

# DSPTune: A Performance Evaluation Toolset for the SHARC Signal Processor

Suleyman Sair  
Guisepe Olivadoti  
David Kaeli  
ECE Department  
Northeastern University  
Boston, MA  
kaeli@ece.neu.edu

Jose Fridman  
Analog Devices  
Norwood, MA  
Jose.Fridman@analog.com

## Abstract

*Performance tuning in the embedded systems domain poses a new set of challenges for software and hardware designers. Techniques proven to work for general purpose architectures can not always be directly applied to the signal processor environment. Program analysis and simulation tools have been shown to be invaluable in the analysis of general purpose microprocessors. We anticipate that similar tools will be needed to analyze the characteristics of signal processing architectures and applications as well. To meet this need, we have developed DSPTune, a program analysis toolset for the Analog Devices' SHARC DSP. This paper describes our toolset, and provides examples of its use.*

## 1. Introduction

The importance of program analysis tools has been clearly demonstrated by the number of published research papers that use some form of tracing or instrumentation system. Tools such as ATOM [2], Shade [3] and SimOS [4] have been developed to enhance our ability to analyze software and hardware systems. The impact of these tools has enabled researchers to evaluate the merits of various design tradeoffs in a timely and cost-effective manner.

The study of embedded systems on the other hand, is a relatively immature field. As the importance of signal processors has grown, researchers have begun to actively study the behavior of these systems [5, 6]. Program analysis and simulation tools which specifically target embedded systems will be in great demand. Such tools will allow us to better analyze, design, and fine-tune hardware and software targeting DSP-based systems.

This paper describes an instrumentation tool targeting Analog Devices' SHARC DSPs, called *DSPTune* [1]. We

have modeled our tool after the ATOM [2] toolkit developed and supported by Compaq on the Alpha architecture. Our toolset provides a platform for building a wide range of customized analysis tools. We provide an interface for instrumenting C and assembly-level programs at various instrumentation granularities.

This paper is organized as follows. In Section 2 we discuss the target architecture for our work. In Section 3 we review related instrumentation tools. In Section 4 we present the design of DSPTune, including examples of its use and discuss several implementation aspects of our tool. In Section 5 we conclude, suggesting directions for future work.

## 2. The Analog Devices SHARC DSP architecture

The *ADSP-2106x SHARC* (Super Harvard ARchitecture Computer) is a 32-bit digital signal processor. As the name suggests, the SHARC has separate on-chip data and program memories. With the addition of a 32-entry instruction cache, the SHARC can perform simultaneous instruction fetch and data access operations. Four independent buses for dual data, instructions and I/O, provide unconstrained data-flow to and from the computation units [8].

Some of the SHARC Instruction Set Architecture (ISA) features include:

- 3 32-bit floating point units,
- 2 sets of 16 40-bit data registers,
- 2 independent data address generators (DAGs),
- 128KB-512KB of on-chip SRAM (divided into two blocks),
- a program sequencer containing a 32-entry instruction cache, and

- a glueless multiprocessor interface.

The processing unit of the SHARC can execute any instruction in a single cycle. It provides a 3 deep pipeline. In addition to traditional arithmetic and logical operations, it provides instructions that target signal processing applications (e.g., multiply-accumulate).

The DAGs provide circular buffers, modulo and bit-reversal operations in hardware. These provide for efficient data access in signal processing algorithms such as filtering. They also support indirect addressing with pre or post modify, along with immediate addressing. The program sequencer allows zero overhead loops with a hardware looping instruction. The maximum loop nest allowed is 6 levels deep. The program sequencer allows every instruction to be executed conditionally.

The complexity of SHARC presents a number of architectural and design issues. We use the DSPTune toolset [1] to perform execution-driven simulation of a range of architectural features. DSPTune is modeled after the ATOM tool developed for Compaq's Alpha microprocessors [2]. DSP-Tune allows the user to instrument applications compiled for the Analog Devices SHARC DSP. Next we review related instrumentation and simulation systems.

### 3. Instrumentation systems

There have been a number of instrumentation systems developed on commercial platforms. Tools for the Sun SPARC, Compaq Alpha, and MIPS microprocessors have been developed to support research in architecture and system software. We will begin by briefly describing these related toolsets.

*ATOM* [2] allows for the selected instrumentation of executable binaries on Compaq Alpha processors. This tool uses *Object Modifier* (OM), a link-time code modification system. OM inserts procedure calls to user defined analysis routines at link time. ATOM can instrument programs at the procedure level, the basic block level or the instruction level. Various simulations can be performed by calling a rich set of analysis routines at execution time.

*Shade*[3] is a tool which combines instruction-set simulation with extensible trace generation capability. The application program is simulated and traced by dynamically cross-compiling and caching. The user defined *analyzer* utilizes the simulation and the tracing capabilities of Shade. Analyzers see Shade as a collection of library functions. Analyzers call these functions to identify the application program to be simulated and specify the level of tracing detail.

*PatchWrx* [7] on the other hand, is a static binary-rewriting instrumentation tool for capturing *full* instruction and data address traces on the Compaq Alpha platform run-

ning Microsoft Windows NT. This toolset modifies the binary image prior to execution to capture traces as the program runs. PatchWrx allocates a fixed trace buffer in physical memory at system boot time. Trace records are captured when the modified executable encounters an instrumented instruction at run-time. Data load and store addresses can also be captured using PatchWrx.

PatchWrx modifies the binary image by identifying all branches in the executable, and replaces them with unconditional branches into a *patch section*. The patch section is additional code generated by PatchWrx, that is appended to the end of the executable image. The patch section issues a PALcall (a privileged light-weight call instruction). The target of the PALcall records the necessary trace information in the trace buffer. The trace contains control points in the execution (i.e., conditional or unconditional branches, task switches and interrupts), and register values needed to decipher load and store addresses. This information is used later to reconstruct a full trace. PatchWrx captures all operating system activity in the trace.

*SimOS* [4] is an emulation-based technique. Emulation builds a layer between the host machine and the operating system under evaluation. SimOS emulates enough of the system to allow the operating system and applications to run correctly. This system provides a flexible interface for collecting operating system-rich traces. SimOS provides simulators of CPUs, caches, memory systems and a number of different I/O devices such as disks, ethernet interfaces and etc. Fast simulation techniques *fast-forward* through less-interesting sections of a workload. More detailed and, hence, slower techniques are employed to focus on portions of the execution which are of interest. SimOS provides *annotations* as a mechanism for invoking user-defined routines when a particular event occurs. Annotations do not affect workload execution or timing, but have access to the entire hardware state of the simulated machine.

DSPTune follows many of the same principles present in the ATOM and PatchWrx toolsets. The main goals of DSPTune are to:

1. provide a set of library routines that can be used to instrument a C or assembly code program that will run on an Analog Devices SHARC DSP,
2. provide an environment to study the benefits of new architectural features, and
3. provide a tool that both limits the runtime overhead associated with instrumentation and preserves the original address stream.

One of the key features of this toolset is that we utilize the secondary set of registers provided in the SHARC DSP. This allows us to pass parameters efficiently without sav-

ing/restoring registers. This allows us to provide more accurate simulation, and faster simulation.

The complexity of SHARC presents a number of architectural and design issues. Using the DSPTune toolset, we can begin to address these questions. Next we present the design of our toolset.

## 4. Design of DSPTune

DSPTune allows selective instrumentation at program locations as specified by the user supplied *instrumentation* routine. DSPTune uses *execution driven simulation*. Calls to *analysis* routines are inserted into the application. The analysis routines and the application code are executed in the same run-time environment, thus information passing can be accomplished by procedure calls. In order to preserve the run-time behavior of the application, address values passed to the analysis routines are adjusted to offset the effect of additional calls and fix-up code.

DSPTune works at either the source or assembly level. This presents some issues:

1. the code (C or assembly) for library routines is needed in order to instrument them, and
2. additional fix-up routines are necessary to calculate run-time information such as branch outcomes and instruction addresses.

Though embedded applications may use standard library routines, many developers prefer to write their own library functions for performance reasons. While the first issue is unlikely to occur frequently, we plan to address this issue in the next version of the toolset.

We have implemented our instrumentation toolset on top of Analog Devices' VisualDSP framework. VisualDSP is a Windows based software development environment for Analog Devices DSPs. It provides the following elements:

- an integrated development environment which supports editing programs, managing projects, and performing revision control,
- build tools,
- a source level debugger with support for DSP simulation and emulation, and
- help information and online access to available SHARC documentation.

The build tools are composed of an optimizing C compiler, SHARC assembler, linker, loader, simulator and overlay tool. These tools can be accessed from the Windows IDE as well as through a command line.

DSPTune generates the instrumented executable in four steps. This is illustrated in Figure 1. In the first step, the application code is parsed and transformed into an *intermediate representation (IR)*. This structure treats the program as a collection of procedures, and procedures as a collection of instructions.

Then, the IR is fed to the instrumentation routine. In the DSPTune IR, each instruction has fields to hold pointers to lists of operations to be performed before and after the instruction. The instrumentation routine fills these fields by inserting call instructions to specified analysis routines. Then the instrumented IR is traversed to form the instrumented application code. Finally, analysis and instrumented application codes are linked to form the instrumented object code.

Simulations can be performed by running the DSP executable on a simulator or an emulator. Instrumented code can be run natively on the SHARC, though much of our work uses the cycle-accurate emulator provided by Analog Devices. A simulation will have two outputs:

1. the output of the application program, and
2. the output of the analysis routines

To illustrate how this tool works and evaluate its associated, we provide a set of example instrumentation routines. Our first example captures the maximum level of dynamic loop-nest in a program.

The SHARC ISA provides a looping instruction [8]. An assembly code example is provided in Figure 2. This is a counter-based loop which loads the loop counter with the value of 20, and repeats until the loop counter reaches zero (*lce* stands for loop counter expired). This particular example reads an integer from an array in line (2), calculates its' absolute value in line (3) and stores the result into another array in line (4).

The read in line (2) and the write in line (4) are examples of *indirect addressing with post-modify*. The specified index registers (*i0* and *i1*) are updated by the value in the modify register (*m0*) after the read and write operations have been performed.

When the SHARC executes a *do until* instruction, the program sequencer pushes the address of the last loop instruction and the termination condition for exiting the loop onto the the loop address stack. If the loop is counter based, the loop counter value is pushed onto the loop counter stack, also updating the loop address stack. The top-of-loop address, which is the address of the instruction following the *do until* instruction, is pushed on the PC stack.

In the 3-stage pipeline of the SHARC, the termination condition is evaluated when the second to last instruction is being executed (line 2 in our example). For counter-based

```

lcntr = 20, do label until lce; (1)
r0 = dm(i0,m0); (2)
r1 = abs r0; (3)
label: dm(i1,m0) = r1; (4)

```

**Figure 2. A simple loop example**

loops, the counter value is also decremented at this point. If the termination condition is not met, the SHARC fetches the instruction from the top-of-loop address stored in the PC stack. If the termination condition is true, the sequencer fetches the instruction following the last instruction of the loop and pops the loop address stack (and loop counter stack if the loop is counter-based) and PC stack.

The SHARC ISA provides 6-deep, hardware-based, loop address and counter stacks. Our first analysis routine will see how much of this hardware resource is used by a set of signal processing kernels. For this purpose we selected a subset of the *University of Toronto Benchmark Suite (UTDSP)* [10]. This suite contains both kernels and complete applications performing typical DSP algorithms.

Figure 3 shows our DSPTune instrumentation routine. In order to find the maximum level of loop-nest we need to examine *do until* instructions as well as the last instruction of the loop. We also need to display our results.

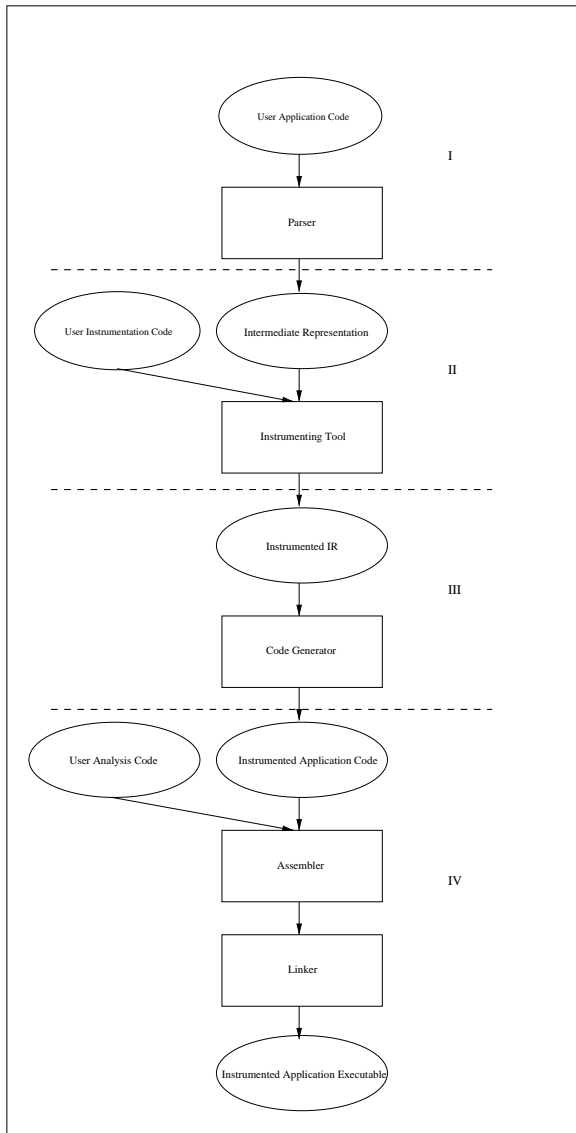
The program is traversed with the help of:

- GetFirstProc (locate the first procedure),
- GetNextProc (locate the next procedure),
- GetFirstInst (locate the first instruction in the current procedure),
- GetNextInst (locate the next instruction),
- isLoopInst (test if the current instruction is a *do until* instruction), and
- isLoopEnd (test if the current instruction is located at the end of a loop body).

Calls to the analysis routines are inserted by means of:

- AddCallInst (insert a call before or after this instruction), and
- AddCallProc (insert a call before or after this procedure), and
- AddCallProg (insert a call before or after the program).

Analysis routines are shown in Figure 4. The `report` analysis routine prints the maximum nesting depth to standard output. The `enter_do` routine keeps track of the maximum depth. The `exit_do` routine decrements the



**Figure 1. Execution flow of DSPTune**

```

Instrument(Prog * p){

Proc * pr;
Inst * i;

for(pr = GetFirstProc(p); pr != NULL; pr = GetNextProc(p)){
    for(i = GetFirstInst(pr); i != NULL; i = GetNextInst(pr)){
        if(isLoopInst(i))
            AddCallInst(i,InstBefore,``enter_do``);
        else
            if(isLoopEnd(i))
                AddCallInst(i,InstAfter,``exit_do``);
    }
}

AddCallProg(ProgAfter,``report``);
}

```

**Figure 3. Loop nest instrumentation routine**

current\_depth counter to indicate that we have just exited a level of nesting. Note that since the SHARC ISA does not allow multiple loops to share the same ending instruction, only one decrement is needed.

Table 1 shows the results of this example. We see that the SHARC uses at most 2 slots out of its 6-deep stack. The compiler can not always utilize a 6-deep stack for the applications studied.

The amount of overhead experienced due to instrumentation is directly proportional to the type of instrumentation used. For the example shown in Figures 3 and 4, the increase in CPU cycles is reasonable (the maximum is 3X). But notice that the analysis routines do not require any parameters. This removes the need for fix-up code needed to calculate run-time information. Moreover, since no parameters are passed, we use the *shadow registers* available on the SHARC for our analysis routines. This eliminates the need to save and restore registers.

In our second example, we count the number of dynamic loads and stores executed in the applications. We would expect a much higher amount of overhead associated with this scenario. For each call we need to load, increment, and restore the counter. This introduces 3 cycles of overhead. Also, saving the clobbered register and restoring takes 2 additional cycles. The call to the analysis routine takes 3 cycles as well. There are 2 more cycles expended to save r2, which is used as a temporary to transfer the frame pointer. So each call adds 10 extra cycles to the execution time. Results are shown in Table 2.

What we can see is that the overhead has increased substantially (as large as 12X). Still, this is reasonable and kept

to a minimum through the effective use of shadow registers where possible.

#### 4.1. Implementation issues

There are some implementation issues that need to be addressed when using DSPTune.

- Instrumentation is performed on an Intel x86-based platform. Thus the instrumentation routine is compiled into an x86 executable and applied to the IR of the application provided by the Parser. All analysis routines on the other hand are compiled by the VisualDSP compiler and target the SHARC. Thus data representations assumed by the instrumentation routine may be incompatible with the analysis routines.
- As we are instrumenting source code, variable names existing in application code may conflict with those defined in the analysis code. This may cause a syntax error during compilation.
- If precompiled library routines are used in the application code, these will not be instrumented (we are presently working on a binary level instrumentation system.)

The calling convention for the SHARC passes the first 4 parameters via registers while the remaining parameters are pushed onto the stack [9]. The return value is stored in r0. Since we do not want to perturb the SHARC stack, analysis routines should not return any values (i.e., they should return void).

```

include<stdio.h>

int max_depth = 0;
int current_depth = 0;

void report(void){
    printf("max. depth : %d \n",max_depth);
}

void enter_do(void)(
    current_depth++;
    if(current_depth > max_depth)
        max_depth = current_depth;
}

void exit_do(void){
    current_depth--;
}

```

**Figure 4. Loop nest analysis routine**

<i>Name Of Benchmark</i>	<i>Maximum Depth</i>	<i>% Increase in Execution Time</i>
FFT	2	37
IIR	2	147
Lattice Filter	1	51
LMS	1	61
Matrix10x10	2	302

**Table 1. Loop nest instrumentation for the kernel benchmarks**

<i>Name Of Benchmark</i>	<i>Number of Loads &amp; Stores</i>	<i>% Increase in Execution Time</i>
FFT	62510	696
IIR	4130	1035
Lattice Filter	11036	755
LMS	15256	1178
Matrix10x10	2267	1075

**Table 2. Load/Store instrumentation for the kernel benchmarks**

Here is a list of techniques we have employed to facilitate fast context switches for procedure calls introduced by DSPTune instrumentation:

- If the analysis routine does not require any parameters, switch to the shadow register set before calling the analysis routine. Work with this secondary set of registers in the analysis routine and switch back to primary set of registers before returning control to the application program.
- If calls are being inserted before or after program execution, no registers need to be saved as their values will not be needed.
- Allocate a new stack frame when a call to an analysis routine is made. This prevents any interference with the application's stack environment.

Program and data segments of the analysis routines are placed after those of the application program. Therefore, address changes are limited to the instruction addresses of the application code. As we mentioned before, additional instructions are necessary to calculate run-time information. These operations are done by wrapper functions. The application code calls these wrapper functions to:

1. save the state of the processor,
2. provide any run-time information to the analysis routine,
3. store parameters in required locations, and
4. pass control to the analysis routine.

If addresses of instructions are needed as parameters, the wrapper routine adjusts the address value to offset the effects of inserting procedure calls.

## 5. Summary

We have developed a program instrumentation and simulation toolset for Analog Devices' SHARC DSP. This tool provides a library of routines that enable the user to capture profile information as a program is run. This information can then be analyzed through user defined analysis routines. Presently we have support for cache, branch prediction and instruction-level modeling.

Although this toolset is still under construction, we have completed the initial framework and have obtained some encouraging results. We hope, when finished, DSPTune will prove to be a useful tool for tuning applications developed in high-level languages targeting high-end embedded hardware.

## 6. Acknowledgements

This work is supported through an NSF Instrumentation Grant NCRI97-29856, and by generous donations from Analog Devices.

## References

- [1] S. Sair, D. Kaeli and J. Fridman, "A Study of Dynamic Branch Prediction for SHARC DSPs," *Proceedings of CASES'99*, October 1999, pp. 39-48.
- [2] A. Srivasta and A. Eustace, "ATOM, A System for Building Customized Program Analysis Tools," *Proc. of the 1994 ACM Conference on PLDI*, 196-205.
- [3] R. Cmelik and D. Keppel, "SHADE : A Fast Instruction-Set Simulator for Execution Profiling," *Proc. of the 1994 ACM Conference on SIGMETRICS*, 128-137.
- [4] M. Rosenblum, S. Herrod, E. Witchel and A. Gupta, "Complete Computer System Simulation : The SimOS Approach," *IEEE Journal of Parallel and Distributed Technology*, 1998.
- [5] A. Sudarsanam, S. Liao, and S. Devadas. "Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures," *Proceedings of the ACM/IEEE Design Automation Conference*, 1997.
- [6] G. Araujo and S. Malik, "Code Generation for Fixed-Point DSPs," *ACM TODAES*, Vol. 3, No. 2, April 1998.
- [7] J. Casmira, D. R. Kaeli and D. Hunter, "Tracing and Characterization of NT-based System Workloads," *Digital Technical Journal, Special Issue on Tools and Languages*, Vol. 10, No. 1, December 1998, pp. 6-21.
- [8] Analog Devices, *ADSP-2106x SHARC User's Manual*, 1997.
- [9] Analog Devices, *C Compiler Guide & Reference for the ADSP-2106x Family DSPs*, 1998.
- [10] C. Lee, *UTDSP Benchmark Suite*, University of Toronto, Canada, 1992.