THE SANDBLASTER AUTOMATIC MULTITHREADED VECTORIZING COMPILER

Sanjay Jinturkar[*], John Glossner[*,†], Vladimir Kotlyar[*], and Mayan Moudgill[*]

[*]Sandbridge Technologies, Inc.
1 North Lexington Ave.
White Plains, NY 10601 USA
{sjinturkar, jglossner , mayan, vkotlyar}@SandbridgeTech.com

[†]Delft University of Technology
EEMCS
Delft, The Netherlands

Abstract

Compilers for Digital Signal Processors (DSP) have been inefficient. The constraints have been two-fold. First, signal processing algorithms that use non-associative arithmetic are not easily described in high-level languages such as C, C++, and Java. Second, historical DSP architectures have been difficult compiler targets due to their non-orthogonal instruction sets. With modern DSP architectures having removed the latter restriction, this paper focuses on compiler generated code. The Sandbridge Sandblaster compiler is able to analyze DSP kernels and use the POSIX infrastructure to automatically generate vectorized multithreaded code from an ANSI-C description. Using this compiler we achieved 3.92 taps per cycle out of a theoretical maximum of 4 taps per cycle for a 20-tap WCDMA receive FIR filter including all overhead for multithreading.

1 Introduction

Programmer productivity is one of the major concerns in complex DSP applications. Because most DSPs are programmed in assembly language, it takes a very large software effort to program an application. The primary reason is due to a fundamental mismatch between DSP data types and C language constructs. A basic data-type in DSPs is a saturating fractional fixed-point representation; C language constructs, however, define integer modulo arithmetic. This forces the programmer to explicitly program saturation operations. Another difficulty has been finding parallelism in code written in C even though it is inherent in DSP applications. This requires super-computer class compiler optimizations. A final difficult consideration is parallelizing saturating arithmetic. Because saturating arithmetic is non-associative, the order in which the computations are computed is significant and a compiler may not reorder these loops.

To overcome these limitations DSP programmers have evolved a number of techniques including inlined assembly[3], creating large libraries of reusable code, using assembly crafted routines “under-the-hood” of the compiler in the form of intrinsics[4][5][6], or adding specialized (and non-portable) keywords into the C code that specific compilers may recognize[2]. The fundamental limitation is that the programmer is still required to manually find all the parallelism and (in the case of VLIW processors) schedule them without conflict onto multiple execution units.

Sandbridge has taken a unique approach to this problem. Rather than force a compiler to fit architecture, we designed a truly compilable architecture with a supercomputer-class compiler. In addition to standard scalar optimizations[7], our compiler supports sophisticated loop optimizations, memory disambiguation, automatic vectorization, and automatic parallelization including automatic multithreading.

In this paper, we describe the Sandblaster compiler, which is able to exploit instruction level, data level and thread level parallelism. Specifically, we describe the compiler and optimizations that allow for automatic generation of non-associative, vectorized, and fully multithreaded DSP algorithms from an ANSI-C level description.

Previously, we have presented compiler solutions targeted for single-threaded compilation[11]. In this paper we extend our work to describe the data level, instructions level and thread level optimizations. In Section 2 we describe classical optimizations including DSP semantic analysis. In Section 3 we describe how we optimize and parallelize programs. We then provide concluding remarks and results.

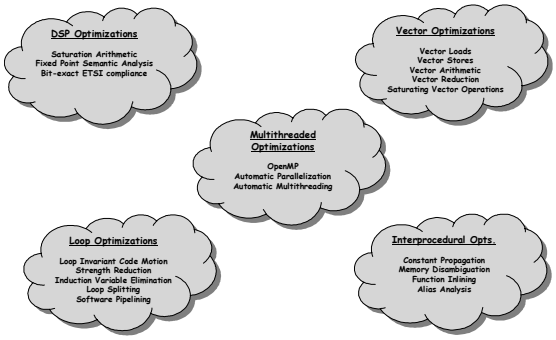

Figure 1. Compiler Optimizations

2 Sandbridge Optimizing Compiler

Sandbridge Technologies has built a new best-in-class optimizing ANSI C compiler for the Sandblaster DSP (described in [8][9]), which applies a number of high performance supercomputer class compiler optimizations to enable the generation of very efficient assembly code and exploit instruction level (ILP), data level (DLP) and thread level (TLP) parallelism .

The Sandbridge Technologies software tool chain consists of an Integrated Development Environment (IDE), optimizing compiler, assembler, linker, simulator and a debugger[11]. The IDE provides an easy to use graphical user interface to all the software tools. The IDE is based on the Open source Netbeans integrated development environment[12]. The IDE is the graphical front end to the C compiler, assembler, simulator and the debugger .

Figure 2 shows the optimizations and their ordering in the compiler backend. The ordering is constantly fine tuned to address new application areas.

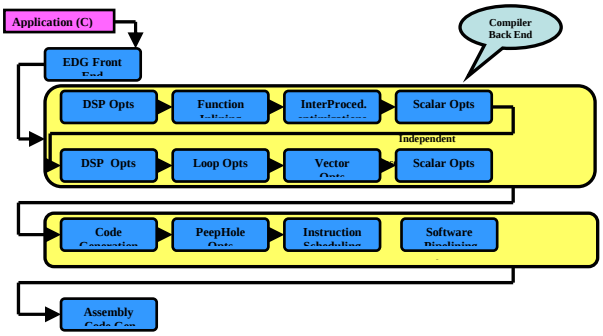

Figure 2. Compiler Phases

2.1 Scalar Optimizations

The compiler applies a number of classical scalar optimizations to improve the performance of control code [7]. These optimizations are applied early in the compilation process. The optimizations include common sub expression elimination, constant propagation, range propagation, dead code elimination, strength reduction, predication and bit width analysis.

2.2 Loop Optimizations

In addition to classical compiler optimizations, the compiler applies sophisticated loop optimizations. These include invariant code motion, loop unrolling, unroll-and-jam, loop distribution, and loop splitting, cross iteration load elimination, invariant load/store elimination and use of the loop counter.

2.3 Memory Disambiguation

Memory disambiguation plays a critical role in enabling the above optimizations. In the absence of memory disambiguation, the compiler will make conservative choices that do not reorder memory reads and writes. This may make instruction and data level parallelism difficult to detect. The Sandbridge compiler applies integer programming based memory disambiguation to loop, in addition to interprocedural memory disambiguation.

For example, in the following code, the compiler must determine if the iterations can be executed in any order:

```
for (i = A; i <= B; i++) {
    … X[f(i)]…
    X[g(i)] = …
}
```

To determine independence, the compiler needs to find if for two iterations i1 and i2, (A <= i1, i2 <= B), is there:

```
f(i1) == g(i2)? Or
i2 < i1
```

This can be formulated as an integer linear programming problem and can be applied to a number of DSP codes. The formulation can also be generalized to any level of loop nesting. In practice, this technique provides very fast alias

analysis for loops with few variables, small coefficients and sparse constraints.

2.4 DSP Semantic Analysis

Our compiler analyzes C code and automatically extracts DSP operations using a technique called semantic analysis. In semantic analysis, a sophisticated compiler searches for the meaning of a sequence of C language constructs. The compiler effectively ignores all type information associated with variables and automatically replaces it with DSP types the compiler may then optimize. This technique has a significant software productivity gain over intrinsic functions and does not force the compiler writers to become DSP assembly language programmers.

There are two general styles of coding saturating addition and subtraction: sub-range and full-range. In the sub-range style, overflow and underflow are detected by performing the operation and comparisons using a wider data type. Sub-range saturation also presents a simple pattern and can be detected using existing techniques, such as predicated global value numbering [16].We call this style sub-range saturation. The code below computes 16 bit saturating addition using 32 bit operations (assuming short's are 16 bit wide and int's are 32 bit wide):

```
short addsat16 (short x, short y){
  int z = (int) x + (int) y;
  if (z > MAX16) {
    return MAX16;
  } else if (z < MIN16) {
    return MIN16;
  }
  return (short) z;
}
```

We focus on a second style, where overflow and underflow are detected by testing for inconsistencies between the signs of the operands and the sign of result. We call this style full-range saturation. Usually, this style is used when the data type is the widest that is efficiently supported in hardware. Full-range saturation is not amenable to pattern matching, since there is a variety of ways that it can be expressed. However, the Sandblaster compiler is able to successfully perform semantic analysis and generate DSP instructions for such code, since it does not rely on any pattern matching. Consider an example the following code:

```
#define MAX32 (0x7fffffff)
#define MIN8 (0x80000000)

int addsat_1 (int x, int y) {
  int z = x + y;
  if ( (~x & ~y & z) & MIN32 ) {
    z = MAX32;
  } else if ( x & y & ~z) & MIN32 ) {
    z = MIN32;
  } return z;
}
```

Our compiler can uniquely analyze this code and generate saturating DSP instructions.

3 Parallelizing Optimizations

In this section, we describe parallel optimizations for ILP, DLP and TLP.

3.1 Instruction level parallelism

The scheduling optimizations enable the compiler to exploit instruction level parallelism. The compiler applies software pipelining [17] in combination with aggressive inlining to make use of the multiple operations inside a compound instruction.

3.2 Data-level parallelism (DLP)

The DSP architecture provides vector instructions to exploit data level parallelism. The compiler performs high performance inner and outer loop optimizations that use these vector instructions to exploit the data level parallelism inherent in signal processing applications. These optimizations include vector load, vector store and vector multiply add reduce and saturate. In conjunction with loop optimizations, these provide very efficient and tight loops that can provide more than 16 RISC operations in a single cycle. As an example consider the single instruction, which implements a dot product:

```
L0: lvu %vr0,%r3,8
    ||  vmulreds %ac0,%vr0,%vr0,%ac0
    ||  loop %lc0,L0
```

The above instruction implements an entire loop and contains a vector load with pre-increment (4 16-bit loads with index address update), vector multiply and reduce with

saturation (4-16 bit saturating multiplies and 4 32-bit saturating adds) and a loop instruction (decrement loop count register lc0, compare it against zero and branch to L0).

3.2.1 Vectorizing DSP arithmetic

It is important to note that although saturation operations are non associative (i.e. the order of computation is important), our compiler is able to safely vectorize them because special hardware support guarantees serial semantics of non-associative operations.

3.3 Thread level Parallelism

The programming interface to the resources on the DSP (multiple threads, peripherals, memories, etc.) is provided via POSIX threads[14]. This interface is based on ANSI C and is commonly used in multi threaded/multi processor environments. It provides the ability to create, destroy, and join threads. The underlying operating system supports the ability to prioritize and schedule the software threads. An example of the use of the programming interface is provided below:

```
#include <stdio.h>
#include <pthread.h>
void *kid (void *arg){
    /* print in the kid thread */
    printf("hello world from kid\n");
    pthread_exit(arg);
       return 0;
}

int main(void){
    pthread_t k; void *v;
    /* Create a new thread (kid) */
    pthread_create(&k, NULL, kid, NULL);

    /* print in the main thread */
    printf("hello world from main\n");
    /* wait for the kid thread */
    pthread_join(k, &v);
    return 0;
}
```

This interface can be used to exploit thread level parallelism at a program level, function level or loop level. However, the parallelism at function level or loop level can also be done automatically by the compiler, thus relieving the programmer of this burden.

3.3.1 Open MP

The OpenMP Application Program Interface (API) supports shared-memory parallel programming in C and is portable across architectures[15]. It gives shared-memory parallel programmers a simple and flexible interface for developing multithreaded applications. The compiler supports openMP directives to exploit thread level parallelism in loops and functions.

3.3.2 Thread level parallelism in loops

The many loops of a typical DSP application codes are very compute intensive and generally do not have loop carried dependencies between iterations. The iterations of these loops can be distributed across threads to exploit thread-level parallelism (in addition to instruction level and data level parallelism being exploited inside the loop).

To facilitate the threading process, compiler provides a DOALL pragma. The pragma is inserted by the user (or compiler) above the loop that is a candidate for multithreading. The number of threads to be used may also be specified. A default is used in the absence of a specific quantity of threads. The pragma indicates to the compiler that it is safe to multithread the loop.

It is important to note the multithreaded code generated by the compiler maintains use of the pthreads infrastructure. The pragma merely relieves the programmer of detailed pthreads API usage.

3.3.3 Thread level parallelism in functions

The compiler also provides the ability to execute functions in a program on separate threads via a pragma PARALLEL SECTIONS. The compiler then uses the underlying pthreads implementation to perform the multithreading.

In the following code, functions filter() and descramble() are being executed sequentially.

```
for (i = 0; i < n; i++) {
   filter_result = Filter(input_data[i]);
   Descramble(filter_result);
}
```

However, the by modifying the code so that the filter and despread functions operate on different

sets of data, they can then be executed in parallel..

```
   filter_resm= Filter (input_data[0]);
   for (i = 1; i < n; i++) {
     filter_result = filter_resm;
     Descramble(filter_result);
    filter_resm = filter(input_data[i]);
     }
```

In the above loop, the function descramble() is not dependent on the results of the filter() for the same iteration of the loop. Now a parallel sections pragma can be used to execute the two functions in parallel on separate threads:

```
filter_resm= Filter (input_data[0]);
   for (i = 1; i < n; i++) {
     filter_result = filter_resm;
#pragma PARALLEL SECTIONS
     Descramble(filter_result);
#pragma PARALLEL SECTION
    filter_resm = filter(input_data[i]);
     }
```

3.3.4 Automatic Multithreading

The most significant part of our compiler is that it capable of analyzing complex DSP kernels and using the POSIX pthreads infrastructure to *automatically* generate multithreaded code. Thus, the user is not responsible for specifying any thread-related code for DSP kernels (either pthreads details or the pragma details).

The following is an example of a 20 tap FIR filter, which processes the I & Q streams of data oversampled by 4 for WCDMA downlink filter. It is identically as it appears in the applications code without. There is no programmer use of pragmas nor any other device to specify parallelism.

```
filter(int n)
{
    int i, j;
    int s1, s2;

    for (i=0, k=0; i<n; i++, k+=4) {
      s1 = 0;
      s2 = 0;
      for (j = 0; j < NTAPS; j++) {
       s1 += AI[i+j] * B[j]; //I Samples
       s2 += AQ[k+j] * B[j]; //Q Samples
      }
       DI[i] = s1 >> 16;
       DQ[i] = s2 >> 16;
    }
}
```

In this example, the inner loop (index j) is executed NTAPS times for each iteration of the outer loop (index i). Dependence analysis will show that there is no loop carried dependency in the outer loop. Therefore each iteration of the outer loop (or a set of iterations) can be executed on a separate thread of the processor. In the above code, each thread will execute n divided by the number of threads used per iteration. The following shows the assembly code generated by the compiler for the critical inner loop. This is executed by each thread

```
T0028D9B0:
   vmulred   %ac3,%vr5,%vr7,%ac3 ||
   lvu       %vr5,%r7,8
//A vector MAC and load in parallel
   vmulred          %ac2,%vr6,%vr7,%ac2
   || lvu           %vr6,%r8,8
//A vector MAC and load in parallel
   vmulred          %ac1,%vr5,%vr7,%ac1
//A vector MAC
   vmulred          %ac0,%vr6,%vr7,%ac0
   || lvu           %vr7,%r6,8    ||
   loop             0,%lc1,T0028D9B0
//A vector MAC, load and loop in parallel
```

In the above code, a vector mac is being executed in each cycle. This produces 3.92 taps/cycle (including overhead) out of a theoretical best case 4 taps/cycle.

The automatic multithreading may be applied to all loops in a program. The user can control the threading process by specifying the minimum nesting depth of the loop to be threaded, number of threads to be used on a processor, and the minimum loop count needed before the threading is triggered (to ensure that cost of threading is properly amortized). In absence of any command line user information, the compiler uses built-in heuristics.

3.4 Future Work

Most of our work is focused on enabling more automatic generation and optimization of multithreaded DSP code.

Interestingly, some DSP applications (speech coders, for example [1]) do not exhibit significant data dependence. A program that is data dependent will give significantly different execution times and execution paths through the program depending upon what data input the

program receives. When programs are not heavily influenced by the dataset choice, profile directed optimizations may be effective at improving performance [13].

In profile driven optimization the program is executed based on a set of data inputs. The results of the program and the execution path through the program are then fed back into the compiler. The compiler uses this information to group highly traversed paths into larger blocks of code, often converting variable into constants, which can then be optimized and parallelized.

4 CONCLUSIONS

In this paper we describe the Sandblaster compiler and optimizations that allow for automatic generation of non-associative, vectorized, and fully multithreaded DSP algorithms from a generic ANSI-C description. On a 20-tap WCDMA receive FIR filter with no pragmas nor explicit multithreading, we achieved 3.92 taps per cycle out of a theoretical maximum of 4 taps per cycle on the Sandbridge Sandblaster SB3000 processor.

5 BIBLIOGRAPHY

[1] European Telecommunications Standards Institute, Digital Cellular Telecommunications System, ANSI-C code for the GSM Enhanced Full Rate (EFR) speech codec (GSM 96.53), March, 1997, ETS 300 724.

[2] K.W. Leary and W. Waddington, "DSP/C: A Standard High Level Language for DSP and Numeric Processing", Proceedings of the International Conference on Acoustics, Speech and Signal Processing, IEEE, 1990, pp. 1065-1068.

[3] R. Stallman, "Using and Porting GNU CC", Free Software Foundation, June 1996, version 2.7.2.1.

[4] D. Batten, S. Jinturkar, J. Glossner, M. Schulte, and P. D'Arcy, "A New Approach to DSP Intrinsic Functions", Proceedings of the Hawaii International Conference on System Sciences, Hawaii, January, 2000.

[5] D. Chen, W. Zhao, and H. Ru, "Design and Implementation Issues of Intrinsic Functions for Embedded DSP Processors", in Proceedings of the ACM SGIPLAN International Conference on Signal Processing Applications and Technology (ICSPAT '97), September, 1997, pp. 505-509.

[6] D. Batten, S. Jinturkar, J. Glossner, M. Schulte, R. Peri, and P. D'Arcy, "Interaction Between Optimizations and a New Type of DSP Intrinsic Function", Proceeding of the International Conference on Signal Processing Applications and Technology (ICSPAT '99), Orlando, Florida, November, 1999.

[7] A. Aho, R. Sethi, and J. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley Publishing Company, CA, 1986.

[8] J. Glossner, E. Hokenek, and M. Moudgill, "Multithreaded Processor for Software Defined Radio", Proceedings of the 2002 Software Defined Radio Technical Conference, Volume I, pp. 195-199, November 11-12, 2002, San Diego, California.

[9] J. Glossner, D. Iancu, J. Lu, E. Hokenek, and M. Moudgill, "A Software Defined Communications Baseband Design", IEEE Communications Magazine, Vol. 41, No. 1, pages 120-128, January, 2003.

[10] S. Jinturkar, J. Thilo, J. Glossner, P. D'Arcy, and S. Vassiliadis, "Profile Directed Compilation in DSP Applications", Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT'98), September, 1998.

[11] S. Jinturkar, J. Glossner, E. Hokenek, and M. Moudgill, "Programming the Sandbridge Multithreaded Processor", Proceedings of the 2003 Global Signal Processing Expo (GSPx) and International Signal Processing Conference (ISPC), March 31-April 3, 2003, Dallas, Texas.

[12] T. Boudreau, J. Glick, S. Greene, J. Woehr, and V. Spurlin, "NetBeans: The Definitive Guide", O'Reilly & Associates, Sebastopol, CA, 1st edition (October 15, 2002).

[13] S. Jinturkar, J. Thilo, J. Glossner, P. D'Arcy, and S. Vassiliadis, "Profile Directed Compilation in DSP Applications", Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT'98), September, 1998.

[14] B. Nichols, D. Buttlar, and J. Proulx-Farrell, "Pthreads Programming: A POSIX Standard for Better Multiprocessing", O'Reilly & Associates, Sebastopol, CA, 1st edition (September 1996).

[15] "OpenMP specifications", www.openMP.org

[16] B. Alpern , M. N. Wegman , F. K. Zadeck,"Detecting equality of variables in programs," Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, p.1-11, January 10-13, 1988, San Diego, California, United States

[17] Monica Lam, "Software pipelining: an effective scheduling technique for VLIW machines", Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, Atlanta GA 1988.