Development of an Efficient DSP Compiler Based on Open64

Subrato K. De, Anshuman Dasgupta, Sundeep Kushwaha, Tony Linthicum, Susan Brownhill, Sergei Larin, Taylor Simpson

Qualcomm Incorporated, San Diego & Austin, USA.

{sde,adasgupt,sundeepk,tlinth,yzhu,slarin,tsimpson}@qualcomm.com

ABSTRACT

In this paper we describe the development of an efficient compiler for digital signal processors (DSP) based on the Open64 compiler infrastructure. Our development has focused on state-of-the-art DSP architectures that allow high degree of instruction level parallelism, support hardware loops, address-generation units, DSP-specific addressing features (e.g., circular and bit-reversed), and many specialized instructions. We discuss the enhancements made to the Open64 compiler infrastructure to exploit the architectural features of contemporary DSPs.

1. Introduction

Open64 is an open source C/C++/Fortran77/90 compiler that is currently used in various industry and academic research projects. It originates from the SGI Pro64(TM) compiler suite that was released under the GNU General Public License. Even though Open64 was not originally intended to be a DSP compiler, we decided to use and enhance it for advanced DSP architectures. We wanted to exploit its powerful set of compiler analyses and support for multiple languages. To place our discussion in context, we will briefly describe the typical features of DSP architectures and different components of the Open64 compiler infrastructure in the next two sections.

1.1 Architectural features of a typical DSP and challenges to compilation

DSPs typically contain heterogeneous register sets, irregular data paths, multiple buses, separate program and data memory, address generation units with specialized addressing modes such as circular and bit-reversed addressing, zero-overhead hardware loops, and instruction-level parallelism. Most DSPs support zero-overhead hardware loops that reduce the control-flow cycles in a loop. Many DSPs have low latency CISC like instructions (e.g., Multiply-Accumulate), and allow fixed point arithmetic, saturation, and rounding. To achieve high performance, application software has to effectively utilize these hardware features.

1.2 Using Open64 to develop an efficient compiler for a DSP

Open64 uses an intermediate representation (IR) called WHIRL that has multiple levels of representation and serves as the common interface for the compiler phases. The important phases of Open64 are described below:

- The very high level optimizer (VHO) lowers aggregates, flattens nested calls, etc.
- The inter-procedural analysis (IPA) first gathers data flow analysis information from each procedure locally. It then generates the call graph, performs inter-procedural analysis and transformations. It performs global variable optimization, dead function elimination, inter-procedural alias analysis, cloning analysis, constant propagation, function inlining, etc.
- The loop nest optimization (LNO) phase calculates dependence graph for array accesses and performs loop transformations, and automatic vectorization.
- The global optimizer (WOPT) computes the control flow graph, the dominator tree, dominator frontier, control dependence set, and then converts the IR to a hashed SSA form. It performs def-use analysis, alias classification, pointer analysis, induction variable recognition/elimination, copy propagation, dead code elimination, partial redundancy elimination, register variable identification, bitwise dead-code elimination.
- The code generator (CG) performs target specific optimizations, instruction selection, scheduling, software pipelining, hyper-block scheduling, register allocation and emits the assembly code.

The details of these phases can be found in [6, 7].

Our goal was to enhance Open64 to better support DSPs. Features of traditional DSPs have been briefly described in section 1.1 Many recent DSPs, however, are load-store VLIW architectures and support some degree of general-purpose computing. Given the new trends in DSP architectures, we believed that Open64 could be modified to be an efficient compiler for these processors.

This paper is organized as follows. Section 2 discusses the C-language extensions we implemented in Open64. Section 3 describes the enhancements we performed on the global optimizer. Section 4 describes the improvements in backend including hyperblock scheduler and register allocator. Finally, we present our conclusions in section 5.

2 C-language extensions for DSP

In recent years, mobile wireless applications and DSP architectures on which they are implemented have continued to increase in complexity. Consequently, compiler support for DSP applications written in a high-

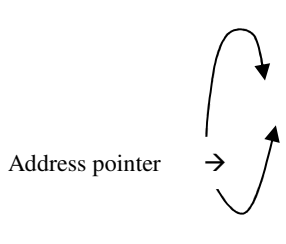
level language (HLL) has become important. While DSPs have traditionally been programmed in assembly, application programmers are transitioning to higher-level languages such as C and C++ for maintainability and in an effort to reduce time-to-market. Therefore, an efficient optimizing DSP compiler becomes very important. However, a HLL such as C lacks support for many of the DSP-specific features such as multiple memory spaces, saturated arithmetic, circular and bit-reverse addressing, and other specialized instructions. So many DSP compilers use C language extensions to address this shortcoming. This section describes the DSP-specific language extensions added to the C/C++ Open64 compiler.

2.1 Intrinsics for DSP-specific addressing modes

DSP's have specialized addressing modes like circular and bit-reverse addressing for efficient implementation of signal and image processing algorithms. These addressing modes are in addition to the standard indirect addressing modes found commonly in different microprocessors. This section describes the programmer level APIs developed to facilitate the use of these two specialized addressing modes (i.e., circular and bit-reversed) and the enhancement needed in Open64 to implement the APIs.

2.1.1 Circular addressing

Circular addressing performs modulo-N wrap around access over a contiguous memory region (called a circular buffer) of size N. In between the two bounds of the buffer, the pointer can be incremented linearly. Figure 1 shows a circular buffer of length 4 words with start address 0x0F04 (i.e., buffer locations are 0x0F04, 0x0F08, 0x0F0C, and 0x0F10).



Hex Address	word data
0x0F00	
0x0F04	
0x0F08	
0x0F0C	
0x0F10	
0x0F14	

Figure 1: A circular buffer of length 4 words and start address as 0x0F04.

Circular addressing is commonly used in digital filters. The C-code fragment below for a FIR-filter shows that the pointer variable (coeffPointer), accessing the array "coeff[]", must be re-initialized to the start location (i.e., &coeff[0]) of the array for every iteration of the outer loop, when using linear pointer increment:

```
int *coeffPointer;
for(i=0; i < noOfInputSamples; ++i){ sum = 0;
   coeffPointer = &coeff[0];
   for(j =0; j < 4; ++j)
      sum += inputSample[i+j] *  *coeffPointer++;
   outputSample[i]  = sum;}
```

However, the re-initialization will be unnecessary if circular addressing is used.

2.1.2 Bit-Reversed Addressing

Bit-reversed addressing is useful for fast-fourier transforms (FFT), viterbi decoding, and any algorithm (e.g., fast DCT) that is based on FFT. Typical FFT algorithms either take an in-order indexed array input and produce a bit-reverse indexed array output or take a bit-reverse indexed array input and produce an in-order indexed array output. Table 1 shows the relationship between the standard index and its bit pattern that is repeatedly incremented by 1, and a bit-reversed pattern and the bit-reversed Index for 3-bit address. For 3-bit addresses the accessible buffer is of length 8 with indices 0 to 7.

Standard Index	Standard Bit Representation	Reverse Bit Representation	Bit-Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 1: relationship between the standard and bit reversed index for a buffer of length 8.

As an example, let's consider Simple radix-2 FFT C code, outlined below. Refer [13] for the complete C-code listing. The function "ReverseBits()" needs to perform explicit bit-reversal if support for bit-reverse addressing is unavailable.

```
unsigned ReverseBits ( unsigned index, unsigned NumBits ){
 unsigned i, rev;
 for ( i=rev=0; i < NumBits; i++ ){
  rev = (rev << 1) | (index & 1);  index >>= 1;
 }
 return rev;
}
void fft( unsigned  NumSamples, int *RealIn, int    *ImagIn,  int
*RealOut,int    *ImagOut ){
   unsigned NumBits;   /*bits needed for indices */
   NumBits = NumberOfBitsNeeded (NumSamples);

  //data copy and bit-reversal ordering into outputs.
  for ( i=0; i < NumSamples; i++ ){
    j = ReverseBits ( i, NumBits );
    RealOut[j] = RealIn[i];
    ImagOut[j] = ImagIn[i];
  }

 //Inner kernel operations of FFT which reads RealOut[] and
 //ImagOut[] and then writes the final output in them. The code
 //below is actually within nested loops.
  . . . . . .
  tr = ar[0]*RealOut[k] - ai[0]*ImagOut[k];
```

```
    ti = ar[0]*ImagOut[k] + ai[0]*RealOut[k];
    RealOut[k] = RealOut[j] - tr; ImagOut[k] = ImagOut[j] - ti;
    RealOut[j] += tr;  ImagOut[j] += ti;
    . . . . . .
}
```

2.1.3 Design of intrinsics for circular and bit-reverse addressing

For circular and bit-reversed addressing, a user level intrinsic can specify the buffer (defined by the start location and the length), the memory location to access the data, and the increment or decrement value used to compute the next location in the buffer. A pointer variable in a high level language can be used to access data from the buffer. We provided four user level intrinsics for circular and bit-reverse load and store:

- LOAD_CIRC_DTYPE(v, p, s, l, a);
- STORE_CIRC_DTYPE(v, p, s, l, a);
- LOAD_BREV_DTYPE(v, p, s, l, a); and,
- STORE_BREV_ DTYPE(v, p, s, l, a).

where "v" is the variable loaded or stored; "p" is the pointer variable used to access the buffer and is an l-value; "s" is a signed value for linear increment/decrement and is a constant; "l" is the buffer length in # of data elements in buffer; "a" is the Buffer start address and is an l-value; "DTYPE" denotes the data type in the buffer could be int, short, char, double, etc depending on the data types supported in the DSP.

As an example, when the API's for circular load is used, the FIR filter code shown in section 2.1.1 becomes:

```
int *coeffPointer; int value;
for(i=0; i < noOfInputSamples; ++i){ sum = 0;
  for(j =0; j < 4; ++j){
    LOAD_CIRC_INT(value, coeffPointer, 1, 4,  &coeff[0]);
     sum += inputSample[i+j] * value;
  } outputSample[i]  = sum;
}
```

The API "LOAD_CIRC_INT" loads the elements of the array "coeff[]", in the integer variable "value", in a modulo wrap-around fashion, creating a circular buffer of length 4. There is no need to re-initialize the "coeffPointer" in the outer loop body, unlike in section 2.1.1.

Similarly, when the intrinsics for bit-reverse store is used the function "ReverseBits()" is no longer needed in the Simple Radix-2 FFT implementation shown in section 2.1.2. The code below the comment statement "data copy and bit-reversal ordering into outputs" in function "fft()" can be implemented using bit-reverse store intrinsics as:

```
int *realOutPointer; int *imagOutPointer;
for ( i=0; i < NumSamples; i++ ){
  STORE_BREV_INT(RealIn[i], realOutPointer, 1,
                        NumSamples, &RealOut[0] ).
  STORE_BREV_INT(ImagIn[i], imagOutPointer, 1,
                        NumSamples, &ImagOut[0] ).}
```

2.1.4 Implementation issues in Open64

WHIRL is a strict tree form with each node representing an operator that takes zero or more operands and produces a single output.. Therefore, WHIRL cannot represent the semantics of an instruction such as "result = *(ptr++incr)" as a single node. Consequently, a single Open64 "__builtin" cannot be used to implement the user-level intrinsics/APIs described in section 2.1.3. Also the characteristics of the load and store operations need to be maintained. Hence, these user-level intrinsics are implemented as macros that expand to two internal operations: (i) indirect load/store with the circ/bit-reverse pointer, and (ii) an internal intrinsic "circular/bit-reverse update". For example, LOAD_CIRC_INT() would expand to an indirect load and a internal circular update intrinsic that was transparent to the user:

```
#define LOAD_CIRC_INT(v, p, s, l, a) \
( (v) =  *( p);   (p) = (int *) circ_update((void *) (p), (s), (l),
(void *) (a)) )
```

Similarly, STORE_CIRC_INT expands to:

```
#define STORE_CIRC_INT(v, p, s, l, a) \
( *( p) = (v);   (p) = (int *) circ_update((void *) (p), (s), (l),
(void *) (a)) )
```

These WHIRL nodes (i.e., the load/store and the circular/bit-reverse update) are combined into a single multi-output operation in the CG-phase. The IR looks like (we are using the circular load/store to illustrate):

```
<result> = load_indirect(pointer);  or,
          store_indirect(pointer) = <source>;
and,
<pointer> = circ_update (pointer, step, CR)
```

CR is the configuration register that defines the associated circular/bit-reverse buffer based on buffer size and the start address. DSP architectures can have different hardware implementations of the actual circular/bit-reverse instruction; so we assume a generalized form of the instruction where a configuration register is used to keep track of the buffer associated with a particular circular/bit-reverse load/store. The operation showing the initialization of CR is not shown. The load/store and the corresponding circular/bit-reverse update operations are then combined in the CG phase into a single 2-output CGIR operation (2 outputs for the circular/bit-reverse load: one for pointer update and the other for the loaded value):

```
<result, pointer> =load_circular_update(pointer, step, CR),
or
<pointer> =store_circular_update(source, pointer, step, CR)
```

The alias analysis for the circular/bit-reverse load/store operations are made very conservative, since the pointer updates are no longer linear. Some enhancement is needed in Open64 to perform efficient alias analysis for circular

updates, where the pointer wraps-around the buffer and accesses the same memory locations periodically.

2.1.5 Hoisting and allocation of configuration registers

A DSP can have multiple configuration registers (CRs) to support multiple circular/bit-reverse buffers. The CRs need to be assigned and allocated effectively. In our implementation within Open64, CRs are considered dedicated temporary names (TNs) and require special handling. A simple algorithm for the hoisting of the loop-invariant CR assignment statements and a balanced allocation of the CR TNs, for two CR registers (i.e., CR0 and CR1) is described below:

1. The "def" of a CR is inserted just before its "use" in the circular/bit-reverse load/store.

2. A special pass hoists the "def" of CRs from the loop body, based on the following:

- A "def" of a CR can be hoisted only if a loop-invariant TN is assigned to it.
- Multiple "def"s of the same CR from the same loop-invariant TN are redundant (to a single assignment) and can be hoisted above the loop.
- If different TNs (even if loop-invariant) are assigned to the same CR at different points within the loop body, none of the assignments can be hoisted above the loop.

3. The CR allocation mechanism follows use-def chains, since it's the use of the TN in the circular/bit-reverse load/store instruction that determines if a CR needs to be allocated. At the "def" of the TN a copy of the TN to the CR is inserted. Thus, if the same TN has any use other than in circular/bit-reverse load/store, it gets a GPR through actual register allocation.

- A vector data structure is maintained for the CR allocations to TNs in each PU.
- If there is a pre-existing CR allocated to the same TN, it is reused.
- When a CR allocation is requested for a TN, the least allocated CR (CR0 or CR1) within the BB is selected to keep the CR usage balanced within the BB.

2.2 Pragmas for loop optimization

Many DSP architectures have hardware loops. Efficient usage of the hardware loop support can be facilitated if certain loop properties (i.e., minimum / maximum number of loop iterations, loop step values) are known by the compiler.

2.2.1 Loop trip count pragmas

We added three pragmas – LOOP_TRIP_COUNT_MIN, LOOP_TRIP_COUNT_MAX, and LOOP_TRIP_COUNT-_MODULO -- that the programmer can use to specify the lower bound, the upper bound, and a divisor of the loop trip count respectively. The compiler can use these information to omit loop guards, omit cleanup or alternate loops during unrolling and software pipelining. The compiler can decide whether it is profitable to unroll or software pipeline the loop. If the "min" is specified, the guard condition for a loop can be eliminated. If the "modulo" is a multiple of the unroll factor, the remainder loop can be eliminated while unrolling. If both the "min" and the "max" values are same, the loop is guaranteed to iterate min=max times. It is useful in cases when ((loop end – loop start)/loop step) is a constant quantity, but both "loop end" and "loop start" are variables. In this situation the trip-count computations can be made redundant.

2.2.2 Pragma for loop unrolling

The pragma LOOP_UNROLL(N), where "N" is a positive constant, allows programmers to notify the compiler that a loop should be unrolled "N" times. Scheduling opportunity and parallelism can be increased by enlarging the loop body. The jump overhead can be reduced for processors that implement software loops

2.2.3 Implementation of the loop pragmas in Open64

Even though many industrial DSP compilers [1,2,3] provide support for similar pragmas, the focus is on supporting these pragmas in Open64. We present an overview of the approach:

1. Pragmas are recognized in the C/C++ front-end and added as whirl "PRAGMA" nodes in the IR just before the whirl node for the loop.

2. The function "CODEREP *IVR:: Compute_trip_-count()" is modified as:

- Bypass loop guard generation depending on the LOOP_TRIP_COUNT_MIN
- Loop trip count computation is replaced with a constant trip count, if LOOP_TRIP_COUNT_MIN == LOOP_TRIP_COUNT_MAX.

3. BB_Add_Annotation() is used to add the pragmas from Whirl to CG IR.

4. The following modifications are done during CG loop optimizations.

- If the loop min pragma >= SWP stage count, bypass generation of alternate low-trip count loop during software pipelining.
- Modify functions "Unroll_Do_Loop" and "Unroll_Dowhile_Loop" to bypass "remainder loop" and "unrolled loop guard" based on the loop count MODULO or loop count MIN == loop count MAX.

5. The loop count min/max/modulo pragma are adjusted if loop unrolling occurs in CG.

The pragma based loop unrolling is implemented in the LNO phase as a generalized unroller using existing APIs.

Even though Open64 performs limited loop unrolling in the CG phase, the benefit is in implementing at LNO because of the machine independent optimizations (WOPT) performed after LNO. The implementation of the generalized pragma based loop unroller in LNO is outlined below:

1. Detect a whirl "DO_LOOP" IR and any loop pragmas associated with it.

2. Determine the loop start, end, step, and trip count from the "DO_LOOP" IR.

3. For constant trip count loop, determine if the unroll factor perfectly divides the trip count (or the loop count modulo pragma value, if specified), giving remainder =0.

4. If remainder=0 in step 3, invoke "Unroll_Loop_By_Trip_Count()", unroll by the specified unroll factor, go to step 7.

5. If remainder! = 0 in step 3, split the loop into two loops: the main loop having trip count that is completely divisible by the unroll factor, and the remainder loop, whose trip count = original trip count % unroll factor. Figure 2 illustrates loop splitting with an example.

6. Call "Unroll_Loop_By_Trip_Count()" for the main loop, unroll by the specified unroll factor, (remainder loop is left untouched). Go to step 7.

7. If unrolling occurs, adjust loop count min/max/modulo pragmas. Set unroll pragma to 1.

8. END

Original loop:

```
#pragma LOOP_UNROLL(6)
 for (j = 0; j <77; j+=5)
```

IR of the DO_LOOP after PreOpt is equivalent to:

```
 for (j = 0; j*5 <= 76; j++)
```

The loop after splitting becomes:

```
Main loop (first 12 iterations, perfectly divisible by 6):
 for (j = 0; j*5 <= 59; j++)
Remainder loop (Last 4 iterations):
  for (j = 12; j*5 <= 76; j++)
```

Figure 2: Illustration of loop splitting followed by unrolling of the main loop.

We implemented a parser to extract the loop start, end, and step information (both for constant and variable trip count loops) from the WN_end() of the DO_LOOP abstract syntax tree (AST). Figure 3 shows the generalized structure of the AST obtained from WN_end() of the DO_LOOP structure. For some variable trip count loops, the AST structure has different canonical forms for the same functional loop end condition.

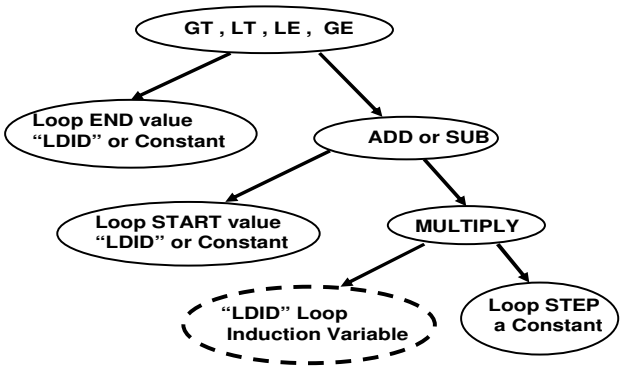


Figure 3: Generalized form of the WN_end() AST for a DO_LOOP structure

3. Enhancement of the Global Optimizer

3.1 Register promotion of small structures

Many advanced DSPs support bit-level operations that can insert/extract a number of consecutive bits at a given bit-offset to/from a register. Applications like network protocols, cryptography have many bit level operations involving structures and unions (with a mix of bit-fields and standard C-data type) of size less than or equal to 8 bytes (called "small structures" in this paper). A considerable reduction in stack size and cycle performance is possible if the compiler is able to allocate the small structures completely in registers. The algorithm uses the whirl level "EXTRACT_BITS" and "COMPOSE_BITS" operations to replace individual structure-member auxiliary symbols with the full structure-sized auxiliary symbol in the WOPT phase, as shown in figure 4.

- The auxiliary symbol table for a structure/union variable contains entries for each member of the structure or union that are actually "used" or "defined" in the program. As in figure 4, auxiliary symbols "st 8", "st 9", "st 10", and "st 4" are auxiliary symbols for the type "PAIR". Their mappings: "st 8" is for "w[0]";"st 9" is for "h[1]"; "st 10" is for "h[3]", and "st 4" is for "d".
- The register variable identification (RVI) cannot promote structure/unions to registers because the overlapping data layout introduces "may-use" and "may-def" nodes, resulting in allocating the small structure always in stack.

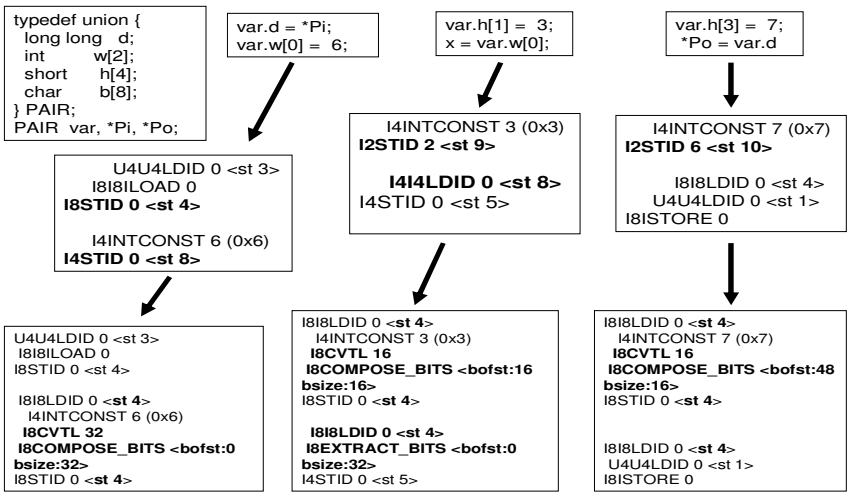


Figure 4: Use of EXTRACT_BITS and COMPOSE_BITS for register promotion of small structures.

- However, when all the structure-member auxiliary symbol accesses are replaced with the unique full-sized auxiliary symbol access, using "EXTRACT_BITS" at the "use" locations, and using "COMPOSE_BITS" at the "def" locations, the "may-use" and "may-def" nodes are no longer present. This facilitates register variable identification (RVI).

This optimization is always beneficial for reducing stack size. Moreover, there can be a significant improvement in cycle performance if the insert and extract operations can be further removed. As an example, if the access involves 32-bit integer members for a 64-bit structure, the insert and extract operations can be completely removed.

3.2 Removal of redundant alias stores encountered during C++ inlining

We noticed many instances of redundant stack usage and dead stores to stack locations when compiling C++ code having variables of type small structures and classes with bit-fields. The problem is especially acute when C++ overloaded operator functions and small class member functions are inlined. For a DSP embedded on a mobile device, memory (program, data, or runtime memory) is a critical resource. Hence, even though the discussion in this section in not specific to a DSP, the impact of the modification is of high importance for DSP's embedded in mobile devices.

The Whirl IR has address of a stack variable saved using the operation "LDA". The first WOPT pass removes the "LDA" operation, since the stack variable can be directly accessible after inlining the C++ overloaded operator functions and class members. But the second WOPT pass still sets the POINTS_TO ALIAS flag, which results into an alias store that is actually a dead store. Figure 5 illustrates the problem: the "may-def" node is still associated to the store of stack variable "anon1" in the second WOPT pass even after the WHIRL operation "LDA anon1" is deleted in the PreOpt pass, leading to dead store of "anon1". The problem is solved as follows:

1. VHO lowering doesn't lower aggregate types when bit-fields are present. It lowers by converting a single MTYPE_M structure copy to multiple MTYPE_Ix/Ux depending on the number & the type of the structure members (it flattens nested structures). This was upgraded to make a single MTYPE_M structure copy lower to an equivalent single MTYPE_I/U structure copy for copies <= 8 bytes, irrespective of whether bit-fields are present.

2. The POINTS_TO is computed from the attributes of "ST" set by the front-end, and not from attributes recomputed in WOPT and set in class "BE_ST". This gives rise to stale alias information in the second pass of WOPT. The attributes of BE_ST are used instead:

- Class "BE_ST" has fields like "address_used_locally", "address_passed", etc, as in "ST".
- Instances of "BE_ST" are created when WOPT creates its own symbol table (auxiliary symbols). "address_used_locally", "address_passed" in BE_ST are recomputed:
- An address saved operation (e.g., "LDA") done on a symbol sets "address_used_locally" in an instance of the class BE_ST for the symbol.
- If the symbol's address is ever passed to a function, it set "address_passed".
- The attribute fields of ST are never updated. Hence "POINTS_TO" of a symbol (created using information from both ST and BE_ST), can have outdated information for a two pass WOPT (e.g., when a previous WOPT pass deletes a redundant LDA).

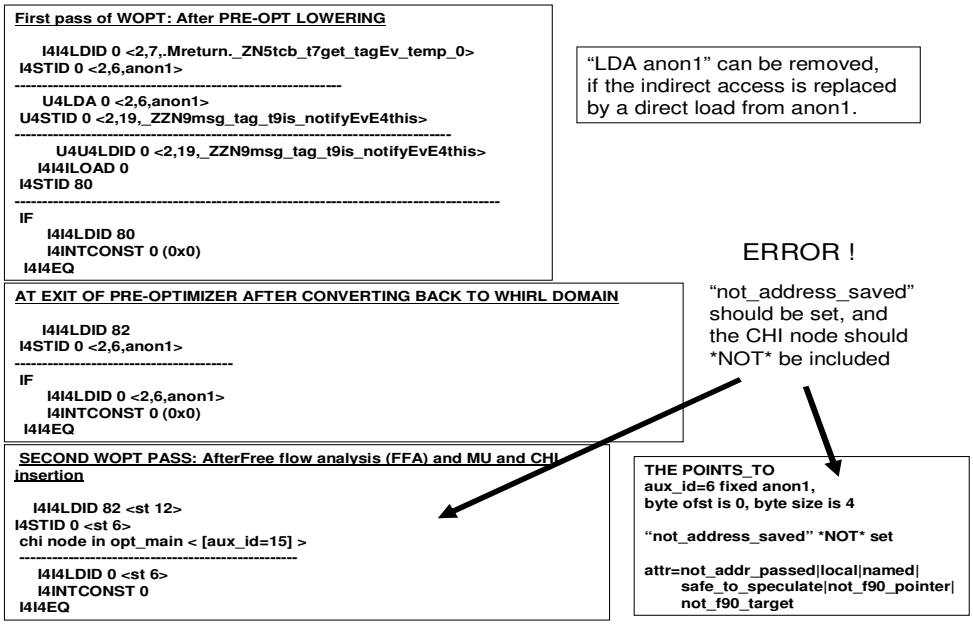


Figure 5: Example of redundant alias information

- If the first WOPT pass deletes all address saved operations (LDA's) for a local stack symbol, the "address_used_locally" field of BE_ST is FALSE in the second pass, even though "address saved" in "class ST" is TRUE (i.e., ST is never updated). The solution is to use the "address_used_locally" field from BE_ST to compute POINTS_TO (instead of the "address_saved" field of "class ST").

We noticed cycle performance improvements ranging from 3% to 40%, stack size reductions of as much as 50% and code size reductions by 1% to 2% for some modem applications when register promotion of small structures are enabled as described in section 3.1. These applications have extensive use of small structures and unions and are a mix of unstructured control flow and some loops. In a C++ kernel code having extensive use of small structures and classes, we noticed stack size reductions of up to 80% when both the modifications described in sections 3.1 and 3.2 are applied.

4. Enhancement of the code generator: hyperblock scheduling and register allocation

In Open64, the main phases of the code generator are: instruction selection, register allocation, and scheduling. The compiler should strive to exploit the instruction-level parallelism provided by the architecture and also efficiently allocate registers minimizing spills or copies. The register allocation process can be complicated by the presence of register pairs. In this section, we will discuss our efforts to modify the code generator of Open64 to exploit and accommodate architectural features typically present on a DSP.

4.1 Aggressive hyperblock scheduling

Optimal instruction scheduling has been shown to be a NP-complete problem [9][1]. As a result, compilers typically use heuristic-driven scheduling algorithms. Most conventional compilers use a variant of list-scheduling. List scheduling, described in [11], is an efficient algorithm that encodes scheduling constraints in a directed-acyclic graph (DAG). The scheduler traverses the DAG and consults a set of heuristics to schedule instructions. This approach works well in a single basic-block i.e., in the absence of control-flow. Several algorithms such as extended-basic block scheduling, trace scheduling, superblock scheduling, and hyperblock scheduling have since tried to accommodate control-flow in an instruction scheduler. In the next few sections, we will focus on one such technique: hyperblock scheduling. The use of predicated instructions if available in a DSP can be extremely beneficial in reducing the number of branches in the compiled code and enabling aggressive scheduling across basic-blocks. Therefore, we wanted to tune Open64 compiler to fully exploit this feature.

[1] Interestingly, eliminating control-flow does not improve the complexity of the problem from a compiler-engineering perspective. Single basic-block scheduling for realistic architectural models has been shown to be NP-complete. [10]

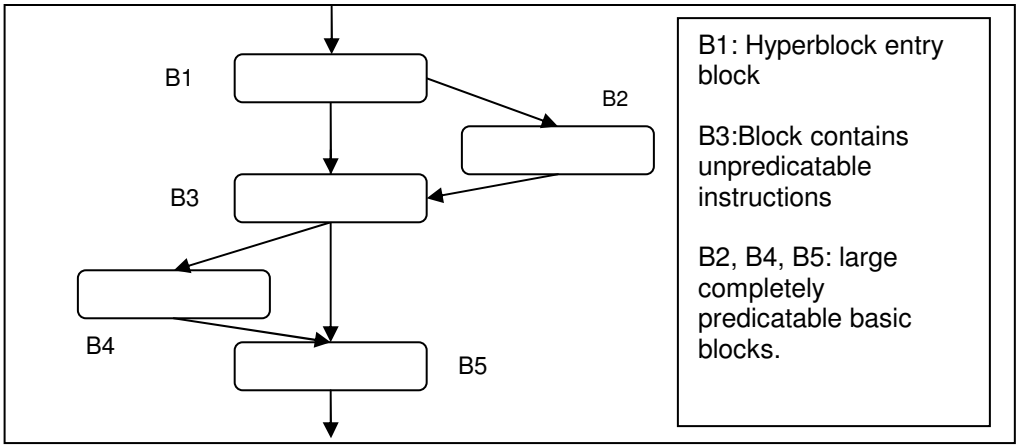


Figure 6: Potentially profitable hyperblock rejected by default formation algorithm

4.1.1 Hyperblock scheduling in Open64

The Open64 compiler contains a hyperblock scheduler that is largely derived from the algorithm described in [12]. To obtain efficiently scheduled code, it was important that the hyperblock scheduler was tuned for our processor. In this paper, we describe the modifications we made to the default Open64 hyperblock algorithm to accommodate constraints in the target instruction set.

4.1.2 Predicated execution

The VLIW target contained support for predicated execution. However, in comparison to processors such as the Itanium, the predicated execution model on our target processor was more restricted. In particular, not every operation in the instruction set could be predicated. Due to this constraint, we modified the hyperblock-selection mechanism in Open64. This restricted form of predication stems from a focus on power and resource savings and is common on DSP architectures. Thus, changing Open64 to handle these constraints makes Open64 more attractive for embedded targets.

4.1.3 Hyperblock formation

A hyperblock, as described in [12], is a set of basic blocks that contains a single entry block that dominates all other blocks in the set. Note that the set can contain multiple blocks where control-flow exits out of the hyperblock. The default hyperblock-formation mechanism in Open64 rejected any basic blocks that contained instructions that could not be predicated. This requirement proved to be too restrictive for our architecture. Rejecting every set of basic blocks that contained an *unpredicatable*[2] instruction resulted in too few hyperblocks being formed. We wanted to aggressively construct and schedule hyperblocks. Therefore, we relaxed the restrictions in the hyperblock-selection algorithm to accept a larger set of hyperblock candidates.

4.1.4 Modifications to the hyperblock-formation algorithm

In addition to the architectural constraint described in section 4.1.2, the changes in the formation algorithm was motivated by code that was generated by Open64. Figure 6 shows an example of compiled code we observed on several benchmarks. The basic-block labeled B3 in the figure contains an unpredictable instruction. Thus, this set of blocks would be rejected by the default hyperblock formation algorithm. However, note that blocks B4 and B5 contain a large number of instructions and, therefore, it can be profitable to consider the blocks shown in the figure as candidates for hyperblock scheduling. To that end, we relaxed the selection criteria in the Open64 formation algorithm. After our modifications, a candidate block can contain unpredicatable instructions if it post-dominates the entry block of the hyperblock. Consider block B3 in Figure 6 that contains an unpredicatable instruction. Since it post-dominates the entry block – B1 – it will be considered for inclusion in the hyperblock.

4.1.5 Safety

As we shall discuss in the next section, this modification allows the hyperblock scheduling to be more aggressively applied in our benchmark suite. More importantly, however, this modification preserves the semantics of the program i.e., this transformation is safe. We outline two key attributes of the modified algorithm that ensure safety:

• First, the entry block of the hyperblock dominates all other blocks in the hyperblock. Consider a basic block in

[2] A note on terminology: in this paper, we have used the terms *unpredicatable* to refer to an instruction that cannot be predicated on our architecture and *predicatable* for instructions that can be predicated.

Benchmark	Hyperblocks formed by selection algorithm *M/D*	Basic blocks considered by selection algorithm *M/D*
networking applications	1.13	1.72
telecommunication applications	1.00	1.49

Table 2: Hyperblock formation by the default (D) and modified (M) algorithms

the hyperblock that post-dominates the entry block. The instructions in such a basic block will always execute if control flows along the hyperblock.

• Second, the Open64 hyperblock-aware instruction scheduler constructs a data-precedence graph for the entire hyperblock and does not move instructions across basic blocks if the results of the instruction clobber a value in the live-out set of the destination basic block.

These two properties of the algorithm guarantee that the modification does not compromise program safety during instruction scheduling.

4.1.6 Profitability

The modification described in section 4.2.4, the Open64 compiler increased the aggressiveness of the scheduler – in particular, the compiler was able to consider significantly more basic blocks as candidates for hyperblock scheduling. Table 2 lists the additional number of basic blocks that were considered and hyperblocks that were formed because of the modification for two classes of applications. Both application classes contained a set of programs commonly used in DSPs for wireless communications. In the table, D refers to the default Open64 algorithm and M refers to the modified algorithm. The numbers have been normalized to the default hyperblock algorithm. As can be seen in the table, the modified algorithm allowed the examination of basic blocks that would have been rejected by the default block-selection algorithm. In the next section, we will discuss a change we made in the code-generator to effectively handle register-pairs.

4.2 Efficient register pair allocation

Many DSP applications that operate on 64-bit quantity also need access to the upper half (bits 32-63) or the lower half (bits 0-31) of the data. This can be problematic on DSPs that support 64-bit registers by grouping two adjacent 32-bit registers as register pair. For example, adjacent registers r0 and r1 can be grouped together as r1:0 to hold a 64-bit value.

On these architectures, a simple solution to the problem is to introduce a copy of the upper or lower register to another register. Introducing a copy can be expensive, especially when it appears in a tight loop. We want to eliminate this copy. The allocation of register pairs has been the focus of prior compiler research [14,15]. However, we wanted to improve register-pair allocation within the framework of the Open64 compiler. We were particularly interested in avoiding drastic changes to the default Open64 allocation algorithm.

4.2.1 The solution strategy

We decided to handle this problem in the register allocator since the allocator already performs preference copying. The allocator recognizes these operations as special pseudo copy instructions. We introduced 2 pseudo instructions as follows:

• pseudo_pair_low: source operand is 8-byte TN, result operand is lower 4-byte of source

• pseudo_pair_high: source operand is 8-byte TN, result operand is upper 4-byte of source.

Figure 7 shows original program and IR dump after CG Expand and EBO. EBO recognizes that 8-byte TN242 is right shifted by 0x20 and replaces it with pseudo_pair_high instruction. Next, the register allocator needs to recognize pseudo_pair_high as a special copy and assign TN252 the same color as the upper register of GTN242. In the next few sections, we discuss our implementation of this optimization.

4.2.2 Challenges

Implementation of pseudo register pair optimization proved to be more difficult than originally anticipated. The Open64 register allocator works in 2 independent phases. First GRA allocates live ranges which are live across multiple basic blocks, also known as global live ranges. Next, LRA allocates live ranges local to every basic block.

The source and result operands of pseudo pair instructions can be either global or local. Thus, we need to handle the following 4 cases:

Source	**Destination**	**Comments**
Global	Global	Both handled in GRA
Global	Local	Need interaction between GRA and LRA
Local	Global	Need interaction between GRA and LRA
Local	Local	Both handled in LRA

Original C code	After CG Expansion

extern long long bar(int a, long long b);
extern int baz(long long);
int foo(long long a, int c) {
long long tmp = c + bar(1, a);
int retval = (tmp>>32);
if (c > 0) tmp++;
return retval + baz(tmp); }
	[11] TN249:8 :- asr_i_p TN242:8 (0x20) ;
	[11] TN250:4 :- tfr TN249:8 ;
	[11] TN1(r0):4 :- add TN248:4 TN250:4 ;
	After EBO
	[11] TN252:4 :- pseudo_pair_high GTN242:8 ;
	[11] GTN1(r0):4 :- add GTN1(r0):4<defopnd> TN252:4

Figure 7: IR before and after CG expand and EBO

This problem is aggravated by the fact that LRA does not build an interference graph and hence can not reason about preference copies. To reduce the problem space, we globalize any local live range which appears in pseudo pair instruction, provided the other operand is global.

For example, in Figure 8, TN252 will be globalized since it is the result operand of pseudo_pair_high and the source operand TN242 is global. After promoting local live ranges to global, we reduce the original problem to the following 2 cases:

Source	Destination	Comments
Global	Global	Both handled in GRA
Local	Local	Both handled in LRA

4.2.3 Implementation details

We can group most of the changes in the following categories: Globalize local live ranges, GRA Changes, and LRA Changes

4.2.3.1 Globalize local live ranges

First, we identify local live ranges that should be promoted to global by inserting a call to a new function Identify_Pseudo_Globls in Create_GRA_BBs_And_Regions for each basic block. In Identify_Pseudo_Globls, we iterate over each instruction in the basic block and mark a TN as pseudo global if it is the source or the destination of a pseudo pair instruction and the other TN is global. We add a new field pseudo_globls in bbregs structure to hold promoted pseudo global TNs in each basic block. Next, pseudo global TNs are added to GTN_UNIVERSE and the needs_a_register set of GRA_BB.

After Create_GRA_BBs_And_Regions, all pseudo global TNs are recognized as globals since they are present in GTN_UNIVERSE. Create_Live_BB_Sets is called to populate live basic block sets of each global TN. Since pseudo globals are not truly global (live only in 1 basic block), Create_Live_BB_Sets does not account for them. We populate a pseudo global TN's live set by iterating over every basic block.

4.2.3.2 GRA changes

We were able to use the existing preference copying mechanism to handle pseudo pair instructions. The only modification needed was in the CGTARG_Is_Preference_Copy function to return TRUE for pseudo pair instructions. A pseudo pair instruction was treated like a normal copy instruction that GRA attempts to remove by preferencing.

4.2.3.3 LRA changes

LRA is expected to be a quick single pass allocation to color all local live ranges. LRA does not create interference graph and performs limited preference coloring. To support pseudo pair instructions, we made the following changes in LRA:

1. Added a new field preftn_list to live_range structure to keep track of preferencing TNs.

2. For each basic block, we added a local pass to ensure that source and result operands of pseudo pair instructions are defined only once. This is a relatively inexpensive way of ensuring that source and result operands can preference each other without building an interference graph.

3. If pseudo pair instructions of the basic block do not interfere, populate the preftn_list list of each local live range.

4. Add code to the Open64 function Allocate_Register that checks if a TN can be preferenced. If a member of a TN's preftn_list has been allocated a register, we assign the same register to the TN.

4.2.4 Performance analysis of efficient register-pair allocation

We evaluated the impact of the register-pair optimization on two sets of benchmarks: telecommunication benchmarks and kernel codes commonly used in DSP applications. On telecommunication benchmarks, the optimization improved performance by 3.91% on average. On kernel codes, enabling the register-pair optimization resulted in an improvement of 1.77% on average.

5. Summary and conclusion

In this paper we described improvements to Open64 for state of the art advanced DSP targets. We added support for DSP-specific C-language extensions and loop optimizations. We enhanced the register promotion of small structures and the removal of dead stores in the global optimizer. We also changed the hyperblock scheduling algorithm to support DSP architectures that placed significant constraints on the default block-selection algorithm. Finally, we described the enhancement done to the register allocator to efficiently allocate register pairs so that the register-copies are minimized. Even though the changes described in this paper are not a complete set of changes needed to enhance Open64 for DSPs, they proved to be effective: the modifications improved performance for embedded programs and also allowed compiler users to more effectively author DSP applications. The work to enhance Open64 for DSPs is ongoing and further enhancements are being looked into. We compared the enhanced Open64 compiler with a GNU 3.4.6. C/C++ compiler retargeted for the same DSP. On average, the enhanced Open64 compiler with interprocedural analysis and optimizations (i.e., IPA) performed 5% to 40% better (cycle comparison) than GCC 3.4.6. In a few benchmarks, GCC 3.4.6 performed slightly better than Open64 and we are investigating the causes.

6. Acknowledgements

Special thanks to Don Padgett and John McEnerney for their initial effort on retargeting Open64 for the target DSP. We also thank Sun Chan for some of the valuable discussion we had with him with different issues in Open64 compiler.

REFERENCES

[1] TMS320C6000 Optimizing Compiler User's Guide, spru187l, May 2004.

[2] TMS320C55x Optimizing C/C++ Compiler User's Guide, spru281e, March 2003.

[3] C/C++ Compiler and Library Manual, for TigerSHARC® Processors. Revision 2.0, January 2005, Part Number 82-000336-03.

[4] Programming DSPs using C: efficiency and portability trade-offs, Embedded Systems, May 2000.

[5] Extensions for the programming language C to support embedded processors, ISO/IEC JTC1 SC22 WG14 N1021, Date: 2003-09-24, Reference number of document: ISO/IEC DTR 18037.

[6] Open64, http://open64.sourceforge.net/

[7] WHIRL Intermediate Language Specification, whirl.pdf. http://open64.sourceforge.net

[8] WHIRL Symbol Table Specification, symtab_Pro64_SGI.pdf. http://open64.sourceforge.net

[9] Bernstein, D., Rodeh, M., and Gertner, I. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. IEEE Trans. Comput. 38, 9 (Sep. 1989), 1308-1313.

[10] Hennessy, J. L. and Gross, T. 1983. Postpass Code Optimization of Pipeline Constraints. ACM Trans. Program. Lang. Syst. 5, 3 (Jul. 1983), 422-448.

[11] Gibbons, P. B. and Muchnick, S. S. 1986. Efficient instruction scheduling for a pipelined architecture. In Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction (Palo Alto, California, United States, June 25 - 27, 1986).

[12] Mahlke, S. A. 1997 Exploiting Instruction Level Parallelism in the Presence of Conditional Branches. Doctoral Thesis. UMI Order Number: UMI Order No. GAX97-17305., University of Illinois at Urbana-Champaign.

[13] Simple Radix-2 FFT code, http://www.yov408.com/html/codespot.php?gg=36

[14] Daveau, J., Thery, T., Lepley, T., and Santana, M. 2004. A retargetable register allocation framework for embedded processors. SIGPLAN Not. 39, 7 (Jul. 2004), 202-210.

[15] Briggs, P., Cooper, K. D., and Torczon, L. 1992. Coloring register pairs. ACM Lett. Program. Lang. Syst. 1, 1 (Mar. 1992), 3-13