

# A New Approach to DSP Intrinsic Functions

Dean Batten, Sanjay Jinturkar, John Glossner,  
Michael Schulte and Paul D’Arcy

**Abstract**—A compiler is frequently unable to make use of algorithm-specific instructions which DSP’s (digital signal processors) provide. To force their use, programmers use language extensions to define intrinsic functions. Traditionally, these intrinsic functions are implemented with assembly language instructions on the target processor. We propose and implement a new approach to intrinsic functions where the programmer targets a compiler’s intermediate representation rather than the assembly language of a particular processor. The benefits of our approach are both portability and improved performance. We compare the performance of the traditional approach with our new approach for four GSM (Global System for Mobile communication) speech coders. We also examine the interaction of our new approach with two major optimizations—profile-directed function inlining and aggressive loop optimization. We find that, compared to the traditional approach, ours benefits greatly from removing barriers to these optimizations. We find that without inlining or loop optimization, our new approach speeds up execution by a factor of 1.12 compared to the traditional approach; however, with both loop optimization and a moderate amount of inlining, we find a speedup of 2.38. We find a similar improvement in achieved instruction-level parallelism.

**Keywords**— Speech coders, intrinsic functions, profile-directed function inlining, loop optimization, software pipelining, modulo scheduling, speedup, code growth, instruction-level parallelism, performance analysis.

## I. Introduction

ANSI C is the most commonly used language for digital signal processor (DSP) programming, but it is not the best language for this task [1][2]. It lacks ways to specify several types of DSP-specific computations, essentially ensuring that a compiler will not generate efficient machine code.

Intrinsic functions address some of these problems, allowing a programmer to specify the use of specific

Dean Batten ([batten@lucent.com](mailto:batten@lucent.com)) is with Lucent Technologies and Lehigh University. Sanjay Jinturkar and Paul D’Arcy are with Lucent Technologies. John Glossner is formerly of Lucent Technologies, now with IBM Research. Michael Schulte is with Lehigh University.

lower-level instructions. Traditionally, intrinsic functions have been implemented with assembly instructions. Our new approach to intrinsic functions implements them with instructions selected from the compiler’s intermediate representation.

Our results show that our new approach to intrinsic functions is a significant improvement over the traditional method. Besides being portable, it provides substantial increase in performance by providing more opportunities for optimization and ILP (instruction-level parallelism) compared to the traditional method.

**Definitions**— An *intrinsic function* has the appearance of a function call in C source code, but is replaced during pre-processing by a programmer-specified sequence of lower-level instructions. The replacement specification is called the *intrinsic substitution* or simply the *intrinsic*. An intrinsic function is *defined* if an intrinsic substitution specifies its replacement. The lower-level instructions resulting from the substitution are called *intrinsic instructions*.

## A. Motivation for intrinsics

Figure 1 shows `L_add`, a function used by GSM (Global System for Mobile communication) speech coders which performs a saturating add of two 32-bit signed 2’s-complement integers. Lines 3 and 4 check if saturation is necessary, and line 5 provides the correct saturated value. Since the most negative 2’s-complement value `MIN_32` is `0x80000000` (all bits are zero but the sign bit) line 3 checks if the input parameters have the same sign; if not, saturation is never necessary. Line 4 checks if the result of the addition and one of the inputs (either one can be used) have the

```
0 Word32 L_add (Word32 a, Word32 b) {
1     Word32 c;
2     c = a + b;
3     if (((a ^ b) & MIN_32) == 0) {
4         if ((c ^ a) & MIN_32) {
5             c = (a < 0) ? MIN_32 : MAX_32;
6         }
7     }
8     return (c);
9 }
```

Fig. 1. Saturating add function used in GSM coders.

same sign; if not, given that both of the input parameters have the same sign, overflow has occurred, and saturation is necessary. Line 5 provides the correct saturated value based on the sign of one of the inputs.

The `L_add` function, as expressed in C, requires three branches, four logical operations, and an addition, besides function call overhead. In contrast, DSP's typically have a single instruction which can perform the saturating add in one cycle; however, this instruction cannot be generated by the compiler automatically. In order to specify the use of the saturating add instruction, a DSP programmer defines `L_add` as an intrinsic function.

The intrinsic substitution has two immediate performance benefits. First, since the intrinsic instruction(s) are more efficient than the sequence of instructions generated by the original C source, their use should reduce both instruction and cycle counts. Second, function call overhead is eliminated. For DSP's, the use of intrinsic functions often makes the difference between real-time and non-real-time performance on a given processor.

Ideally, a compiler could synthesize DSP instructions, such as a saturating add, as a peephole optimization, but we are not aware of any compilers which do this. The problem of recognizing the patterns of these instructions is part of the overall instruction selection problem; this problem is often approached as selecting a minimal-cost covering of a program DAG (directed acyclic graph). Aho's dynamic programming algorithm [3] solves a restricted version of this problem and is used in many compilers including the DSP compiler CodeSyn [4]. Liao gives an overview of work on the more general problem [5]. CodeSyn provides an example of why automatic recognition of instructions like saturating addition is so difficult. CodeSyn stores two types of patterns for representing instructions—control-flow patterns and data-flow patterns. Even supposing that a saturating add were always expressed the same way in C, recognition of it would require a simultaneous match of both a complex control-flow pattern and a complex data-flow pattern.

## II. Related work

Traditionally, intrinsic functions have been implemented with assembly language instructions. Most C compilers provide an `asm` statement for generating assembly instructions directly. When used for implementing intrinsic functions, these assembly statements are used as macro replacements for C function calls, are maintained through intermediate representations, and specify the generation of a particular sequence of assembly code. While providing the performance ad-

```
#define L_add(a,b) \
({int __value, __arg1 = (a), __arg2 = (b);\
asm ("add_sat %2,%1,%0": "=d" (__value):\
"d" (__arg1) : "d" (__arg2)); \
__value; })
```

Fig. 2. A GCC `asm` intrinsic substitution for `L_add`.

vantages described above, this method of specifying intrinsic functions has several disadvantages.

Figure 2 is an example of a GCC [6] `asm` intrinsic substitution for the `L_add` function, for a target machine with an `add_sat` instruction which performs a saturating add. This `asm` statement defines two integer input registers `__arg1` and `__arg2`, an integer output register `__value`, and a template for insertion of the register references into an assembly instruction—`add_sat %2,%1,%0`. GCC makes no attempt to interpret the assembly mnemonic itself. The assembly language template may contain multiple instructions. In our example, each C reference to `L_add` eventually becomes a single `add_sat` assembly instruction.

There are two disadvantages to using assembly instructions for implementing intrinsic functions. First, assembly instructions are not portable. Of course, using machine-specific code can be extremely effective for performance; with sufficient knowledge of the target machine, a programmer can construct an optimal set of assembly instructions. However, obtaining this level of performance comes at the expense of portability; even for an object code compatible, higher issue rate machine, it is likely the assembly instructions would need to be rewritten in order to further reduce cycle counts.

Second, assembly instructions inhibit many optimizations—most of all instruction scheduling. One simple example of this is that the scheduler cannot account for any latencies of instructions in a GCC `asm` statement, since the scheduler treats it as a block of unknown assembly instructions. Effectively, the entry to and exit from each assembly statement is an optimization barrier. In the context of a multiple-issue processor, the scheduler cannot schedule any other instructions concurrent with the ones in an assembly statement, since it does not know which processor resources they will consume. Also, for an `asm` statement with more than one instruction, the statement must remain a block; none of the individual instructions can be promoted or demoted in the overall schedule to fill delay slots or overlap execution with other instructions. Finally, the compiler should assume that assembly instructions could have side effects which change the state of the machine (a programmer may use them for

exactly that purpose, such as to change a saturation or rounding mode), and so should not move instructions over the assembly instructions.

Besides scheduling, other optimizations are impeded by the barriers imposed by assembly instructions. Instructions before and after assembly instructions cannot be combined. Also, since the assembly instructions are opaque to the compiler and force the placement of operands in registers, the instructions cannot participate in many optimizations, including constant propagation and common subexpression elimination.

Several compilers implement intrinsic functions by promoting assembly instructions to C keywords. Texas Instruments' TMS320C6x compiler [7], for example, has intrinsic instructions which are a subset of their 'C6x instruction set [8]. As with `asm` statements, a programmer using this type of intrinsic function implementation is still writing for a particular architecture.

Many vendors provide optimized, target-dependent libraries for important functions. If libraries are provided for all targets the compiler supports, then this approach provides some portability. However, if libraries are used to implement intrinsic functions, then intrinsic instructions are embedded in a function call, and many of the same barriers to optimizations exist as with assembly instructions.

### III. Our method

The traditional method of implementing intrinsics is with assembly instructions. Our approach to intrinsics intelligently handles a set of instructions taken from the compiler's intermediate representation rather than from a target assembly language. Using this approach, we achieve both portability and improved performance.

#### A. Analysis of traditional method

Section II discussed the disadvantages of implementing intrinsics using assembly instructions: they introduce architectural dependence and inhibit optimizations, especially instruction scheduling. These disadvantages have a common cause: the compiler does not understand the assembly instructions. Therefore, it must make nearly worst case assumptions, partitioning the assembly instructions from surrounding code.

If the compiler understood the semantics and side effects of the computations described by the assembly instructions, it could avoid their disadvantages. Toward this end, either the compiler could interpret the assembly instructions, or the computations could be described by some language other than assembly, but still sufficiently close to it to describe the computations efficiently. An intermediate approach would solve one of the disadvantages: `asm_safe` statements could be de-

finied which convey that there are no side effects; this would allow optimizers to move instructions over the `asm_safe` statements, but other disadvantages would remain.

Teaching the compiler to read assembly language is an architecture-dependent task; this is the approach is taken by assembly-language optimizers. Instead, we take the second approach, and describe the computations in an architecture-independent form. Specifically, we promote specific instructions from our compiler's low-level IR (intermediate representation) to new C keywords.

#### B. Our implementation

Our approach to intrinsics intelligently handles a set of intrinsic instructions, avoiding all three of the disadvantages of using assembly instructions as intrinsic instructions. Our intrinsic instructions are selected instructions from the compiler's intermediate representation rather than from a target assembly language; these instructions become a set of new C keywords.

We avoid the optimization disadvantages of assembly language intrinsic instructions because the compiler understands the meaning of the intrinsic instructions. They are not unknown operations but specific known ones. Furthermore, our approach is architecture-independent, since our intrinsic instructions do not specify assembly language instructions in a target architecture but instructions in the compiler's intermediate representation.

A programmer uses our intrinsics by specifying a substitution such as:

```
#define L_add(a,b) (_add_sat(a,b)).
```

The `_add_sat` instruction specified is an instruction in our compiler's low-level IR, not necessarily an instruction in the assembly language of the target architecture.

The first advantage of writing intrinsic substitutions to the compiler's IR is that intrinsics as such disappear in the IR. Instructions in the IR are equivalent whether they resulted from standard C statements or intrinsic substitutions; intrinsic instructions need not be handled in an extraordinary way. They can fully participate in optimizations with non-intrinsic instructions. Whether instructions are generated by intrinsic substitutions, peephole optimizations or some other method is of no consequence. (We think the IR of a DSP compiler should include DSP-oriented instructions; while this complicates the IR, it allows for better quality output.)

The second advantage of writing intrinsic substitutions to the compiler's IR is that the program remains portable. The target architecture can be changed without any modification to the intrinsic substitutions used,

and the compiler can accommodate the change (provided that the compiler has been ported to the target architecture). If the target architecture supports an intrinsic instruction directly, it can be used. If not, it can be expanded as necessary, perhaps even making a function call to a library; in this case, the speedup from using an intrinsic substitution is negated. We used the function-call strategy to emulate our intrinsic instructions for HP and Sun workstations to verify the correctness of our work.

The ability to target different architectures without modifying the intrinsic substitutions used, while benefiting from those intrinsics when the architecture has relevant support, is a significant benefit to an embedded system programmer or designer. Implementing intrinsic functions with instructions from the compiler’s IR rather than instructions from a particular architecture’s assembly language is a significant step toward the goal of a high-performance, retargetable DSP compiler.

There are two disadvantages of writing intrinsics to the compiler’s IR. First, the intrinsic substitutions are now compiler-dependent; however, this is also true of many `asm` statements. Second, some of the IR is exposed to the programmer. Our implementation exposes only DSP-specific instructions which are difficult to generate from C. An alternative approach is to construct a programmer interface describing allowable intrinsic instructions, which may or may not be directly implemented in the IR; as long as the compiler handles them intelligently and is able to isolate them from the target architecture, all of the same advantages can be maintained.

## IV. Experiments

We claim a performance advantage over traditional assembly language intrinsics, in addition to portability. To validate our claims, our experiments compare the performance of traditional intrinsics with our approach, and compare the interactions of each type with aggressive ILP-improving optimizations.

Since the main performance benefit of our approach comes from removing optimization barriers, we call the traditional intrinsics *blocking intrinsics* and our approach *non-blocking intrinsics*.

The two aggressive optimizations we study are profile-directed function inlining and aggressive loop optimizations. We choose these because we expect the combination of non-blocking intrinsic substitutions and inlining of functions in inner loops to create better candidates for aggressive loop optimization.

## A. Benchmarks

We examine four ETSI GSM (European Telecommunication Standards Institute Global System for Mobile communications) speech applications: the half-rate encoder and decoder [9], and enhanced full-rate encoder and decoder [10]. These four applications are important for GSM cellular systems. All of these applications share ETSI-defined basic operations for emulation of fractional arithmetic using integer arithmetic. The intrinsic functions we define are among the most often called of these basic operations.

Each ETSI application has a series of test vectors associated with it. In addition to the variables described below, we could examine the performance variations among test vectors and average over them, but we do not; our previous research has shown the performance of our compiler on these ETSI applications to be relatively insensitive to the test vector used for profiling. The performance difference between code profiled and run with the same test vector and profiled and run with different test vectors is in the range of 1–5% [11]. Each test vector consists of a number of speech frames, and we choose, for each application, the test vector with the single most computationally intense frame.

## B. Target

The target of all of our experiments is an ideal 8-issue VLIW processor. It is capable of only one branch per cycle but otherwise can execute any combination of instructions; every execution unit is capable of executing any arithmetic or memory instruction. It has 64 general-purpose registers and a two-cycle memory latency. It has a perfect instruction cache, but no data cache; typical DSP’s do not have a data cache, but use fast on-chip SRAM memory. Like most DSP’s, it has a single-cycle multiplier and saturating arithmetic instructions, but unlike many DSP’s, it does not have multiply-shift or multiply-accumulate instructions. Our VLIW processor does not require filling of empty slots with no-op instructions. This target is intended to resemble new, wide DSP’s such as the Star\*Core 140 and the Texas Instruments ‘C6x.

## C. Measurements

For each trial, we measure three things—code size as a static count of instructions, instructions executed as a dynamic count, and the cycle count for completion. The baseline measurement for each benchmark is with no intrinsics, no inlining, and no loop optimization. *Speedup* is computed as cycle count divided by baseline cycle count. *Code growth* is computed as code size divided by baseline code size. *IPC* (instructions per cycle) is computed as dynamic instruction count

Number	Weight	Function	Substitution
1	0.2191	L_mult	mul, asl_sat
2	0.1613	L_add	add_sat
3	0.0549	L_sub	sub_sat
4	0.0433	L_shl	asl_sat
5	0.0213	saturate	sat
6	0.0209	mult	mul, asl_sat, asr
7	0.0177	shr	asl_sat, sat
8	0.0151	extract_h	lsr, extract16
9	0.0110	extract_l	extract16
10	0.0094	L_shr	asl_sat
11	0.0032	shl	asl_sat, sat
12	0.0018	norm_l	norm
13	0.0001	L_abs	abs_sat
14	0.0001	abs_s	lsl, abs_sat, lsr
15	0.0001	norm_s	norm, sub
	0.5794	Total	

Fig. 3. Intrinsic substitutions used. *Weight* is the fraction of total execution time when intrinsics are not used, as measured by `gprof`, average over all benchmarks.

divided by cycle count. A geometric mean over all of the benchmarks is computed for speedup, code growth, and IPC.

## D. Intrinsics

We define intrinsic substitutions for functions in our benchmarks based on two criteria. First, we only define an intrinsic when there is clear gain from it (i.e., there must be obvious DSP-specific instructions for implementing a given function). Second, we only define intrinsics for functions representing more than 0.01% of the program execution time. Based on these criteria, we use the intrinsic substitutions shown in Figure 3. We apply them in the order shown in the figure: for example, when we use three intrinsic functions, they are `L_mult`, `L_add`, and `L_sub`. We vary intrinsic substitutions applied from 0 (none) to 15 (all), and we apply both blocking and non-blocking varieties of the intrinsics.

Other common DSP intrinsic instructions, such as modulo addressing or bit-reversed addressing can be handled by our mechanism. However, since these are not used by our benchmarks, we do not discuss them here.

## E. Other optimizations

The use of non-blocking intrinsics, in addition to direct benefits, has indirect benefits, because of interactions between intrinsics and other optimizations. The two optimizations we isolate in this study are profile-

directed inlining and aggressive loop optimizations. Other standard optimizations are always used [3].

The use of non-blocking intrinsics has substantial indirect benefits, similar to those provided by function inlining. Since we replace function calls with simple instructions, each replacement removes a barrier to optimization. Loops containing function calls that are replaced by intrinsic instructions become better candidates for unrolling or modulo scheduling optimizations. These benefits point toward increased instruction-level parallelism.

### E.1 Profile-directed inlining

Our compiler applies profile-directed function inlining [12]. Profile information is collected by instrumenting the benchmark code with probes, compiling, and running with a training input. Then, at a high-level intermediate code, starting with the most-often-reached call site, functions are inlined, provided they meet a set of criteria. Inlining is controlled by the amount of code expansion allowed across the whole application. A code expansion ratio is selected and function calls are inlined until the code size exceeds the allowed size; a ratio of 1.2 indicates that a 20% overall expansion is allowed. We vary inlining from a ratio of 1.0 (none) to a ratio of 2.6.

### E.2 Loop optimization

We examine the effect of aggressive loop optimization—modulo scheduling [13] and loop unrolling. When a loop can be modulo scheduled, we do so. When it cannot, we unroll it. We favor modulo scheduling over the relatively large code expansion of loop unrolling; however, since modulo scheduling has limitations on the loops it will schedule, we can apply unrolling to more loops. We examine two cases—with and without aggressive loop optimization. Whether or not aggressive loop optimization is applied, we perform standard loop optimizations [3].

## V. Results

We present our results in two sections. The first section compares the performance of blocking and non-blocking intrinsics without the application of function inlining or aggressive loop optimization, and the second section examines the interactions of these optimizations with blocking and non-blocking intrinsics.

### A. New versus old intrinsics

Figures 4 and 5 show speedup and IPC provided by both blocking and non-blocking intrinsics. The application of all 15 blocking intrinsics results in a speedup

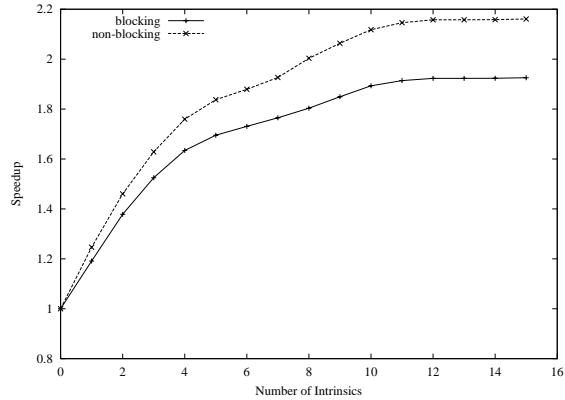


Fig. 4. Speedup versus number of intrinsics for blocking and non-blocking intrinsics. Mean of benchmarks, no inlining expansion, no loop optimization.

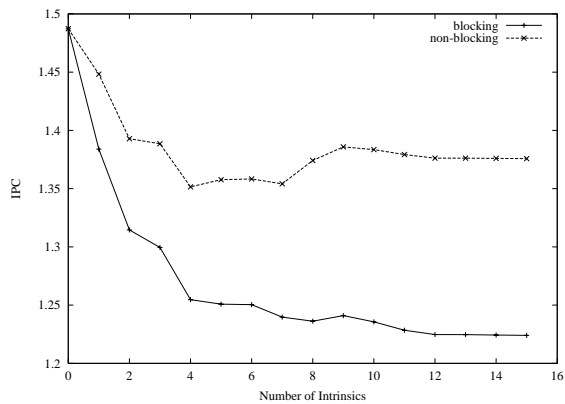


Fig. 5. IPC versus number of intrinsics for blocking and non-blocking intrinsics. Mean of benchmarks, no inlining expansion, no loop optimization.

of 1.93, and the application of all 15 non-blocking intrinsics results in a speedup of 2.16, an improvement of 12%. The application of all 15 blocking intrinsics results in an IPC of 1.22, and the application of all 15 non-blocking intrinsics results in an IPC of 1.38, an improvement of 13%.

IPC is lowered by application of intrinsics, especially for blocking intrinsics. This is because the application of intrinsics lowers both the number of instructions executed and the cycle count, but lowers the number of instructions executed proportionally more. With blocking intrinsics, this effect is exaggerated. The next section will show that the use of inlining and loop optimizations improves IPC for non-blocking intrinsics.

Figures 6 and 7 show the schedule improvement obtained in one critical inner loop. With blocking intrin-

Cycle	Instructions
1	lsl
3	load
3	—
4	add
5	sat (1)
6	load
7	—
8	lsl
9	asl_sat (2)
10	add_sat (3)
11	load
12	—
13	lsl
14	asl_sat (4)
15	add_sat (5)
16	mul
17	and
18	asr
19	and
20	bne
21	sat (6)
22	add_sat (7)
23	lsr
24	extract
25	mul
26	asl_sat (8)
27	mul
28	asl_sat (9)
29	sub_sat (10)
30	add            bgt
31	extract
32	mov            blt

Fig. 6. Schedule for inner loop with blocking intrinsics. Ten intrinsic substitutions occur in the loop.

Cycle	Instructions
1	lsl            add
2	load            load            load            extract
3	—
4	lsl            add            lsl
5	asl_sat (2)    sat (1)        asl_sat (4)
6	add_sat (3)    mul
7	add_sat (5)    and
8	add_sat (7)    asr
9	lsr            sat (6)        and
10	extract        mul            bne
11	mul            asl_sat (8)
12	asl_sat (9)
13	sub_sat (10)
14	bgt            mov
15	blt

Fig. 7. Schedule for inner loop with non-blocking intrinsics. Ten intrinsic substitutions occur in the loop.

sics, the loop body is scheduled for 32 cycles at an IPC of 0.97. With non-blocking intrinsics, the loop body is scheduled for 15 cycles at an IPC of 2.07. The annotations on the figures show the overlapped scheduling of intrinsic instructions.

## B. Other optimizations

The following paragraphs present our results for speedup, code growth and IPC versus number of intrinsics, inlining expansion ratio, and loop optimization for our benchmarks. Since the graphs cannot show all variations simultaneously, they are drawn along axes we consider most likely for production use—all intrinsics, 1.4 inlining expansion ratio, with loop optimization.

### B.1 Speedup

Figures 8 through 13 show our speedup results. Speedup is shown relative to a baseline with no intrinsics, no inlining, and no aggressive loop optimization.

The highest mean speedup we observe is 6.91. This occurs with all non-blocking intrinsics, maximum inlining, and loop optimization. With blocking intrinsics, this speedup is 2.83, so non-blocking intrinsics produces a factor of 2.44 improvement.

As Figure 8 shows, the addition of intrinsics produces a steady increase in speedup. The increase is often not in proportion with the weight of the intrinsic (see Figure 3)—intrinsic 6 (2% weight) shows very little increase, while intrinsic 10 (< 1% weight) shows a large increase. This shows that the increase in speed is due as much to the removal of barriers to optimizations as to the reduction in execution time of the intrinsic functions themselves.

As Figure 9 shows, even a small amount of inlining allows intrinsics to contribute significantly to the overall speedup. This figure shows that with all intrinsics and loop optimization, increasing the inlining expansion ratio from 1.0 (none) to 1.05 increases speedup from 2.08 to 4.55. This trend results from often-called functions which themselves contain calls to intrinsic functions. For example, a function `L_mac` is one of the most often called in our GSM benchmarks, and this function is short, calling only `L_mult` and `L_add` in sequence. Once `L_mult` and `L_add` are implemented as intrinsics and `L_mac` is inlined at call sites in loops, these loops become excellent candidates for optimization.

As Figures 10 and 13 show, non-blocking intrinsics make a huge difference when combined with loop optimization. For all intrinsics and maximum inlining, blocking intrinsics only allow loop optimization to increase speedup from 2.57 to 2.83, an increase of 10%, while non-blocking intrinsics allow loop optimization to increase speedup from 3.85 to 6.91, an increase of 79%.

As Figures 8 and 11 show, all benchmarks respond similarly to the application of intrinsics and inlining.

### B.2 Code growth

For the sake of space, we do not report detailed code growth results here. Typically, increasing the number of intrinsics reduces code size, offsetting some expansion due to inlining expansion and loop optimization. For our typical case of all non-blocking intrinsics, 1.4 inlining expansion ratio and loop optimizations, the total code growth is 27%.

### B.3 IPC

Figures 14 through 19 show our instruction-level parallelism (ILP) results, measured in instructions per cycle (IPC). For all benchmarks, the baseline IPC's (no intrinsics, no inlining, and no loop optimization) are approximately 1.5. All of our graphs and discussion cite absolute IPC, not relative IPC.

The highest IPC we observe for the benchmark mean is 3.79, where the maximum possible with our target architecture is 8. This occurs with all non-blocking intrinsics, maximum inlining, and loop optimization. If blocking intrinsics are used, the IPC is 1.29 (less than baseline), so non-blocking intrinsics produce a factor of 2.94 improvement over blocking intrinsics. With a more moderate inlining expansion ratio of 1.4, the IPC for non-blocking intrinsics is 3.68, while the IPC for blocking intrinsics remains 1.29. Our non-blocking approach interacts very well with other optimizations to improve IPC, while the blocking approach does not.

Figures 14 through 16 show that increasing application of non-blocking intrinsics increases IPC steadily and significantly, when inlining and loop optimization are performed. Figures 14 and 16 show that the increasing application of blocking intrinsics reduces IPC even when inlining and loop optimization are performed.

Figures 17 through 19 show that increasing inlining increases IPC greatly at first, but less for larger inlining ratios; this is with non-blocking intrinsics and loop optimization. Even a small amount of inlining produces large gains in IPC, just as it produced large gains in speedup, and for the same reason. Figure 18 shows that the more non-blocking intrinsics used, the more difference increased inlining makes. On the other hand, Figures 17 and 19 show that blocking intrinsics prevent inlining from contributing to increased IPC.

Figures 16 and 19 show that loop optimization dramatically increases IPC, when used together with non-blocking intrinsics and inlining. Without loop optimization, non-blocking intrinsics and inlining produce relatively small gains in IPC. With blocking intrinsics, loop optimization only produces small gains in IPC.

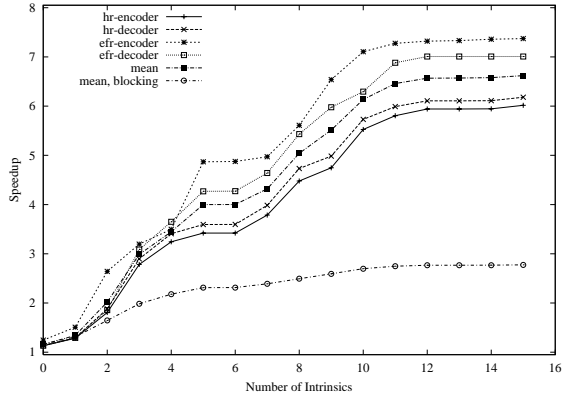


Fig. 8. Speedup versus number of intrinsics for all benchmarks and mean with non-blocking intrinsics, mean only with blocking intrinsics. 1.4 inlining expansion ratio, loop optimization.

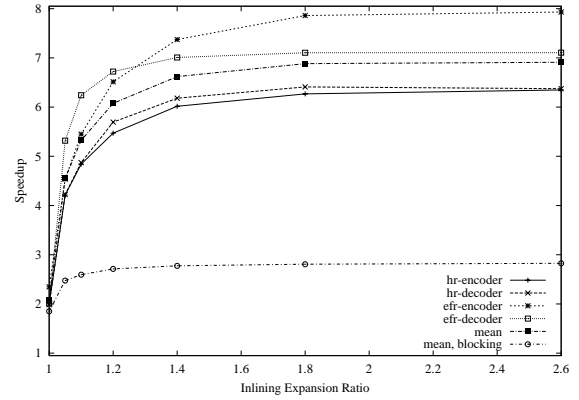


Fig. 11. Speedup versus inlining expansion ratio for all benchmarks and mean with non-blocking intrinsics, mean only with blocking intrinsics. All intrinsics, loop optimization.

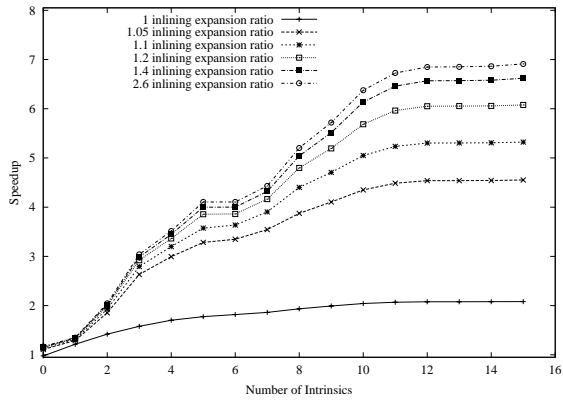


Fig. 9. Speedup versus number of intrinsics for selected degrees of inlining. Non-blocking intrinsics, mean of benchmarks, loop optimization.

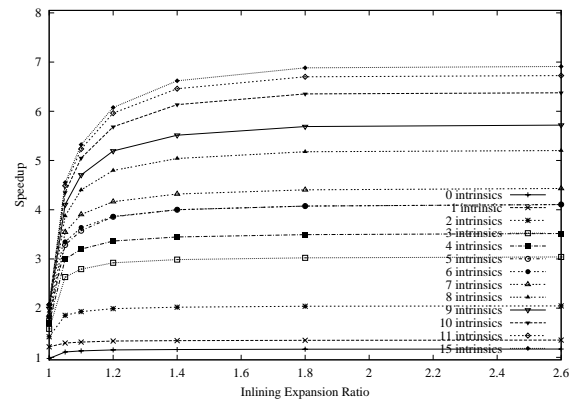


Fig. 12. Speedup versus inlining expansion ratio for selected numbers of intrinsics. Non-blocking intrinsics, mean of benchmarks, loop optimization.

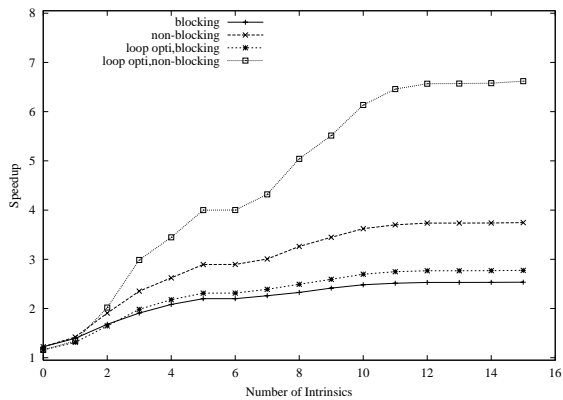


Fig. 10. Speedup versus number of intrinsics with and without loop optimizations with blocking and non-blocking intrinsics. Mean of benchmarks, 1.4 inlining expansion ratio.

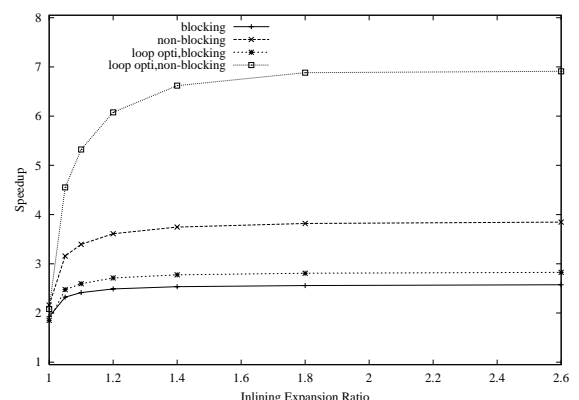


Fig. 13. Speedup versus inlining expansion ratio with and without loop optimizations with blocking and non-blocking intrinsics. Mean of benchmarks, all intrinsics.

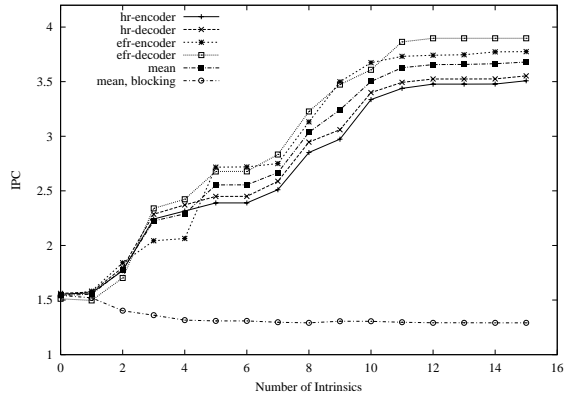


Fig. 14. IPC versus number of intrinsics for all benchmarks and mean with non-blocking intrinsics, mean only with blocking intrinsics. 1.4 inlining expansion ratio, loop optimization.

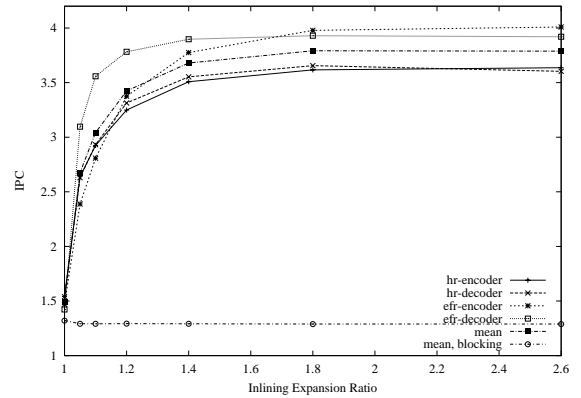


Fig. 17. IPC versus inlining expansion ratio for all benchmarks and mean with non-blocking intrinsics, mean only with blocking intrinsics. All intrinsics, loop optimization.

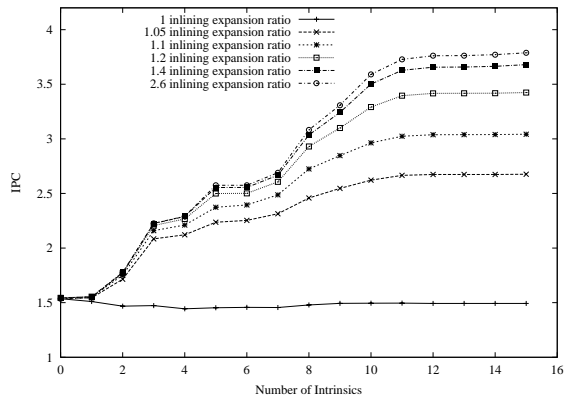


Fig. 15. IPC versus number of intrinsics for selected degrees of inlining. Non-blocking intrinsics, mean of benchmarks, loop optimization.

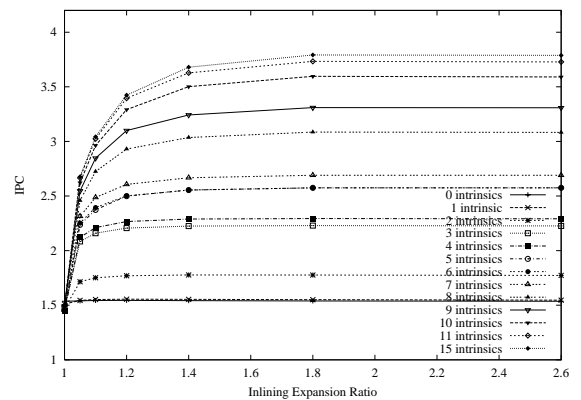


Fig. 18. IPC versus inlining expansion ratio for selected numbers of intrinsics. Non-blocking intrinsics, mean of benchmarks, loop optimization.

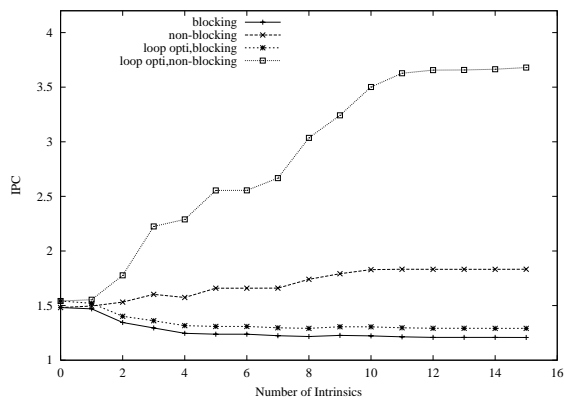


Fig. 16. IPC versus number of intrinsics with and without loop optimizations with blocking and non-blocking intrinsics. Mean of benchmarks, 1.4 inlining expansion ratio.

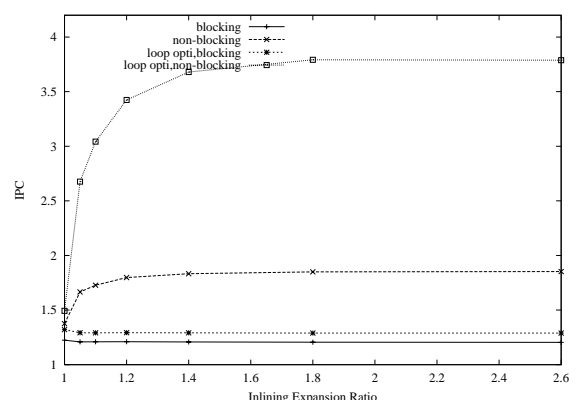


Fig. 19. IPC versus inlining expansion ratio with and without loop optimizations with blocking and non-blocking intrinsics. Mean of benchmarks, all intrinsics.

Intrinsics	Inlining Ratio	Loop Optimization	Speedup	IPC
none	1.0	no	1.00	1.49
none	1.0	yes	0.98	1.54
none	1.4	no	1.22	1.48
blocking	1.0	no	1.93	1.22
non-blocking	1.0	no	2.16	1.38
none	1.4	yes	1.16	1.54
blocking	1.0	yes	1.85	1.32
non-blocking	1.0	yes	2.08	1.49
blocking	1.4	no	2.54	1.21
non-blocking	1.4	no	3.75	1.83
blocking	1.4	yes	2.78	1.29
non-blocking	1.4	yes	6.62	3.68

Fig. 20. Summary of the effect of intrinsics on performance, for both traditional blocking and our non-blocking intrinsics, with profile-directed inlining and loop optimization.

## VI. Future work

We have not measured the performance of our new approach to intrinsics for other DSP applications. However, the effect of applying intrinsics to other applications can be estimated. On the mean, 58% of the baseline execution time in our benchmarks is replaced by our 15 intrinsic substitutions. If this execution time could be reduced to zero, the “ideal” speedup would be 2.38. With non-blocking intrinsics alone (no inlining or loop optimization), our speedup is 2.16, only 9% from the (unachievable) ideal. Therefore, we conclude that the effect of applying intrinsics to an application can be estimated by assuming that the time spent in functions which are replaced by intrinsic instructions is reduced by a factor near one; our data suggest 91%. Furthermore, we anticipate that other applications which benefit substantially from intrinsics alone will also be loop-intensive, and will benefit from the positive interactions among non-blocking intrinsics, profile-directed inlining, and aggressive loop optimizations.

## VII. Conclusions

We have presented a new approach to intrinsic functions which has two substantial advantages over the traditional blocking assembly language method. Our non-blocking method provides both portability across architectures and substantially better performance.

As summarized in Figure 20, for the four GSM speech coder benchmarks, our new approach to intrinsic

functions has better performance than traditional assembly language intrinsics, and interacts substantially better with other optimizations. Without profile-directed inlining or aggressive loop optimization, our approach has 12% better performance (2.16 speedup compared to 1.93 speedup). With profile-directed inlining and aggressive loop optimization, our approach has 138% better performance (6.62 speedup compared to 2.78). Improvement in IPC is similar, 13% without other optimizations (1.38 compared to 1.22) and 185% better with them (3.68 compared to 1.29). We conclude that the substantial improvement of our approach over the traditional method is due primarily to our method’s removal of barriers to optimizations.

## References

- [1] Vojin Zivojnovic, “Compilers for digital signal processors: The hard way from marketing to production tool,” *DSP and Multimedia Technology*, vol. 4, no. 5, pp. 27–45, July 1995.
- [2] Markus Levy, “C compilers for DSP’s flex their muscles,” *Electronic Design News*, pp. 93–107, June 5, 1997.
- [3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986.
- [4] Pierre G. Paulin, Clifford Liem, Trevor C. May, and Shailesh Sutarwala, “DSP design tool requirements for embedded systems: A telecommunications industrial perspective,” *Journal of VLSI Signal Processing*, vol. 9, pp. 23–46, Jan. 1995.
- [5] Stan Yi-Huang Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*, Ph.D. thesis, Massachusetts Institute of Technology, 1996.
- [6] Richard M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, June 1996, For version 2.7.2.1.
- [7] Texas Instruments, *TMS320C6x Optimizing C Compiler*, 1998.
- [8] Richard Scales, “ETSI math operations in C for the ‘C62xx,” Tech. Rep., Texas Instruments.
- [9] European Telecommunications Standards Institute, *Digital cellular telecommunications system; ANSI-C code for the GSM Half Rate (HR) speech codec (GSM 06.06)*, ETS 300 967.
- [10] European Telecommunications Standards Institute, *Digital cellular telecommunications system; ANSI-C code for the GSM Enhanced Full Rate (EFR) speech codec (GSM 06.53)*, Mar. 1997, ETS 300 724.
- [11] Sanjay Jinturkar, Jesse Thilo, John Glossner, Dean Batten, Paul D’Arcy, and Stamatis Vassiliadis, “Profile-directed compilation in DSP applications,” in *Proceedings of the ACM SIGPLAN International Conference on Signal Processing Applications and Technology*, Sept. 1998, pp. 585–589.
- [12] Ben-Chung Cheng, “Pinline: A profile-driven automatic inliner for the IMPACT compiler,” M.S. thesis, University of Illinois at Urbana-Champaign, 1997.
- [13] Daniel Michael Lavery, *Modulo Scheduling for Control-Intensive General-Purpose Programs*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1997.
- [14] Dongni Chen, Wei Zhao, and Huiwen 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*, Sept. 1998, pp. 505–509.