Compiler Optimizations with DSP-Specific Semantic Descriptions

Yung-Chia Lin[†] Yuan-Shin Hwang[‡] Jenq Kuen Lee[†]

[†] Department of Computer Science
National Tsing Hua University
Hsinchu 300 Taiwan

[‡] Department of Computer Science
National Taiwan Ocean University
Keelung 202 Taiwan

Abstract. Due to the specialized architecture and stream-based instruction set, traditional DSP compilers usually yield poor-quality object codes. Lack of an insight into the DSP architecture and the specific semantics of DSP applications, a compiler would have trouble selecting appropriate special instructions to exploit advanced hardware features. In order to extract optimal performance from DSPs, we propose a set of user-specified directives called *Digital Signal Processing Interface* (*DSPI*), which can facilitate code generation by relaying DSP specific semantics to compilers. We have implemented a prototype compiler based on the SPAM and SUIF compiler toolkits and integrated the DSPI into the prototype compiler. The compiler is currently targeted to TI's TMS320C6X DSP and will be extended to a retargetable compiler toolkit for embedded systems and System-on-a-Chip (**SoC**) platforms. Preliminary experimental results show that by incorporating DSPI directives significant performance improvements can be achieved in several DSP applications.

1 Introduction

Since high-throughput data stream processing is required to deliver voice, multimedia, and data access in real-time, many embedded systems employ programmable *Digital Signal Processors* (DSPs) as their core components. As DSP applications rapidly grow more complex, high-level development tools are vital to speedy software implementation [18]. However, this approach tends to fail to extract the optimal performance from DSPs owing to the fact that conventional code generation by DSP compilers produces very inefficient results in either speed or code size [20, 8]. Those compilers cannot automatically discover the best ways to utilize the specific architecture features and to unleash proper optimized code composed of custom-built instructions for DSP operations without intervening from the programmers.

[1] The work was supported in part by NSC-90-2218-E-007-042, NSC-90-2213-E-007-074, NSC-90-2213-E-007-075, MOE research excellent project under grant no. 89-E-FA04-1-4, and MOEA research project under grant no. 91-EC-17-A-03-S1-0002 of Taiwan.

This paper proposes a DSP programming model with a set of user-specified directives called the *Digital Signal Processing Interface* (*DSPI*). The key idea is to transform common high-level DSP operations into code of well-regulated vendor-specific instructions. Moreover, we have constructed a prototype compiler targeted to TI's TMS320C6X [16] based on the SUIF [10] and SPAM [9] compiler toolkits, and we are integrating the compiler support for processing the DSPI directives into the compiler. Furthermore, partial support for our objected-oriented architectural description framework (**ORISAL**) is being incorporated into the prototype compiler as well. Preliminary experimental results show that by incorporating DSPI directives significant performance improvement can be attained in several DSP applications. This work is also a series of our research work to attempt to develop high-performance and low-power compiler toolkit for DSP and SoC environments [15, 14, 22, 24, 6, 5].

The rest of this paper is organized in the following. Section 2 discusses the background of the problems of DSP compilation and several related issues. Section 3 details the concepts of DSPI specification and optimization schemes that are furnished to the compiler. Section 4 briefly mentions our DSP compiler constructions and our ADL-based retargetable system toolkit. Experimental results are presented and analyzed in section 5. Section 6 compares the related work and Section 7 concludes this paper.

2 Portability & optimization issues for DSP programming

The market for embedded and SOC systems with DSPs as the central processors has been expanding substantially over the past few years. Whereas there always is a need for greater product differentiation and innovation, there is also an increasing pressure to shrink design cycles by means of streamlining the complicated development process. With an eye to the unceasing upgrades and options of hardware technologies, software development tools should highlight source code portability jointly with the emphasis on target code quality.

Most software development toolkits are designed upon high-level programming languages to insulate programmers from the complexities of DSP architecture. None the less, high-level development tools seldom produce efficient software that can extract the best performance from the advanced hardware. Consequently, most decisive DSP applications are hand-coded in the low-level assembly languages.

Programming DSP applications in assembly languages is fine only when small kernels are required to be executed on a DSP. Today complicated DSP software is required to handle modern applications. Therefore, a procurable DSP software design can be achieved only by means of high-level development tools that can translate high-level languages into efficient object code. In addition, only the restricted features of traditional DSP architecture could make it easy for programmers to hand-optimize code to achieve the desired speed, size, or low-power. As more specialized architectures have been incorporated into current DSP processors, manually performing optimizations is getting exceedingly arduous. For above reasons, there is a great demand for more effective technologies for compiler optimizations of high-level languages. In this paper, we address the direction to develop high-level languages for efficiency. Our approach is to employ DSP-specific semantic descriptions to be annotated on top of high-level languages by DSP programmers. This information can also be picked up by compilers to perform optimizations.

3 Optimizations by DSP specific semantics

In order to facilitate the compilers to generate efficient object code which can take advantage of the DSP advanced hardware features, we propose a set of directives to allow programmers specifying information-rich descriptions. Our DSPI specification covers more general intrinsic operations than existing DSP directives. In the DSPI specification, each directive is specified by the following syntax:

```
#pragma dspi directive-name [clause[ [,]clause]...] new-line
```

Every directive starts with `#pragma dspi`, avoiding conflicting with other pragma directives. The remainder of the directive conforms to the conventions of the C and C++ standards for language directives. DSPI directives are classified into four categories:

- *Environment Depiction*
- *Operation Intention*
- *Data Storage Characterization*
- *Parallelism Commentary*

3.1 Environment Depiction

For each code segment based on the semantic of source code, programmers should specify directives to guide compiler optimizations or to specify user-controlled options. For example, any program following the first directive in Fig. 1 will be translated into a fixed-point code, while statements and variables preceded by the second directive in Fig. 1 will be converted to floating-point instructions and data formats.

```
#pragma dspi algorithm_type fxp
/* indicating the algorithm is designed for fixed point arithmetics */

#pragma dspi algorithm_type fop
/* indicating the algorithm is designed for
   floating point arithmetics              */
```

Fig. 1. Environment Depiction Example 1

In addition to specifying the data precision, the same code running on different DSP processors may actually be compiled with integers of different lengths. Therefore, we need some annotations to assist the compiler in generating code in a bit-accurate way from the portable source. Fig. 2 shows an example. The

```
#pragma dspi int_type 32bit
/* assume that length of type int in the code is designed for 32bit */

#pragma dspi include filename.def
/* include a prepared file of DSPI directives
   at this position in the source code       */
```

Fig. 2. Environment Depiction Example 2

second directive in Fig. 2 will direct the compiler to load a separate file which contains a sequence of DSPI directives.

3.2 Operation Intention

Directives of operation intention provide several levels of comprehensive descriptions to specify DSP-specific computations, essentially ensuring that a compiler will have sufficient knowledge to generate efficient code. By providing those denotational equations or descriptions, different levels of optimizations could be applied, including low-level smart intrinsic instructions substituting, middle-level smart intrinsic function synthesizing, and high-level architectural prefab function embedding as well as possible library-level optimizations.

The general format of the operation intention directives is defined as:

#pragma dspi op *[label:][enum] [&] operation-clause [[;] clause]... new-line*

Two additional directives **op_b** and **op_e** are defined to let programmers specify the starting and end points of operations respectively, if they cannot be easily determined by the compiler:

#pragma dspi op_b *label*
#pragma dspi op_e *label*

op_b and **op_e** DSPI descriptions should be added to demarcate effective boundaries where the semantics of the operation can apply to. Otherwise, the DSPI description **op** will automatically match the following neighboring block structure such as a function or loop if neither **op_b** nor **op_e** is annotated with *label*. The optional *enum* is an enumeration that can be used for supplying more than one alternative solution on the same specified source code. Multiple operator clauses in the same line are separated by semicolons and are operated in sequence by default if they have operand dependencies. Putting a character & before clauses means that the current line is parallelizable.

The most common operations we have seen from studying many kernel routines and DSP-operation emphatic applications such as standard vocoders, which follow the European Telecommunications Standards Institute (ETSI), can be grouped into two levels of semantic operations. The low-level math operators comprise ordinary arithmetic operators such as **.ADD**, **.SUB**, **.NEG**, **.MUL**, and bit-wise manipulation like **.SHL**, **.SHR**. Operator symbol := is used for assigning values, as **.SET** is for setting values between right clause's operands or initializing operands with some values. Rotation operators **.ROR** and **.ROL** are indigenously supported in contrast to original C or C++ because most processors could carry out such operations by built-in hardware implementation. For describing the iterative operations concisely in the inner loops, we introduce the delay operator **.DLY**<*interval*> which could be applied to give the previous value of the operand before an interval.

```
#pragma dspi fxp a:=<16:16>
#pragma dspi fxp b:=<16:16>
#pragma dspi fxp sadd:=<16:16>,saturated
#pragma dspi op sadd:=a.ADD<b>
int sadd(int a, int b)
{
    #pragma dspi fxp result:=<16:16>,saturated
    int result;
    #pragma dspi op saddop: result:=a.ADD<b>
    #pragma dspi op_b saddop
    result = a + b;
    if (((a ^ b) & 0x8000) == 0) {
        if ((result ^ a) & 0x8000) {
            result = (a < 0) ? 0x8000 : 0x7fff;
        }
    }
    #pragma dspi op_e saddop
    return (result);
}
```

Fig. 3. Operation Intention Example

Fig. 3 shows an example of a math operation known as the saturated addition, which can not be easily expressed in C without DSPI directives. In addition, this example shows that multiple descriptions with different boundaries could exist on some overlapped regions as well. Such a style of redundant semantics could be endowed with selectable optimizations if a compiler is able to utilize it.

Other DSP related basic operators are listed in Table 1.

It is straightforward to express critical DSP processing by these lower-level basic operators. All ETSI specific math functions can be described through one or two directives. Furthermore, when directives operate on vectors, they properly manifest simple procurable results despite complex looped source code.

The high-level operation intentions model the familiar DSP actions that could not be directly mapped into hardware instructions but each of them has an uni-

Table 1. Low-level Math Operators

.ABS	absolute value
.RND	round
.NRM	normalize bit number
.XHL<*source operand*>	extract lower half to upper half
.XLH<*source operand*>	extract upper half to lower half
.MUH<*multiplier*>	multiply upper half
.MAC<*multiplier, addend*>	multiply & accumulate
.MSU<*multiplier, subtrahend*>	multiply & subtract
.EXT<*pattern*>	partition and extract section (depend on pattern)
.PAK<*pattern, operand, ...*>	pack multiple operands into one (depend on pattern)

tary thought on vector or matrix operands. One practical example is convolution, which is the same operation as multiplying the polynomials whose coefficients are the elements of two vectors. Several of these high-level DSP operators working on vectors and matrixes are listed in Table 2 and Table 3, respectively.

Table 2. High-level Vector Operators

.CONV<*vector*>	convolve two vectors
.CCRR<*vector*>	cross correlate two vectors
.ACRR	autocorrelate a vector
.RVRS	reverse a vector
.SCAL <*scaling value*>	scale a vector to range
.SUMM	sum a vector
.SQUR	square a vector
.CLRF <*vector*>	generic linear filtering with coefficients
.FIRF <*vector*>	regular FIR filtering operation
.IIRF <*vector, vector*>	regular IIR filtering operation

Table 3. High-level Matrix Operators

.KRONT <*matrix*>	Kronecker tensor product
.INVRT	inverting a square matrix
.LUFAC	compute LU factorization of a matrix
.TRACE	compute the trace of a matrix
.DETRM	compute the determination of a matrix
.EIGEN	compute the eigenvalue of a matrix

3.3 Data Storage Characterization

Some DSP-customary fixed-length data types such as 24-bit and 40-bit integers are not defined in the ANSI/ISO C/C++ standard, and hence extra care must be taken in order to process these data types efficiently. Therefore, the compiler needs more information of data storage in the source code to process the computation and output the accurate data if using the optimized target machine instructions.

Other data storage characterizations which could let the compiler generate codes that benefit from the specific architectural ability are called accessing features. The accessing features for scalar numbers include saturated, several rounding types, etc. Vector or matrix data storage could be additionally appended with accessing conventions like interleaved, divided, circular, and so on. There is also a special accessing feature called temp, which is used to specify a temporary storage location. Suchlike temporary value is just applied to above operation intention directives. The format of data storage characterization is

```
#pragma dspi type id:=[feature[[,]feature]...] new-line
```

Bit-width feature is described as $\langle$width:width$\rangle$ for fixed-point numbers and $\langle$width$\rangle$ for floating-point numbers. Interleaved or divided feature must append a $\langle$number$\rangle$ to characterize the accessing distance. Fig. 4 presents some examples.

```
#pragma dspi fxp a:=<24:16>
int a;
/* declare that int a is actually a fixed-point number
           of bit-width 24 with integer word-length 16 */

#pragma dspi fop b:=<16>,saturated
float b;
/* declare that float b is a 16bit length
          saturated floating-point number */

#pragma dspi cfxp c:=<24:16>,real=cr,image=ci
int cr, ci;
/* declare a temporal complex number c, which consists
 of a real part cr and an imaginary part ci, could be used
      with DSPI operators in representing the semantics */

#pragma dspi fxp d:=<24:16>,circular
int d[160];
/* declare that vector d will be accessed in a circular manner */
```

Fig. 4. Data Storage Characterization Examples

Moreover, data storage characteristics can also be specified to the return values of functions in C or C++ language, as shown in Fig. 3.

3.4 Parallelism Commentary

DSP applications are practically massive data computations. Accordingly, most DSP processors are built as data flow engines, accompanied by a RISC or CISC processor, and consequently they are multiprocessors by default. From these perspectives, the directives defined in this group will allow users to create and enhance DSP programs with possible compiler optimizations when the programs running on an ILP-aware DSP processor or an SIMD-aware DSP processor. In addition, a system with multiple DSP processors may also be improved with those directives if the compiler is designed with suitable optimizations. The directives extend the C and C++ sequential programming model with data parallelism constructs and simultaneous operation semantics [13].

The directive **parallel_region** is designed to specify a parallel region, which is a part of the program that could be executed in parallel if the hardware architecture permits. This directive explicitly directs the compiler to generate parallel code, as shown in Fig. 5. Variables listed in the *shared* clause are accessed

```
#pragma dspi parallel_region [shared:=var1, [var2[,...]]]
{
  /* block of codes */
}
```

Fig. 5. Parallelism Commentary 1

in the same area in parallel, so that the compiler should take a consideration about the data-path conflicting in the specific hardware.

Another fundamental parallelism is shown in Fig. 6. It specifies that the iterations of the loop could be executed in parallel if possible.

```
#pragma dspi parallel_for [schedule:=chunksize]
for(...;...;...)
{
  /* loop body */
}
```

Fig. 6. Parallelism Commentary 2

The optional clause of *schedule:=chunksize* will be used to guide the compiler to generate codes that iterations are divided into chunks of the specified size. Fig. 7 shows a code fragment of LPC analysis with parallelism commentary and other DSPI directives.

4 Compiler framework

This section describes the current status of our compiler environment. It will also give a glance at the involved retargetability technology of architectural description language that we are developing for the embedded system and SoC

```
static void Fast_Autocorrelation P2((s, L_ACF),
    word * s,       /* [0..159] IN/OUT  */
    longword * L_ACF)   /* [0..8]   OUT     */
{
    ...

    #pragma dspi op s_f:=s
    #pragma dspi parallel_for
    for (i = 0; i < 160; ++i)
        sf[i] = s[i];
    #pragma dspi op f_L_ACF:=sf.ACRR.EXT<0:8>
    for (k = 0; k <= 8; k++) {
        register float L_temp2 = 0;
        register float *sfl = sf - k;
        #pragma dspi parallel_for
        for (i = k; i < 160; ++i)
            L_temp2 += sf[i] * sfl[i];
        f_L_ACF[k] = L_temp2;
    }
    scale = MAX_LONGWORD / f_L_ACF[0];
    #pragma dspi op L_ACF:=f_L_ACF.SCAL<scale>
    #pragma dspi parallel_for
    for (k = 0; k <= 8; k++) {
        L_ACF[k] = f_L_ACF[k] * scale;
    }
}
```

Fig. 7. Code Fragment from LPC Analysis in GSM

systems. At present, our prototype compiler is based on the SPAM compiler developed by Princeton University [9]. The front-end of the SPAM compiler is actually using the SUIF1 [10]. SPAM focuses on back-end optimizations and has only a limited set of back-end support for target architectures. In our work, we expand the SPAM to support TI TMS320C6 DSP processors. We exert the *olive* code generator-generator included in the SPAM back-end to perform instruction selections. In addition, we are in the process of incorporating compiler optimizations to fully support the DSPI directives to optimize DSP programs. The code generating procedures and our compiler infrastructures are illustrated in Fig. 8.

In addition to the attempt of supporting compiler optimizations with DSP-specific semantic descriptions specified from DSP programmers, our system is also incorporating a framework of architectural description languages in expectation to provide fast prototype and retargeting of toolkits for DSP processors. Our architecture description language is called (**ORISAL**) denoting *Object-oriented Reconfigurable Instruction Set Architecture Language*. ORISAL is represented by a Java-like syntax. From the internal design of the core library in the ORISAL framework, there are several layers of abstraction that concern the issues of compiler construction and retargeting. One of the components is a uniform semantic behavior interface that is used by both physical components and software components, providing the coherence of operation selection which the compiler or other development tools could use to match maximal architectural features. Among the predefined sets of behaviors of operations, there is one called canonical-semantics that is treated as the fundamental and the least operational interface acknowledged by instructions and hardware components. All operations except canonical-semantics must be applied with at least one self-evident identical operation semantics composed of canonical ones, thus these identical operations could provide multiple cost evaluation of different approaches or decomposed

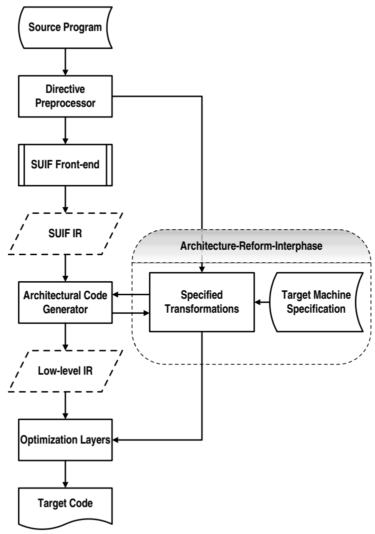

Fig. 8. Compiler Infrastructures

sub-operations that could be scheduled to utilize maximum hardware capability. The rules of completing one operation are configurable from several default built-in operations or user-defined methods. In order to determine the multiple selections of instructions or operation semantics, there are also several ways to reference the related cost functions. The cost of one operation can be chosen from accompanying hardware factors, instruction factors or user-defined compositions. While comparing with other existing ADLs [11] [25] [12], our ORISAL framework has several specialties which could improve the off-the-shelf design process of hardware and associated software development toolkits. ORISAL emphasizes on supporting cycle-accurate simulation, multiple IP descriptions, energy information, and the instruction set architecture for compiler toolkit optimizations.

5 Experimental results

To evaluate the effects of DSPI descriptions for optimization in retargetable compiler construction, we embed *DSPI* in the DSPstone kernel benchmarks [26] and evaluate the performance. The DSP benchmark suite used in this experiments is described in Table 4.

Experiments will be performed on two sets of object programs. The first set of object programs will be generated by compiling the benchmark programs by the compiler in the TI TMS320C6201 development toolkits with option "-O3". In other words, this set is comprised of the optimized object code of the benchmark programs generated without DSPI directives. The performance of this set of object programs will be used as the baseline measurement. The other set

Table 4. DSP Benchmark suite used in this work

Benchmark name	*Description*
convolution	This function behaves as the convolution operation. This is a modified n lags version.
matrix1x3	This program multiplies a 3 by 3 matrix with 1 by 3 vector to generate and store a 3 by 1 vector
lms(least mean square)	This adaptive filter performs a finite impulse response filtering where the tap coefficients are adapted iteratively
fir	One of common kernel operations inside the central applications using DSPs. The FIR benchmark multiplies an array of state variables by an array of coefficients and accumulates the result in the output variable
iir_biquad_N_sections	Another intensively used digital filters in DSP applications. It inputs values through a biquad infinite impulse response filter. The IIR benchmark consists of N serial, second order subsystems named biquads, so that the output of each biquad corresponds tt the input of next biquad
n_complex_updates	This benchmark multiplies n complex numbers with n complex numbers, then adds other n complex numbers, and accumulates the results with n complex numbers updates

of object programs is generated from the benchmark programs with embedded DSPI directives. The prototype compiler tries to select the TMS320C6 intrinsic instructions whenever possible based on the semantics of DSPI directives. As we are still in the early stage of compiler implementations, our software infrastructures are fragile and we manually perform code generation of statements specified by certain DSPI directives for compiler work when our system fails to do that in this early stage. However, we expect the performance will be at least the same when we have the automatic system done, as many of the optimizations with directives are quite syntactic transformations. Both sets of object programs are converted to VLIW code through the TI TMS320C6 assembly optimizer, and then executed on the TI TMS320C6201 software simulator and hardware evaluation module under different settings of data sizes and quality.

The performance of both sets is measured by the the TMS320C6201 profiler. Table 5 and Table 6 list the numbers of cycles are executed for both sets of object programs with the data sizes of 32 bits and 16 bits, respectively. Table 7 presents the execution times when a quadruple of the data is manipulated for the benchmarks, the first two of which use the 32-bit data size and the last two use 16 bits.

Table 5. DSP Benchmark Testsuite 1 (32bit)

Name of benchmark	**convolution**	**matrix1x3**	**lms**	**fir**	**iir_biquad_N_sections**
Original code cycles	417447	21984	110110	55184	36296
DSPI code cycles	352808	18284	106094	48706	34917

Table 6. DSP Benchmark Testsuite 2 (16bit)

Name of benchmark	**matrix1x3**	**iir_biquad_N_sections**	**n_complex_updates**
Original code cycles	19376	24776	94284
DSPI code cycles	15273	23722	54162

The DSPI directives applied on these benchmark programs are concisely described below. Except that using basic operation intentions to construct the

Table 7. DSP Benchmark Testsuite 3 (4x Quantity)

Name of benchmark	lms	fir	iir_biquad_N_sections	n_complex_updates
Original code cycles	413400	194745	82305	371604
DSPI code cycles	374056	150441	60778	194466

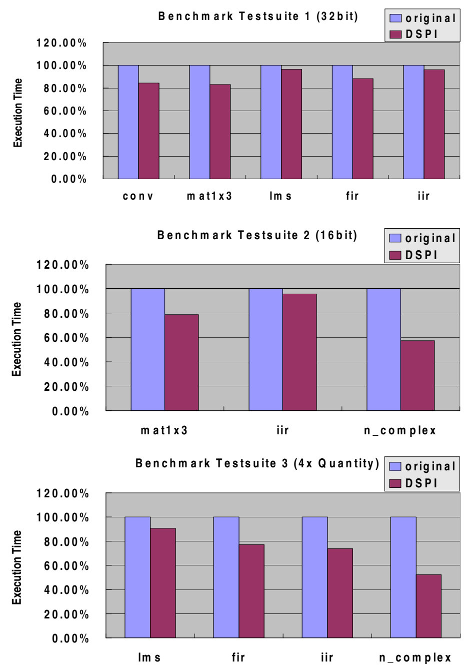

Fig. 9. DSP Benchmark Testsuite Performance Comparison

fundamental semantics, we use the DSPI data characteristics to indicate the applied vector and matrix operations. Besides, in **convolution**, **matrix1x3**, and **n_complex_updates**, we use parallelism commentary directives to annotate the loops; in **n_complex_updates**, we insert complex number feature to precisely define the data type; in the **iir_biquad_N_sections**, we make use of special storage addressing by circular buffer. These tables show that we can get some notable improvement by embedding the DSPI directives. Fig. 9 depicts the performance improvement over normalized baseline execution times.

In addition to DSP kernel program benchmarks, we also take a large application program to test the DSPI optimizations. The application is the full G.723.1 codec from ITU-T, and is compiled with both portions of rate 6.3kbit/s and rate 5.3kbit/s into single object program to simulate the condition that complex and large code could not entirely reside in the DSP core memory. The performance of coder and decoder in each rate is listed in Table 8. The comparison figures are illustrated in Fig. 10 and they reveal significant improvements gained by DSPI descriptions.

Table 8. Large Application Benchmark (G.723.1 Codec)

Name of benchmark	**Coder 6.3K**	**Coder 5.3K**	**Decoder 6.3K**	**Decoder 5.3K**
Original code cycles	8323070	11751868	850180	812517
DSPI code cycles	1298805	1565469	93190	89884

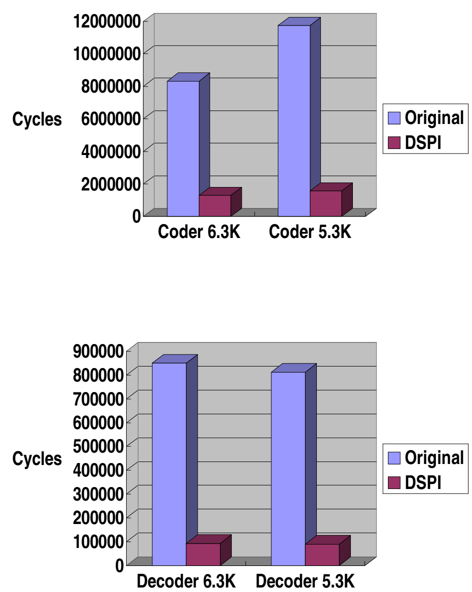

Fig. 10. Full G.723.1 Codec Benchmark Performance Comparison

6 Related works

Several methods are currently used for DSP optimizations. First, it uses the support of libraries for those frequent computation-intensive kernel procedures include FIR filters, IIR filters, FFTs, DCTs, and so on. Another way for DSP optimizations is to use intrinsics for describing specific instructions available in an architecture [2, 3, 7]. Rudimentary intrinsics is like the assembly statement but have the appearance of macro-usage in C/C++ source code. These words are transformed to natively supported instructions during compilation and are rewarded for some tier of optimizations. The third method is modifying the language itself to force programmers explicitly writing DSP architectural data types and expressions of DSP functionalities. Language extensions for this purpose are designed with enhanced constructs like fixed-point types, saturation types, particular memory addressing types, simultaneous execution model, etc [17, 19, 21]. These additions often denote special compilers that accept these none-standard dialects of C/C++ language. This decreases the portability of the programs.

Our compiler directive approach is similar to HPF [4] and OpenMP [1] for the methodology, but specialized in DSP optimizations.

7 Conclusion

In this paper, we offered the possible solution with DSPI, a set of DSP specific semantics directives, which could make the compiler produce more optimized codes but persist source-level compatibility. We also have briefed the on-going research works of our proposed retargetable toolkit framework of SoC, ORISAL, in which the DSPI could be entailed to perform a retargetable optimization layer of the compiler and may benefit with both portability and optimizations. Preliminary experiments reveal the promising prospects of continuation in this research work.

References

1. Eduard Ayguade, Xavier Martorell, Jesus Labarta, Marc Gonzalez, and Nacho Navarro. Exploiting multiple levels of parallelism in openmp: A case study. In *International Conference on Parallel Processing*, pages 172–180, 1999.
2. D. Batten, S. Jinturkar, J. Glossner, M. Schulte, and P. D'Arcy. A new approach to dsp intrinsic functions. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 908–918, January 2000.
3. 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. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT'99)*, November 1999.
4. Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S.Ranka. Fortran 90d/hpf compiler for distributed memory mimd computers: design, implementation and performance results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.
5. R. G. Chang, T. R. Chuang, and Jenq-Kuen Lee. Efficient support of parallel sparse computation for array intrinsic functions of fortran 90. In *Proceedings of ACM International Conference on Supercomputing*, July 1998.
6. Rong-Guey Chang, Jia-Shing Li, Tyng-Ruey Chuang, and Jenq Kuen Lee. Probabilistic inference schemes for sparsity structures of fortran 90 array intrinsics. In *Proceedings of the 2001 International Conference on Parallel Processing*, September 2001.
7. D. Chen, W. Zhao, and H. Ru. Design and implementation issues of intrinsic functions for embedded dsp processors. In *Proceedings of the ACM SIGPLAN International Conference on Signal Processing Applications and Technology (ICSPAT'97)*, pages 505–509, September 1997.
8. Naji Ghazal, Richard Newton, and Jan Rabaey. Predicting performance potential of modern dsps. In *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, June 2000.
9. SPAM Research Group. *SPAM Compiler User's Manual*, September 1997. *http://www.ee.princeton.edu/spam/*.
10. Stanford Compiler Group. *The SUIF Library*, 1994. *http://suif.stanford.edu/suif/suif1/docs/suif_toc.html*.

11. G. Hadjiyiannis, S. Hanono, and S. Devadas. Isdl: An instruction set description language for retargetability. In *Proceedings of ACM/IEEE Design Automation Conference*, October 1997.
12. Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design, Automation, and Test in Europe conference (DATE)*, 1999.
13. Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. Integrating automatic data alignment and array operation synthesis to optimize data parallel programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing (LCPC'97)*, Augest 1997.
14. Gwan-Hwan Hwang, Jenq Kuen Lee, and Roy Dz-Ching Ju. A function-composition approach to synthesize fortran 90 array operations. *Journal of Parallel and Distributed Computing*, 54:1–47, 1998.
15. Yuan-Shin Hwang, Peng-Sheng Chen, Jenq-Kuen Lee, and Roy Ju. Probabilistic points-to analysis. In *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, August 2001.
16. Texas Instruments Incorporated. *TMS320C62x/C67x CPU and Instruction Set.* Texas Instuments Incorporated, 1998. *http://www.ti.com/sc/psheets/spru189d/spru189d.pdf.*
17. M. Jersak and M. Willems. Fixed-point extended c compiler allows more efficient high-level programming of fixed-point dsps. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT'98)*, October 1998.
18. J.Glossner, D. Routenberg, E. Hokenek, M. Moudgill, M. Schulte, P. Balzola, and S. Vassiliadis. Towards a very high bandwidth wireless handheld device. Technical report, Sandbridge Technologies, Inc., 2001. White Paper.
19. B. Krepp. Dsp-oriented extension to ansi c. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT'97)*, pages 658–664, 1997.
20. Ashutosh K. Kulkarni and Aditya Dube. Benchmarking code generation methodologies for programmable digital signal processors. April 1997.
21. K. Leary and W. Waddington. Dsp/c: a standard high level language for dsp aad numeric processing. In *Proceedings of the International Conference on Acoustic, Speech, and Signal Processing*, pages 1065–1068, 1990.
22. Chingren Lee, Jenq Kuen Lee, TingTing Hwang, and Shi-Chun Tsai. Compiler optimization on instruction scheduling for low power. In *Proceedings of the 13th International Symposium on System Synthesis*, pages 55–60, September 2000.
23. The vice president at Motorola. Paul Marino. Optimizing dsps for wireless world, April 2001. Online Issue by eMedia Asia Ltd.
24. Yi-Ping You, Ching-Ren Lee, Jenq-Kuen Lee, and Wei-Kuan Shih. Real-time task scheduling for dynamically variable voltage processors. In *Proceedings of IEEE Workshop on Power Management for Real-Time and Embedded Systems*, May 2001.
25. V. Zivojnovic, S. Pees, and H. Meyr. Lisa – machine description language and generic machine model for hw/sw co-design. In *Proceedings of IEEE Workshop on VLSI Signal Processing*, October 1996.
26. V. Zivojnovic, J.M. Velarde, C. Schlager, and H. Meyr. Dspstone, a dsp-oriented benchmarking methodology - final report. Technical report, Aachen University, Germany, August 1994. Technical Report.