SquishDSP – A process tool for systematic identification of parallel operations in
instructions generated from an optimizing compiler or present in custom code

M. R. Smith
Electrical and Computer Engineering, University of Calgary, Calgary, Alberta, Canada
and Waltham-Stow Consulting Ltd., Calgary, Alberta, Canada
Email: -- smithmr @ ucalgary.ca, Waltham-Stow @ home.com

Abstract—The 2106X SHARC processors are typical of DSP processors capable of
handling highly parallel operations. Writing code to take advantages of the SHARC
architecture is difficult because of the interdependencies between the uses of CPU
and memory resources within the instruction sequence. SquishDSP is a process tool that
takes hand generated assembly code, or the output of an optimizing compiler, as input.
The output is a highly parallel version of the original source code. A paradigm shift
within SquishDSP is the use of a business tool for scheduling DSP resource usage. Initial work using the
Microsoft Project GUI interface shows how the developer can visualize the timesavings associated with particular code
adjustments. The optimized code sequence can then be automatically generated as parallel instructions. Preliminary
results are provided where a code sequence generated by the optimizing VisualDSP++ V1.0 compiler had its execution
time reduced by a further factor of two after application of the SquishDSP tool. The speed advantages obtained will depend
on the form and design of the DSP algorithm and associated code and processor architecture. Work is underway
to determine if other business tools, such as those from Primavera, permit easy modeling of more complex DSP
resource dependencies.

Index Terms-- Code Optimization, Parallel Instruction
Identification, Analog Devices ADSP2106X SHARC processor

I. INTRODUCTION

The highly parallel architecture of DSP CPUs permit
instructions where there are simultaneous execution of
multiplications, additions and subtraction together with data
and program memory operations alongside cached
instruction fetches. In recent articles [1], [2] and [3] a
process was introduced for writing Analog Devices
SHARC ADSP2106X assembly code in a form that would
permit the development of highly parallel instructions for
the efficient implementation of DSP algorithms.

Such optimization processes, which involve hand
manipulation of instructions, are straightforward for simple
DSP loops involving only a few lines of code. Application
of the process to larger code sequences is difficult because
of the issue of reliably identifying all of the
interdependencies in the use of resources within a highly
parallel code sequence. This effect is demonstrated in the
following simple SHARC code example.

1) \[ F_6 = \text{dm}(4, M_4); \]
2) \[ F_1 = F_2 * F_4, F_8 = F_8 + F_{12}, F_{12} = \text{pm}(I_{12}, M_{12}); \]
3) \[ F_3 = F_1 * F_5, F_8 = F_8 + F_{12}, F_{12} = \text{pm}(I_{12}, M_{12}); \]
4) \[ F_5 = F_3 * F_6, F_8 = F_8 + F_{12}, F_{12} = \text{pm}(I_{12}, M_{12}); \]

Instruction (1) could easily be scheduled in parallel with
either the existing operations of instructions (2) or (3) to
reduce overall execution time. However instruction (1)
cannot be moved in parallel with instruction (4) because of
the interdependencies in the use of resource F6, shown in
bold, within both instructions.

The recognition of the interdependence of the use of the
DSP CPU resources and associated movement of
instructions to create shorter code sequences is something
that requires automated assistance to ensure reliable code
generation. In this paper we demonstrate the use of
SquishDSP, a process-optimizing tool. A paradigm shift
within SquishDSP is to involve a business-scheduling tool
to schedule DSP resource usage. Preliminary results using
Microsoft Project [4] show that is possible to visualize the
DSP processor’s resource inter-relationships in any
sequence of instructions. The GUI interface of MS Project
allows the developer to see the effect of further
manipulation of the code and assist in quickly identifying
additional improvements in execution time.

SquishDSP was intended to assist in the identification of
possible parallel operations within hand-generated custom
code sequences. Preliminary results are drawn from using
SquishDSP to identify further parallelizations in code
sequences generated from the VisualDSP++ V1.0
optimizing compiler [5]. As with all optimizing techniques,
timesavings are dependent on the specifics of the DSP
algorithm, the manner in which it is coded and the
processor upon which it is placed.
In the conclusion we discuss how it may be possible to combine SquishDSP with other business tool characteristics to permit modeling of a variety of processor architectures and complex resource relationships such as cache utilization and DAG transfer restrictions.

II. INTERNAL CONSTRUCTION OF THE SquishDSP TOOL

SquishDSP is currently developed as a research tool, rather than as a commercial product. The initial incorporation of the business tool, Microsoft Project, was a cost saving measure to decrease the development time of the SquishDSP utility. The cost of a license is less than one day of a programmer’s time. However the serendipitous inclusion of a business tool has proved to be a paradigm shift in investigating DSP CPU resource utilization. An unexpected advantage was how the GUI interface permitted straightforward further manipulation of existing code sequences with fast and easy identification of execution time improvements.

To simplify the construction of SquishDSP, it was assumed that the input would be assembly code that was known to have the correct SHARC syntax. This implies output from an optimizing compiler or code that had already been passed through the assembler without warnings or errors. The code from the White Mountain VisualDSP++ V1.0 optimizing compiler has proved particularly suitable. This compiler generates code with the register usage already suitable for further custom parallelization. Multiplication and addition operations are described in the form

\[ F_n = F(0, 1, 2, 3) * F(4, 5, 6, 7) \]
and
\[ F_n = F(8, 9, 10, 11) + F(12, 13, 14, 15) \]

with memory operations typically written using post-modification operations.

SquishDSP performs a series of passes on the input data. In the first group of passes, the input instructions are broken up into “project tasks”. Each task is an “atomic”, or non-parallel, SHARC instruction. Local resource dependencies must be identified. For example the instruction

\[ R1 = R2 + R3, \text{dm}(I4, M4) = R1; \]

has local resource dependencies on register R1. This instruction must be broken down to become

\[ \text{dm}(I4, M4) = R1; R1 = R2 + R3; \]

and not

\[ R1 = R2 + R3; \text{dm}(I4, M4) = R1; \]

Copies of the original instruction sequences are maintained as embedded comments. These comments can be used as part of process quality control measure by permitting easier validation of the final restructured code.

A second group of passes reformat the file into one suitable for input into Microsoft Project. These are essentially parser stages where resource interdependencies are identified.

The developer temporarily pauses the use of the SquishDSP to bring in a “txt – Default Task Information” file into Microsoft Project. Figure 1 shows the Project GUI interface for a typical code sequence generated by SquishDSP V1.0. Note the overuse of many resources within any one SHARC execution cycle. The power of commercial project management tool is brought into play on the rescheduling of the instructions by selecting the RESOURCE LEVEL NOW option. Figure 2 shows the rescheduled tasks.

At this point in the development process, the programmer may identify inefficient time periods in the execution of the code. These efficiencies typically occur at the start and end of a loop as are evident with the first two instructions of the loop shown in Figure 2.
Using the project management tool’s GUI interface, it is a simple matter of “cutting and pasting” tasks to investigate the new code efficiency obtained by moving one or more instructions to change the characteristics of the “loop unrolling”. The new leveled, rescheduled code sequence is then saved into a second text file for further processing by the SquishDSP utility.

In the final pass, the SquishDSP utility reformat the rescheduled code sequence into a series of allowed parallel instructions involving multiplier, ALU and memory resources. The final output is in the form of a standard SHARC “.asm” file.

A process control review phase should then be performed. This review phase can make use of the original code sequence, which is embedded as ordered comments within the new optimized source file.

It is a simple process for the developer to leave open both a NotePad file containing the original source file and the project management program, and continually operate SquishDSP. The efficiency associated with minor, or major, modifications to the design of the code in the source file can be quickly brought to a visualization stage.

III. EXAMPLE OF THE USE OF THE SQUISHDSP TOOL

Figure 3 shows a simple DSP algorithm to convert temperatures from Centigrade to Fahrenheit. This algorithm has the typical DSP characteristics of a loop, involving constant and variable operations, multiple memory accesses and address calculations together with extensive multiplication and addition operations.

The code has been written in a format to encourage a compiler to make optimum use of the SHARC ADS2106X parallel architecture. This includes writing the code with a fixed loop size (MAXSIZE = 128) rather than passing in the array length as a subroutine parameter. This approach allows the optimizing compiler to efficiently determine how to “unroll the code loop” and identify possible parallel operations. The use of a dm array for input and a pm array for output was intended to invoke the optimizing compiler into generating parallel multiply memory operations.

This code, passed through the VisualDSP++ V1.0 compiler, produces a tight loop of 2 assembly instructions to implement each line of the original loop. However the compiler did not maximize the resource usage of the SHARC processor.

Figure 4 shows “C” code sequences with variants of “unrolling the loop”. From previous experience it was anticipated that upper code sequence would produce the tighter code. However, the VisualDSP++ compiler produced code using a software loop and a poorly optimized sequence of 5 assembly instructions for each of the original code calculations.
through the original source sequence permitting SquishDSP to identify a new series of parallel ADSP2106X instructions. The new code has an improved efficiency of 1.5 cycles per conversion compared to the original 2 cycles per conversion. However, visualization of the resource interdependence using the GUI interface quickly identifies processor resource stalls at the beginning and end of the hardware loop operation.

Various code manipulations were investigated using SquishDSP to identify other code formats. The optimum code sequence was found to occur by pulling a further two instructions outside the loop (see figure 7). This resulted in a final SquishDSP sequence, figure 8, where the resources of the ADSP2106X architecture are fully utilized. This code sequence has the maximum possible efficiency of 1 cycle per conversion compared to the original 2 cycles per conversion from the original compiler code.

This is obviously a rather straightforward example. However it does demonstrate the potential behind using the research tool SquishDSP, combined with a commercial project management scheduling tool to assist in the optimization of time critical DSP code sequences.

![Figure 5](image5.png)

**Figure 5.** The VisualDSP++ V1.0 optimizing compiler generates code that does not fully utilize the parallel resource usage possible within the SHARC instruction.

![Figure 6](image6.png)

**Figure 6.** The following set of highly parallel instructions was obtained after passing the VisualDSP++ code through SquishDSP. The loop operations are reduced from 12 cycles to 8 cycles.

![Figure 7](image7.png)

**Figure 7:** The use of the SquishDSP identified that this version of the original source code led the most efficient parallel use of processor resources.

![Figure 8](image8.png)

**Figure 8:** The final source code generated by SquishDSP tool provides the maximum use of the processor’s resources.
IV. CONCLUSION AND RESEARCH DIRECTIONS

It is well known that the form of the DSP algorithm implementation has a profound effect on the background noise level associated with the final processed signal. Turner and Smith [6], [7] have discussed this in the context of the SHARC processor. Recognizing the efficiencies possible within the reorganized code sequence is difficult on processors, such as the SHARC, whose architecture is capable of handling many operations in parallel.

In this paper, we have discussed the use of a tool, SquishDSP, as part of a process to identify additional parallel operations within a DSP code sequence produced by hand, or from an optimizing compiler. An example was provided to show the increased code efficiencies possible on code produced for Analog Devices SHARC 2106X processor using the Visual DSP++ V1.0 optimizing tool.

Even when many of the features present in SquishDSP are incorporated into later versions of the optimizing compiler, the tool will permit easier further customization of code using knowledge from the developer. This specialized a priori knowledge is difficult to incorporate into any compiler designed to handle the general characteristics of DSP algorithms.

As the efficiencies are very algorithm dependent, it is important to allow the developer to try a variety of different code formats to identify the design that best matches the characteristics of the DSP processor used. This is achieved within the SquishDSP tool by incorporating a pass that used the scheduling capability of a business project management tool. It was demonstrated using Microsoft Project that the GUI interface of these business tools provides the capability of allowing the DSP code developer to quickly visualize code efficiencies. The GUI interface makes further code adjustment, such as moving additional instructions out of the loop, become straightforward and reliable.

As mentioned earlier, SquishDSP is currently developed as a research tool, rather than as a commercial product. The paradigm shift provided by the serendipitous inclusion of a commercial project management program has proved to be a major factor in three areas: - the rapid development of the SquishDSP tool, the ease of its use and the efficiency of the rescheduled code. However a major research thrust must be undertaken in determining the best approach to adding resource dependencies to the SHARC DSP code before presentation to the process management program.

An additional pass [9] has been incorporated into SquishDSP to recognize these cache conflicts in the final parallel code sequence, and identify how to remove the cycles lost through “Cache Thrashing” by the appropriate placement of additional NOPs in the code. Typically the time lost through executing each additional NOP is gained back as the 3 or 4 cycles lost through the cache conflicts are removed through code adjustment.
However, research needs to be done on how to visualize this cache resource dependency and permit the pm() operations to be automatically placed in a way that avoids the cache conflicts without wasting cycles through additional NOPs.

In order to model cache dependency it is necessary, in colloquial business terms, to invent a resource utilization characteristic that will

- Schedule all the normal tasks for 4 groups of consultants. (ADDER, MULTIPLIER, DM_BUS and PM_BUS).
- The tasks must be scheduled taking into account that the contracts for the PM_BUS group allow these consultants to work out at a health club on the second day after they have performed a task.
- However, the business bottom line means that there are only 32 passes to the health club which must be used within certain 16-day cycles across the elapsed time of the total project.
- The scheduling task is complicated by the fact that the health passes can also be used in the first 16-day cycle at the start of a project. This is to compensate for the fact that late in the project the consultants may not be able to use the passes because of other time conflicts.

We are currently investigating whether other project management tools, such as those from Primavera [10], have the ability to permit custom resource scheduling in such a totally “un-business like” manner.

This paper has been used to report the preliminary results of the use of SquishDSP V2.0 in ADSP2106X applications. Further work is required to ensure that the final code sequence is reliable under all circumstances. In particular, will important resource dependencies be missed through idiosyncrasies of personal coding style?

It is intended that SquishDSP will be demonstrated at a workshop at the upcoming SHARC2001 conference. Copies of the executable or source files can be obtained from the author for a minimal fee. If there is enough industrial interest, the author is willing to assist in maintaining SquishDSP in an open source format. Any further successes of matching more complex DSP CPU resource utilization to the functionality of commercial project management tools will be the subject of a future paper [11].

A tutorial of the use of SquishDSP V2.0 tool will be made available through web-links starting at www.enel.ucalgary.ca/People/Smith and followed through to the ADSP2106X application notes within the ENCM515/ENEL515 sub-web. ENCM515 is a 4th year DSP processor-specific course offered at the University of Calgary, Canada. The course is based around a series of hands-on laboratories involving the manipulation of audio channel characteristics to improve the sound stage of head phones. The laboratories make use of the ADSP21061 EZ-LITE or ADSP21065 EZ-LAB evaluation kits.

V. ACKNOWLEDGEMENTS

The author wishes to acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University Support Program of Analog Devices (Con Korikos). Useful feedback on the SquishDSP tool was obtained from David Levine (Analog Devices, Boston), Brian Howse (Cell-Loc, Calgary, Canada) and Roch Demaere (University of Calgary, Canada).

VI. REFERENCES