation. In addition, whole program analysis with binary instrumentation will be significantly more representative of the programs behavior, since it will see all of the phases of the program's execution [4].

Post Processing Huge Traces. Another path for gathering some of this information is to generate huge traces on disk and continuously post process those traces to gather statistics about the program's execution. Real programs run for many seconds, minutes, or longer and execute billions or trillions of instructions. Storage of full trace information for such programs is difficult or infeasible. A key advantage of binary instrumentation is that the analysis can gather aggregate statistics as the program runs with only a small slowdown over the uninstrumented binary's execution time. Then at the end of execution or after a specified amount of time, the aggregate statistics can be written out. In doing this, you no longer have to deal with large bulky traces, and the information can still be gathered in a short amount of time. This is typically how tools built with BitRaker Anvil are used.

1.3 Paper's Contributions and Outline

This paper presents the first binary instrumentation system made available for the ARM platform. Section 2 describes how the system works by building an example data cache simulator. This section describes *InHost* instrumentation. This approach to binary instrumentation always links the analysis code into the original binary providing a new instrumented binary.

A key new feature of our binary instrumentation framework is *ReHost* instrumentation described in Section 3. ReHost is built on the realization that for cross-platform ARM development the instrumented image size and analysis performance overhead needs to be kept small. It is therefore advantageous to keep the analysis code and its data separate from the instrumented ARM binary. Our ReHost solution keeps the analysis code and data as an x86 binary. Then when an instrumented ARM binary executes a hook, the call is *ReHosted* to the x86 analysis binary. Using this for cross-platform ARM development results in 10x faster analysis and significantly smaller instrumented ARM binary images over the prior techniques. This is shown in Section 4 where we provide instrumentation and run-time overhead results using the current BitRaker Anvil release. We then summarize the paper in Section 5.

2 ARM Binary Instrumentation

We now describe in more detail the process of binary modification, and some of the issues that must be dealt with to preserve precise program behavior.

2.1 How BitRaker Anvil Works

BitRaker Anvil uses symbol information to read an ARM binary into an Intermediate Representation (IR) [1] and provides a programmable API to allow developers to traverse over the IR adding calls to the developer's analysis routines. The Analysis routines can be used to profile information or simulate structures the developer is interested in. Thus a developer creates both an instrumentation tool using the BitRaker Anvil API and an analysis shared library. Together the instrumentation and analysis program constitute an ARM Analysis Tool, which can be used over and over again to analyze the behavior of ARM binaries.

The tool developer starts by creating an instrumentation program using the BitRaker Anvil API. The instrumentation program takes as input an original binary and outputs a new instrumented binary. This instrumented binary, when run, will call the analysis routines at the appropriate points to gather profile information or to perform detailed architecture simulation. This process is shown in Figure 1. To illustrate the ease with which BitRaker Anvil can be used to create an analysis tool, we will

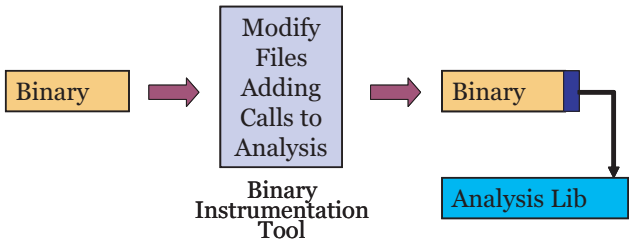

Figure 1: The process of Binary Instrumentation.

describe how to use BitRaker Anvil to quickly create an ARM Data Cache Simulator.

2.1.1 Creating the Instrumentation Tool

The developer begins by creating an instrumentation tool (shown in Figure 1) using the BitRaker Anvil API to specify the type of information the developer is interested analyzing. The BitRaker Anvil API separates the user from the details of binary modification allowing them to easily focus their study of the parts of the program they care about. Building a tool with BitRaker Anvil to perform analysis requires first identifying what information a user wants to gather. The developer needs to then figure out where in the binary (what IR structures need to be instrumented) to gather this information.

What Information to Gather. A user first needs to decide what type of information they want to gather. Is it information about branches, control flow, memory references, instructions counts, etc? Or are they trying to simulate the behavior of a part of the architecture, like the data cache? For a cache simulator, the memory address stream needs to be examined, and this is accomplished by instrumenting every load and store instruction. This dynamically tracks the effective addresses used during execution.

What type of BitRaker Anvil IR Construct to Instrument. Once you have determined what type of information to gather, a user needs to decide what type of BitRaker Anvil IR construct to instrument. For example, to instrument the memory address stream you will be using the BitRaker Anvil IR Instruction construct to attach hooks to load and store instructions. Then when any load or store instruction executes the corresponding analysis routine is called. If you want to track information about the procedure calls (e.g., a trace of procedure call invocations and their procedure names), you would instrument the BitRaker Anvil IR Procedure construct.

Where to Instrument the Program. Once you have decided which type of IR structure to instrument, you need to decide if you want to call your analysis routine before or after the structure executes. You also need to decide if you want to instrument all of the structures of that type or only select structures of that type. For example, do you want to instrument all procedures, or only trace a few select procedures to reduce the profiling overhead. The BitRaker Anvil API provides the ability to easily specify all of this.

Once you have made the above decisions, the process of creating a BitRaker Anvil tool consists of creating an Instrumentation Tool and an Analysis Library. The steps required to create an instrumentation tool consists of:

1. Declare analysis function prototypes using the `CreateFunctionPrototype` API, so that the instrumentation tool knows what analysis routines to link the instrumentation hooks to. For the cache simulator there are three analysis routine prototypes shown at the start of Figure 2. These are `profile_start`, `cache_sim` and `profile_end`. These routines are all defined and provided in the analysis library the user will write. The rest of the arguments to the `CreateFunctionPrototype` routine specify the number of parameters, and the types of each of the parameters.
2. Find all the instances of the type of structure (procedures, basic blocks, or instructions) in the BitRaker Anvil IR where you want to place the instrumentation calls to your analysis routines. For our example, this is done by traversing over all of the instructions and examining, which instructions are of type load or store as shown by the `for` loop in Figure 2, and then using `GetInstFlag` to determine the type of the instruction.
3. Insert the instrumentation calls into the static executable. This is shown with the `InstrumentInst` API in Figure 2. The parameters specify which analysis routine to call, and what parameters to pass to it. The example shows inserting an instrumentation hook before a load or store instruction. When this hook is executed the corresponding analysis routine will be dynamically called passing dynamic information (the effective address) and static (whether a load or store) to the analysis routine for profiling. Note, the effective address has to be dynamically calculated before it is passed to the analysis routine. The API knows to do this, since the argument type was specified to be `eAEFFADDR` in the `CreateFunctionPrototype` definition of `cache_sim`.
   Also shown in Figure 2 is that insertion of a function call to `func_setup` before the binary starts executing, and a call to `func_finish` when it finishes executing using the `InstrumentProject` API. This allows the analysis code to initialize its data structures, and to output to the file `datafile` with the cache statistics at the end of execution.
4. The final step is writing out the new binary, which is shown using the `WriteObj` function. When this function is called, the new binary is created with the instrumentation hooks added, and analysis code linked in.

Once the code is created to perform the instrumentation, it is compiled using the BitRaker Anvil library to create an ARM Instrumentation Tool for Windows or Linux. The input to this tool will consist of (at the minimum) an ARM binary, and it will output a new ARM binary with the instrumented hooks added.

2.1.2 Creating the Analysis Routines

The next step is for the developer to create the analysis routines (for the library in Figure 1) to be called when the hooks execute. These routines record or simulate the information being profiled

```
func_setup = CreateFunctionPrototype(project,
           "profile_start", 0);
func_cache_sim = CreateFunctionPrototype(project,
           "cache_sim", 2, eAEFFADDR, eACONSTWORD);
func_finish = CreateFunctionPrototype(project,
           "profile_end", 1, eACHARPOINTER);

...

//call the func_setup analysis routine at start
InstrumentProject(project, eABEFORE, func_setup);


//traverse over all of the instructions in binary
for(SetFirstInst(bb, inst); !IsInstNull(inst);
                            IncInst(bb,inst))
{

  //get the type of current instruction
  EAInstFlagType inst_flag = GetInstFlag(inst);

  //check the type of the instruction
  if ( inst_flag & eAIFLAG_Load ) {
     //add instrumentation hook for load
     InstrumentInst(inst, eABEFORE, func_cache_sim,
                    eAEFFADDR, CACHESIM_LOAD);
  } else if ( inst_flag & eAIFLAG_Store ) {
     //add instrumentation hook for store
     InstrumentInst(inst, eABEFORE, func_cache_sim,
                    eAEFFADDR, CACHESIM_STORE);
  }
}

//call the func_setup analysis routine at end
InstrumentProject(project, eAAFTER, func_finish,
                  datafile);

WriteObjInHost(obj,newBinFileName,analysisLibName);
```

Figure 2: Instrumentation Code Using the BitRaker Anvil API for Building a Cache Simulator.

```
void profile_start( void )
void cache_sim( unsigned long addr,
                int accesstype )
void profile_end( char* outfile  )
```

Figure 3: Corresponding Cache Simulation Analysis Routine Definitions.

during the execution of the instrumented program. For the example, an analysis routine for the cache simulator would take as input a memory address (for a load or store) and initiate the cache access. The analysis routine will be called by the hooks inserted into the binary during execution. With BitRaker Anvil, the memory addresses seen during profiling are the same addresses seen during the execution of the original uninstrumented binary. For the cache simulation example, the analysis routine definitions are shown in Figure 2.

2.1.3 Running the Tool and Instrumented Binary

When the instrumentation tool is run on the original binary (shown in Figure 1), the binary is read into the BitRaker Anvil IR. The IR is then traversed adding instrumentation calls (e.g., at every load and store) to the analysis routines (e.g., cache simulator) to analyze the behavior of the program the developer is interested in.

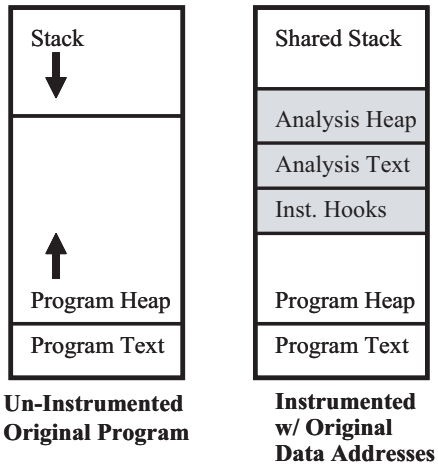

Figure 4: Layout in Memory of Original Program (left) and Instrumented Binary (right).

When instrumentation is finished, the instrumentation tool, using the BitRaker Anvil API, writes out a new instrumented binary, which has been augmented with the instrumentation calls to the analysis code. Then when the instrumented binary is run the analysis information is gathered using the developer's analysis routines. When the program finishes execution, the analysis code reports to the user the final results of the profile (e.g., the data cache miss rate).

2.2 Preserving Precise Program Behavior

When using a binary instrumentation system for simulation and workload analysis, it is important to be able to see the same original data addresses in the instrumented binary as the original binary.

To achieve this, we must first understand how ARM binaries are laid out in memory. ARM binaries place their code combined with data starting at a low address in memory. The heap is then placed after that and grows upward increasing in the memory virtual address space as shown in Figure 4. Then the stack starts at address 0x08000000 and grows down toward the heap.

BitRaker Anvil ensures for simulation analysis that the original addresses are maintained by not moving the original global data, heap, and stack. BitRaker Anvil inserts the instrumentation hooks, and the analysis code and data in between the original heap and stack, ensuring that enough room is left for both of them. This is shown in the right side of Figure 4.

If the instrumentation hooks or the analysis code need to use temporary storage, it shares the stack with the thread. This is preferred, since each thread has its own stack and stack pointer. When execution transfers to the instrumentation hooks and analysis code, all data placed on the stack will be removed before returning to the original executed instructions. This guarantees that the stack is kept clean for the original instructions, and the original code is none the wiser.

3 ReHost Instrumentation

In prior art [5], the analysis code and data is directly added to the program's address space being analyzed. We call this *InHost Instrumentation*, since the analysis code lives within the original binary. It is added by statically linking it in, or dynamically inserting it into the binary. When an analysis call is invoked, the thread that encountered the hook will only continue executing when the analysis call returns. When the instrumented binary is run natively on a machine with a lot of memory, this provides efficient execution of the analysis code for instrumentation.

In the ARM embedded environment there are two key issues that need to be addressed for binary instrumentation – (a) how to keep the memory and execution footprints of the analysis code and data low, and (b) Cross–platform ARM software development using emulation on x86 desktop machines (as is most often done) requires the *emulation of analysis code*, making the approach unusable. Both of these issues are addressed by using our new *ReHosted Instrumentation* approach.

ReHosted Instrumentation is the process of taking an original ARM binary and producing an instrumented ARM binary, which has hooks added to it to trigger analysis calls, but all of the analysis code and data is kept in a separate native x86 binary. Figure 1 shows that InHost instrumentation links the analysis code into an ARM binary. In comparison, ReHost instrumentation rehosts the analysis call by performing communication between the instrumented ARM binary and an analysis x86 binary. The ReHost memory layout stores the analysis code and data in a separate process address space, so it is does not exist in the InHost memory layout shown in Figure 4.

3.1 ReHost Implementation

ReHost instrumentation works as follows:

1. ReHost Instrumentation - The instrumentation API is exactly the same as with InHost described in Section 2. The only difference is that the writing out of the binary is done by calling `WriteObjReHost` instead of `WriteObjInHost`. This tells the BitRaker Anvil API to not link in the analysis code and data, but instead link in a ReHost Communicator, which will appropriately transfer the hooks to the analysis code.
2. Native Analysis Code - The instrumented ARM binary is kept separate from the analysis x86 binary. This allows the binaries of the user code and the analysis code to be of different ISAs. The analysis code is compiled into an x86 shared library with the appropriate routines exported so they can be called. This shared library is then dynamically loaded by the ReHost Dispatcher when it starts up on the x86 machine. One advantage of this is that the x86 analysis code can be built using any language supported on the x86 machine.
3. ReHosting the Hooks - When the instrumented ARM binary executes either on a cross-platform emulator or on an ARM board, a corresponding ReHost Dispatcher is running on the x86 desktop machine. When an analysis hook is encountered during ARM execution, the ReHost Communicator will transfer the analysis routine ID along with any parameters to the ReHost Dispatcher.
4. Processing Analysis Hooks - The ReHost Dispatcher on the x86 side is responsible for interpreting the analysis routine ID and parameters and performing the call to that routine.

The two main advantages of ReHost are that it keeps the analysis overhead low and enables efficient cross-platform development.

3.2 Keeping a Low Profile

For the ARM embedded platform it is important to keep the memory and the execution time footprint of the analysis code to a minimum. For example, hand held game developers trying to analyze their product have only a small amount of memory free to use for added instrumentation code.

Another important aspect is reducing the execution time overhead. Ideally the analysis code should not not pause the original program, while the analysis code chews on the data. This is what happens with traditional binary instrumentation techniques [5]. This can cause issues when using instrumentation to perform very detailed analysis. For example, the game developers can only tolerate a 33% slow down (going from 60 FPS down to 40 FPS) while performing the analysis. In addition, the ability to get useful information from hardware counters relies on the analysis code not interfering too much with the execution time. Therefore, ReHosting is beneficial for providing low perturbation of the performance data.

To address these issues, ReHost instrumentation keeps analysis code and data separate from the original binary and its execution. There is no reason to keep the analysis code and data in the same process, since it only consumes data from the hooks in the original program's execution. In addition, there is no reason why the original program's execution has to wait until the analysis code is done processing the hook before it can continue executing. Therefore, the analysis code and data can live as an x86 binary, and have this data buffered and communicated to it over a network link from the ARM board. This approach is enabled by ReHost instrumentation.

3.3 Efficient Cross-Platform Development

Another difference in the use of binary instrumentation for ARM development is that a significant amount of ARM development is performed cross-platform on x86 desktop machines. During ARM software and hardware development, ARM binaries are often run on a emulator or simulator that runs on a x86 machine. Under these conditions, if one was to use binary instrumentation and link the analysis code and data into the ARM binary, then the analysis code and data would have to also be emulated. This would result in a significant slow down and would prevent the analysis of the binary from being practical for this type of cross-platform development.

This problem is solved with our ReHost analysis, since all of the analysis code and data are kept in an x86 binary, which runs at native speeds. In fact, for cross-platform development the analysis code runs so fast, in comparison to the emulated ARM binary being analyzed, we can perform aggressive analysis with only a 2 to 5 times slow down.

4 Cross-Platform Performance

In this section we examine the performance overhead of using binary instrumentation for the following two applications:

- Instruction Count (IC) - This tool counts the total number of instructions executed when running a binary. This is accomplished by adding an instrumentation hook to every basic block, passing to the analysis code the number of static instructions in the basic block. The analysis code has a global counter that accumulates the number of instructions, which is written out when the application completes.
- Data Cache Simulator (CS) - This tool gathers data cache simulation results (hits and miss rates), for a first level data cache. This is accomplished by profiling every load and store instruction as shown in Figure 2.

Results are gathered by running the ARM binaries on our own instruction set simulator called BitRaker SimForge. SimForge is a ARM binary emulator that allows users to run ARM binaries on Linux and Windows machines for cross-platform development. Results are reported for ten common embedded benchmarks, and the results compare the benefit of using ReHost instrumentation to InHost instrumentation.

Table 1 shows the increase in binary size due to binary instrumentation. The results show that ReHost instrumentation only increases the binary size by 20 to 30%, whereas InHost instrumentation increases the binary size 2 to 3 times.

Table 2 shows the execution time increase due to binary instrumentation. The results show that InHost instrumentation can have significant performance overhead when performing cross-platform development, because the analysis code has to be emulated along with the original binary. For the InHost cache simulator, an additional 20x increase in execution time is seen on average due to having to emulate the cache simulation each time a load or store instruction is executed. In comparison, since the ReHost instrumentation keeps all of its analysis code in an x86 binary, it does not suffer this overhead. The overhead for ReHost cache simulation is only a 2x increase in execution time. By keeping the analysis in native format, ReHost instrumentation enables complex analysis to be performed with very little additional overhead.

Binary	Orig	IC-Rehost	IC-Inhost	CS-Rehost	CS-Inhost
adpcm	44.5kB	1.23x	3.29x	1.28x	3.71x
crc32	39.2kB	1.23x	3.48x	1.29x	3.96x
forth	197.0kB	1.27x	2.27x	1.26x	2.49x
gsm	164.9kB	1.22x	2.13x	1.27x	2.56x
jpeg	344.6kB	1.17x	1.80x	1.24x	2.31x
mad	703.9kB	1.17x	1.71x	1.23x	2.12x
patricia	78.0kB	1.26x	2.77x	1.30x	3.19x
rijndael	100.3kB	1.12x	2.06x	1.38x	3.31x
susan	134.1kB	1.26x	2.43x	1.37x	3.23x
tiff	561.9kB	1.18x	1.76x	1.20x	2.03x
Avg	236.8kB	1.21x	2.37x	1.28x	2.89x

Table 1: Binary Size Increase. The original binary file size is in kilobytes. The next two columns show the increase in binary size when performing instruction count instrumentation for ReHost and InHost. The final two columns show the increase in binary size for ReHost and InHost for data cache simulation.

Binary	Orig	IC-Rehost	IC-Inhost	CS-Rehost	CS-Inhost
adpcm	286.8s	2.35x	16.93x	1.59x	10.98x
crc32	124.9s	1.71x	9.60x	2.39x	25.64x
forth	186.2s	2.20x	14.04x	2.28x	21.72x
gsm	290.7s	1.88x	11.37x	1.54x	10.45x
jpeg	55.6s	1.83x	10.70x	2.20x	22.15x
mad	190.2s	1.33x	4.95x	2.28x	24.06x
patricia	222.5s	1.55x	7.81x	1.77x	16.17x
rijndael	244.1s	1.22x	3.46x	2.74x	29.85x
susan	37.9s	1.57x	7.85x	1.96x	18.85x
tiff	527.3s	1.91x	11.33x	2.09x	20.26x
Avg	216.6s	1.76x	9.80x	2.08x	20.01x

Table 2: Cross-Platform Execution Time Increase. The original execution time in seconds on BitRaker SimForge. The next two columns show the increase in execution time when performing instruction count instrumentation for ReHost and InHost. The final two columns show the increase in execution time of ReHost and InHost for data cache simulation.

5 Summary

This paper presents the first commercial binary instrumentation system made available for the ARM platform called BitRaker Anvil. Binary instrumentation has a wide range of applications for ARM developers, whether they are architects, compiler writers, or software developers. Binary instrumentation enables the rapid creation of high speed profiling and simulation tools which are useful in all manner of workload analysis and software development.

We presented ReHost instrumentation, which keeps the analysis code and data in a separate x86 binary, and demonstrated the importance of using ReHost instrumentation for cross-platform development. We showed that complex analysis tools can be run in ReHost mode with only a factor of 2 slowdown over uninstrumented binaries.

BitRaker Anvil equips it's users with the ability to perform static and dynamic analysis, and provides powerful building blocks that users can combine to build new tools limited only by their imagination. We encourage you to explore its advantages for simulation, compiler profiling and programmer productivity analysis.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, 1986.

[2] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[3] T.B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, January 1996.

[4] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[5] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.