The Scc Compiler: SWARing at MMX and 3DNow!

Randall J. Fisher and Henry G. Dietz

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285
`rfisher@ecn.purdue.edu` and `hankd@ecn.purdue.edu`

Abstract

Last year, we discussed the issues surrounding the development of languages and compilers for a general, portable, high-level SIMD Within A Register (SWAR) execution model. In a first effort to provide such a language and a framework for further research on this form of parallel processing, we proposed the vector-based language SWARC, and an experimental module compiler for this language called Scc which targeted IA32+MMX-based architectures.

Since that time, we have worked to expand the types of targets that Scc supports and to include optimizations based on both vector processing and enhanced hardware support for SWAR. This paper provides a more formal description of the SWARC language, describes the organization of the current version of the Scc compiler, and discusses the implementation of optimizations within this framework.

1. Introduction

In the SWAR processing model, a wide data path within a processor is treated as multiple, thinner, SIMD-parallel data paths with each register word effectively partitioned into *fields* that can be operated on in parallel. In this paper, we refer to one word of fields as a *fragment*, and a multi-fragment object as a *vector*.

Current compiler support for the SWAR model falls into four major categories: compiler "intrinsics", classes or types representing a fragment, semi-automatic vectorization of loops, and languages which provide first-class vector objects. The first two categories are typically limited to giving the programmer low-level access to the SWAR instructions of the target. The third provides an abstract model which hides the use of these instructions by employing well-known techniques to parallelize some loops in existing C code. The fourth category provides an abstract model in which the semantics of the language indicate which operations can be automatically parallelized. We will briefly discuss these four methods of support as they apply to the MMX and 3DNow! families of SWAR extensions.

Compiler intrinsics are preprocessor macros which provide a function-call-like interface to the target's SWAR instructions. Generally, these are trivial to implement and simply hide inlined assembly code which is used to execute a single SWAR instruction. Intel's C/C++ compiler [Int99a][Int99b], MetroWerks CodeWarrior [Met99], Sybase's Watcom C/C++ [Syb99], and the lcc-win32 (`http://www.remcomp.com/lcc-win32`) project all provide some level of support for using MMX and/or 3DNow! instructions via intrinsics.

New class or type definitions which represent a fragment provide a first-class feel to these objects and the operations on them. To do this, class definitions provide overloaded operators, while non-class definitions typically require a modification to the language. Often, these models are built on top of a set of intrinsics, and support only the partitionings and operations directly associated with the targeted hardware. Several compilers employ this technique. The Intel compiler includes class libraries for MMX and SSE. Free Pascal [CaK98] includes predefined array types for MMX and 3DNow!, and extends Pascal to allow some first-class operations on these types. Oxford Micro Devices' compiler for its A236 Parallel Video DSP chip [Oxf96] provides predefined struct types for MMX, but requires modifications to the C language.

Under strict conditions, and with hints from the programmer, some compilers are able to vectorize simple data-parallel loops. This support is in the early stages and is limited in the data types and operations that can occur in the body of the loop. We expect the development of this style of parallelism to follow that of Fortran loop manipulation. The Intel and MetroWerks compilers both provide limited vectorization for MMX. The Intel compiler also supports SSE, and the MetroWerks' compiler supports 3DNow!.

In languages which provide first-class vector objects, both the precision of the data and the number of elements in the vector are virtualized beyond the limits of the target's SWAR extensions. These vectors are treated as first-class objects, with a full set of operations being provided, rather than a subset based on the SWAR instructions available on the target. To the best of our knowledge, it is still true that only the SWARC language provides this type of model.

We have chosen this last approach because we believe it offers the best opportunity for performance gains over a large range of applications and target architectures. Using the SWARC module language and related compilers, users can write portable SIMD functions that will be compiled into efficient SWAR-based modules and interface code that allows these modules to be used within ordinary C programs.

For reasons discussed in our previous work [FiD98], the level of hardware support for SWAR is inconsistent across architecture families. The set of available instructions varies with the precision and type of the data to be processed. Many useful data precisions are not supported at all. Also, instructions useful for implementing SIMD-style conditional branches are often not provided. To support a high-level language such as SWARC, these problems must be efficiently managed by the compiler.

Previously, we have discussed the basic coding techniques needed to achieve efficient execution of general SIMD constructs on any type of SWAR target including 32-bit integer instruction sets that have no explicit support for SWAR. This paper will focus on our work since LCPC 98. Section 2 contains a brief explanation of the SWARC language. Section 3 provides an overview of the organization of Scc, the first implementation of an experimental SWARC compiler. In section 4, we consider the implementation of compiler optimizations which we introduced last year within the Scc framework. In section 5, we discuss some preliminary performance numbers. A brief summary, and pointers to the support code that we have developed, are given in section 6.

2. The SWARC Language

The SWARC language is intended to simplify the writing of portable SWAR code modules that can be easily combined with C code to form a complete application. It does this by allowing the programmer to specify both object precision and SIMD-style SWAR parallelism in a vector-based language which is similar to C. While a complete definition of the SWARC syntax is beyond the scope of this paper, we will briefly describe the most important extensions beyond C's semantics.

The key concept in SWAR is the use of operations on word-sized values to perform SIMD-parallel operations on fields within a word. Because the language should be portable, and the supported field sizes and alignments depend on the target machine, the programmer's specification should only provide *minimal constraints* on the data layout selected by the compiler. Also, as a module language, SWARC code should integrate well with ordinary C code and data structures. While SWARC is based on C, these issues lead to adjustments in the type system, operators, and general structure of the language.

2.1. The SWARC Type System

A SWARC data type declaration specifies a first-class array type whose elements are either ordinary C-layout objects or SWAR-layout integer or floating-point fields with a *specified minimum precision*. The syntax for declaring a SWARC object is similar to the bit-field specification used in C `struct` declarations, and takes the general form: *type*:*prec*[*width*].

The base C type *type* may be any of the C types `char`, `short`, `int`, or `float`. The modifiers `signed`, `unsigned`, and `const`, and the storage classes `extern`, `register`, and `static` can be applied to this base, and have the same meanings as in C. Also, many SWAR targets support operations on natural data types which require *saturation arithmetic*, in which results that do not fit in the range of the data size are set to the nearest storable value. Therefore, SWARC allows either `modular` or `saturation` to be specified as an attribute of the type.

The precision specifier `:` indicates that the object should have a SWAR layout, and that the minimum precision required for the data may be specified with an optional integer precision *prec*. Omitting the precision specifier indicates that the object should have a C, rather than a SWARC, layout. For example, the SWARC C-layout declaration `char c;` is equivalent to the C declaration `char c;`. Using the precision specifier without an integer precision is equivalent to specifying a SWAR-layout with the native precision for the equivalent C-layout type, e.g., `float:` is equivalent to `float:32`. Note that while the compiler may store data with a higher precision than specified, saturation is always to the declared precision of the data.

The optional `[width]` specifier indicates the C-layout array dimension or number of SWARC-layout vector elements. `char:7[14] d;` declares `d` to be a vector with 14 visible elements, each of which is an integer field with *at least* 7 bits precision. If the `[width]` is omitted, it is taken to be one.

These type extensions require several modifications to the C type coercion rules. Scalar objects are promoted to the dimension of the other type. If neither object is a scalar, and the dimensions are mismatched, a warning is generated and the wider object is truncated to the width of the other. Expressions which mix C- and SWAR- layout objects, result in the SWAR-layout even if this requires the precision to be reduced. Otherwise, an expression with mixed precision yields a result with the higher precision. Also, where mixed, `modular` expressions are cast to `saturated` expressions.

2.2. Control Constructs and Statements in the SWARC Language

Control flow constructs in SWARC are a superset of those in C, and operate similarly to those in MPL [Mas91]. From the point of view of the programmer using SWARC, conditionally executed statements must be applied only to those vector elements for which the condition is true. Because SWAR instructions are applied across all the elements stored in a CPU register, a conditionally executed instruction must be applied under an *enable mask* which limits its effects to the elements which are enabled for the operation. SWARC control constructs must be modified to properly deal with this situation and to hide the underlying operations from the applications programmer. The SWARC constructs include:

- *if* statements, which operate as do C `if` statements if the conditional expression has a width of one. Otherwise, the *if* body is executed iff the condition is true for some enabled element of the conditional vector. In this case, the body is executed under enable masking to limit the effects to those elements for which the condition is true. Likewise, the *else* body is executed, under masking, iff the condition is false for some enabled element of the conditional vector.

- *where* statements, which operate as do SWARC `if` statements, except that the *where* and *elsewhere* bodies are always executed. These bodies are masked to limit their effects to the correct set of elements.
- *everywhere* statements, which enable all elements of the vector for the following statement.
- *while* statements, which operate as do C `while` statements if the conditional expression has a width of one. Otherwise, the *while* body is executed as long as the condition is true for at least one enabled element in the vector. An element is disabled when the condition becomes false for that element, and stays that way until the loop is exited. Thus, the set of enabled elements is monotonically non-increasing with each iteration. Once all the elements become disabled, the loop exits, and the enable mask is restored to its condition before entering the loop.
- *for* statements, which are related to the SWARC *while* in the same way that the C *for* is related to the C *while*.
- *do* statements, which are related to the SWARC *while* in the same way that the C *do* is related to the C *while*.
- *continue* and *break* statements, which operate as in C except that an optional expression indicates how many nesting levels to continue or break from.
- *return* statements, which operate as in C except that no expression is allowed to be returned from a SWARC function.
- labels, block statements, and empty statements, which all operate as in C.
- function calls, which operate as in C except that arguments are passed by address, not by value. The call is executed as the body of an implied `everywhere`. This ensures compatibility with ordinary C code.
- A special block statement, which encloses ordinary C code and can be inserted wherever a statement can appear, or as a top-level declaration. These blocks are enclosed by a ${ $} pair, and will be emitted into the output code. Within these blocks, a dollar sign is used wherever a pound sign should appear in the output C code.

2.3. The SWARC Operators

The definition of the standard C operators have been modified to work in a consistent and intelligent way with SWAR data:

- The trinary operator operates as in C, but applies enable masking to block side-effects from disabled fields.
- Logical and comparison operators operate as in C, but return 0 in every false field and -1 (all 1 bits) in every true field. This modification to the C definition makes the implementation of enable masking significantly simpler.
- Bitwise, unary and binary arithmetic, and shift operators all operate as in C, but within each field on vector data.
- The assignment operators work as in C, but are extended as in C* [TMC90a][HaL91] to perform reductions on associative operators when storing a vector value into an array, or when the operator is used as a unary prefix.

New operators have also been added to facilitate operations common to SIMD processing to be performed in the SWAR environment:

- The minimum (?<), maximum (?>), and average (+/) operators have been added to facilitate the computation of these values for vectors, and can be used as assignment operators.
- The operators ||= and &&= have been added to perform the SIMD ANY and ALL operations for assignments and reductions.
- The `typeof` operator returns the type of its expression argument. This allows parameterized functions to be written to handle many types.
- The `widthof` operator returns the declared dimension of its expression argument.
- The `precisionof` operator returns the declared precision of its expression argument.
- The vector element shift ([<<*n*] and [>>*n*]) and rotate ([<<%*n*] and [>>%*n*]) operators have been added to ease the implementation of inter-element communication and similar algorithms.

As a final note, the short-circuit evaluation of the binary and trinary logical operators is optional in SWARC.

2.4. An Example SWARC Function

An example of code that can be written in SWARC is the Linpack benchmark DAXPY loop. While the language does not preclude double-precision data, current SWAR hardware does not support it. Therefore, we will perform a SAXPY (Single-precision AXPY) instead. A C version of the original loop looks like this:

```
for (i = 0;i < 4; i++)
        dy[i] = dy[i] + da*dx[i];
```

In SWARC, the same code is written as a vector expression. Here, we show the code wrapped in a function body which can be in-lined or copied directly into the SWARC source:

```
void swar_saxpy(float:[4] x, float:[4] y, float a) { y += (a * x); }
```

In the next section, we will discuss the organization of the experimental SWARC compiler, Scc, and follow this example as it is compiled to C code.

3. The Organization of the Scc Compiler

The Scc compiler consists of the front end, a back end, and a set of utilities which are used throughout the compiler. The purpose of the front end is to determine what type of processing must be performed on each source file, parse SWARC source code, and convert the SWARC source into a type-coerced, intermediate representation (IR) tree representing the vector operations. The back end has the task of converting the intermediate vector tree form into lists of tuples representing operations on word-sized data *fragments*, and generating C code to implement the operations described by these tuples based on the capabilities of the target architecture.

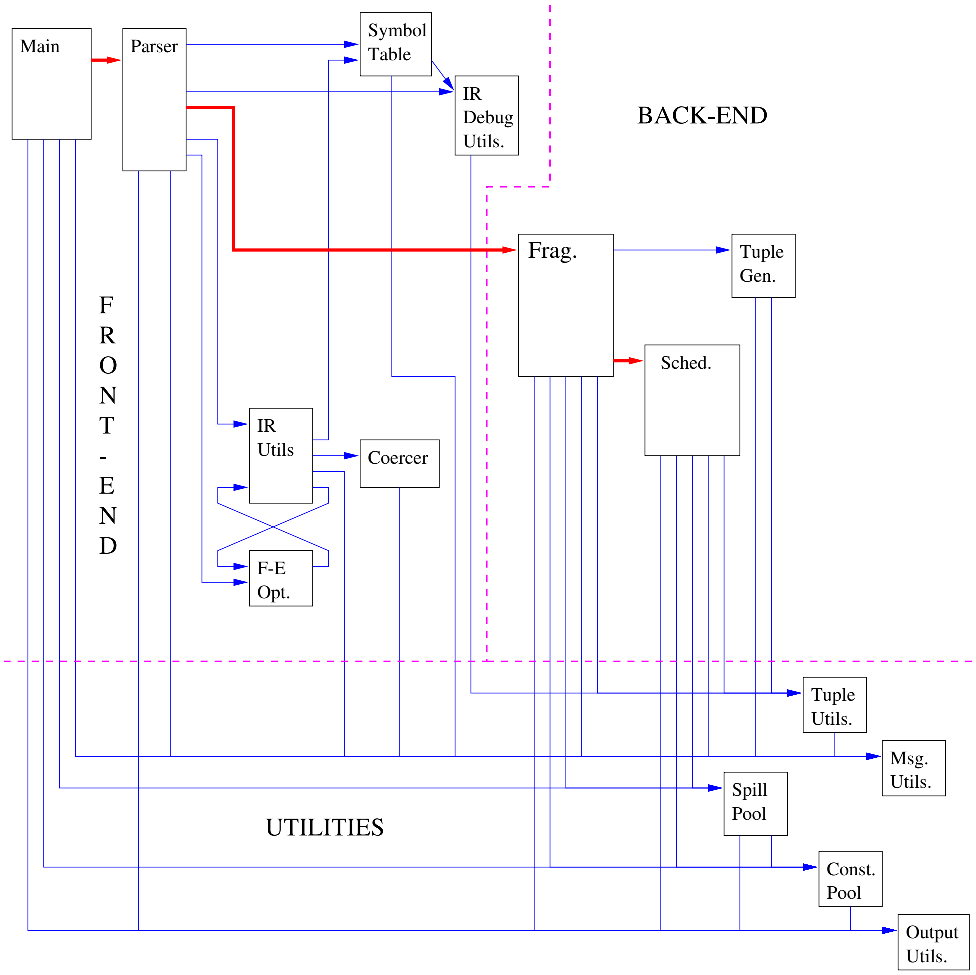

Figure 1: Organization of the Scc Compiler

Figure 1 is a diagram of the compiler showing the functional units as blocks and the calls between them as arrows. An arrow from unit A to unit B indicates that some function in unit A calls a function in unit B. The primary flow of data through the compiler follows the heavy line from the main function, through the parser, the fragmenter, and finally the scheduler. The dashed lines roughly indicate the separation between the front end, back end, and the utilities.

3.1. The Front End

Figure 2 is a diagram of the front end showing its six major functional units.

- The main function, which handles command-line options and determines how each source file is to be handled.
- A SWARC parser, which parses SWARC input and drives the remainder of the front end.
- The symbol table, which stores information on the identifiers in the SWARC source, and provides utilities to the rest of the front end to access this information.
- The coercer, which performs type checking and coercion on the IR tree.

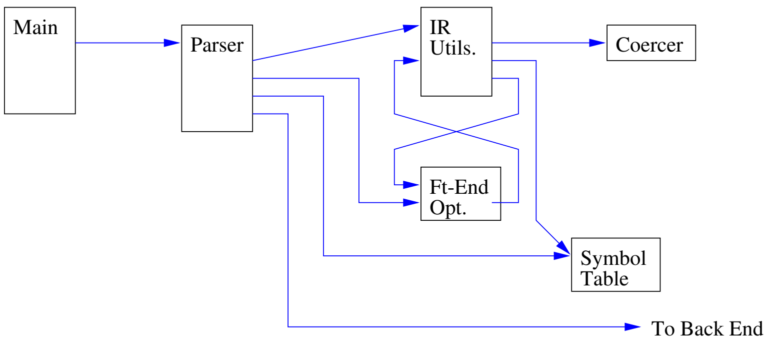

Figure 2: The Front End of the Scc Compiler

- The IR utilities, which generate pieces of the IR tree to be connected by the parser and coercer.
- The front end optimizer, which does constant-folding and other vector-based optimizations on the generated IR tree.

The more interesting of these units will be explained in the following subsections.

3.1.1. The Main Function

SWARC sources are preprocessed if necessary, then passed to the SWARC parser to be translated into C code and written to the output C file. Ordinary C sources are passed through intact to the output C file, and C header code is written to the C header output file, for handling by the C compiler. Library, object, preprocessed or raw assembly code is passed along to the C compiler for linking.

3.1.2. The Parser

The parser was built using PCCTS (the Purdue Compiler Construction Tool Set, see the network newsgroup `comp.compilers.tools.pccts`). As it reads the SWARC source code, it generates top-level declarations and prototypes for the C output. As each function body is parsed an IR tree is built to represent it. This tree contains nodes representing scalar and vector operations, and is optionally passed to the front end optimizer before being passed to the back end for code generation.

3.1.3. The Intermediate Representation Utilities

Several utilities are available to the parser and coercer to help build and restructure the IR tree. The tree has a child-sibling structure in which each operation is stored in a parent node. This node's child will be the top of the tree representing the operation's first operand, and each of the child's siblings will be at the top of a tree representing one of the remaining operands. These operands can be leaves representing a constant or identifier, or trees representing more complex expressions, code blocks, or control constructs.

3.1.4. The Front-end Optimizer

The front-end optimizer reconfigures the IR tree for a function by performing several optimizations. These include constant folding on scalar and vector operations, removal of code to compute conditionals with constant values and the related unused conditional bodies, and aggressive vector-level algebraic simplification. These optimizations depend not only on the type of the values, but also on their precision and size. We discuss these optimizations in more detail in section 4.

3.1.5. An Example Intermediate Representation Tree

Figure 3 is a representation of the IR tree that the front end generates for our SAXPY example. The notation “4x32f” indicates an entity or operation which has four fields containing 32-bit floating point values. We see that a 4x32f add is performed on the 4x32f value loaded from memory location *y* and the product of the scalar (1x32f) *a*, casted to a 4x32f, and the 4x32f *x*. The 4x32f result is then stored in memory location *y*.

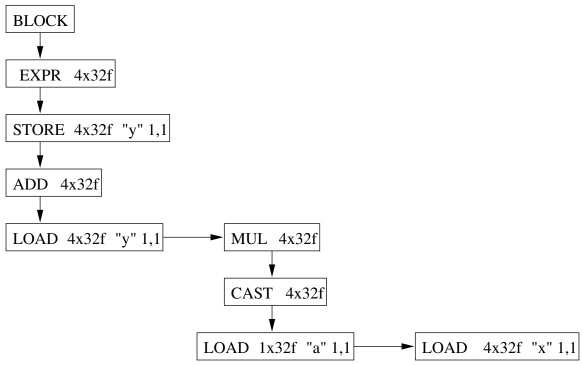

Figure 3: IR tree for SWAR SAXPY

3.2. The Back End

Figure 4 is a diagram of the back end showing its three major functional units.

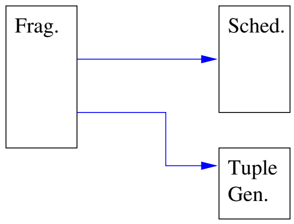

Figure 4: The Back End of the Scc Compiler

- The fragmenter, which divides vector data into word-sized *fragments*, and drives the other parts of the back end.
- The tuple generator, which generates a tuple tree for each fragment.
- The scheduler, which schedules the tuples and generates output code.

These units will be explained in the following subsections.

3.2.1. The Fragmenter

The primary task of the fragmenter is to convert the IR tree into lists of *tuple DAGs*, which more closely represent the operations in the output language. SWARC data is generally in the form of vectors which contain more elements than can be stored in a single CPU register. These vector operations are encoded in the IR tree as single tree nodes, but need to be converted into a series of equivalent operations which operate on the word-length *fragments* of the vector arguments. These operations are stored as trees of tuples in the master tuple list, and are indexed via an array of tuple indices for each vector operation. Once the tuple lists for the operations in a basic block have been generated, the fragmenter passes the tree of lists to the scheduler to be converted into C output code.

The actual fragmentation of vector operations is done by a set of functions, each of which is dedicated to handling one type of operation. These functions allocate an array to hold the tuple index list for the vector operation, then construct that list by repeatedly calling the tuple generation functions for each fragment of the operation. This produces a list which is constructed in a manner dependent on the language construct or operation being fragmented.

Thus, what we call fragmenting is very closely related to strip mining and serves a similar purpose. The primary difference is that fragmenting does not generate any loops, nor does it generate the associated indexing or conditional end-of-vector tests. It simply generates (large numbers of) fragment-based operations that have the minimum possible overhead and maximum flexibility in scheduling. While we have found no previous references to fragmenting, it seems to be too obvious to be new. Future versions of Scc may use strip mining in combination with fragmenting for very long vectors, where excessive fragmented code size might limit performance.

In figure 5, we see how an 8-element vector addition is fragmented into four word-sized parallel additions. In the diagram, and this discussion, the notation *n-fxb* indicates an entity with *n* parts, each of which has *f* fields of *b* bits. In the top half of the figure, a single vector addition is conceptually applied to two vectors, A and B, each of which has eight 32-bit data elements. Assuming that the target's registers have a width of 64 bits, the fragmenter can only pack two 32-bit fields into each fragment as a 2x32 SWAR entity. The lower half of the figure shows how the vector is fragmented, with each pair of elements assigned to a single fragment. The corresponding fragments of the two vectors are then added with a single hardware operation.

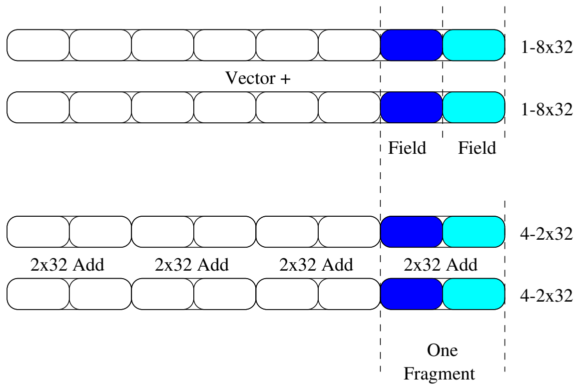

Figure 5: Fragmentation of a Vector Addition

3.2.2. The Tuple Generator Functions

Each tuple generator function is responsible for constructing a tuple tree to perform an operation on one fragment of a vector, and providing this tree's index to the fragmenter. The trees generated by these functions are stored in the master tuple list, with each node representing data or one of the sub-operations to be performed in the execution of the operation. Any arguments to the operation should already be stored as tuple trees, and are accessed via the indices to these trees. Thus, a sub-operation is stored as a tree node whose children are its operands, and whose parents are the operations or sub-operations which require its results.

In previous work, we have shown that the available operations vary widely across SWAR target architectures and often support only certain data precisions. These variations must be accounted for during the construction of the tuple DAGs in the fragmentation phase.

This may be done either through the promotion of data to supported field sizes or through the emulation in software of unsupported operations using techniques which we have developed [FiD98]. Scc employs both of these methods during tuple generation.

Common subexpression elimination is performed in these functions by reusing previously generated, equivalent tuple trees when possible. Reduction in strength optimizations can also be performed in these functions; however, care must be taken, because these optimizations depend on the availability of an instruction with the correct data type and field width. Finally, several compiler optimizations can be applied at the fragment level during the generation of tuples to lessen the overhead of enable masking and spacer manipulation, or to take advantage of the special situations created by the use of fragmentation, spacer bits, and enable masks. These optimizations will be discussed in section 4.

3.2.3. The Scheduler

Once a tuple tree list for a basic block has been generated, the fragmenter calls the scheduler to generate output code for the list. The combined scheduler/register-allocator then performs a modified exhaustive search of the possible schedules for the tuple list based on schedule permutation [NiD90]. A detailed model of the target pipeline is used to estimate the cost of each schedule. At first, the scheduler attempts to schedule the block without allowing any register spills. If this is not possible, it tries again allowing one spill, then two, etc. until it is able to schedule the block, or until a precompiled maximum number of spills is reached.

In each of these iterations, the scheduler attempts to find an optimal schedule by first building an initial schedule by back-tracing the stores, then trying to improve upon this schedule. It starts its attempts at improvement by placing certain restrictions on the types of memory access modes allowed, then performing a search for the best schedule possible under these restrictions. If no schedule can be found, then the restrictions are loosened by a degree, and the scheduler tries again. This relaxation process continues until the scheduler finds that it cannot schedule the block without spilling more registers than it is currently allowed.

Once a schedule for the basic block is found, output code is generated for it. This schedule is known to be optimal for the target architecture based on the pipeline cost estimation. This cost estimate takes into account emulation overhead, multiple pipeline usage, target-specific instruction costs, operand source differences, and costs related to register renaming.

Unfortunately, on average, our current cost estimation model over-estimates the expected cost of memory references, sometimes ignoring shorter pipeline estimates because it thinks they are not visible, when in fact they are. This estimation is based on the assumption that memory accesses will always hit in the second-level cache, when many will hit in the first-level cache, thereby returning sooner than the cost estimation predicts. This causes the scheduler to choose non-optimal code sequences in certain situations.

3.3. Example of C Output from Scc

Returning to our SAXPY example, the generated C code for the SWAR version of the loop targeting an AMD K6-2 (with four elements to keep it brief) is:

```
void swar_saxpy(p64_t *x, p64_t *y, float *a)
{
    register p64_t *_cpool = &(mmx_cpool[0]);
    {
        movq_m2r(*(((p64_t *) a) + 0), mm0);
        pand_m2r(*(_cpool + 2), mm0);
        movq_r2r(mm0, mm1);
        psllq_i2r(32, mm0);
        por_r2r(mm0, mm1);
        movq_r2r(mm1, mm2);
        pfmul_m2r(*(((p64_t *) x) + 1), mm1);
        pfmul_m2r(*(((p64_t *) x) + 0), mm2);
        pfadd_m2r(*(((p64_t *) y) + 1), mm1);
        pfadd_m2r(*(((p64_t *) y) + 0), mm2);
        movq_r2m(mm1, *(((p64_t *) y) + 1));
        movq_r2m(mm2, *(((p64_t *) y) + 0));
    }
_return: femms();
}

p64_t mmx_cpool[] = {
        /*    0 */      0x0000000000000000LL,
        /*    1 */      0xffffffffffffffffLL,
        /*    2 */      0x00000000ffffffffLL,
        /*    3 */      0xffffffff00000000LL
};
```

The first five statements inside the inner block load the 32-bit float value `a` into both fields of a 64-bit register. The sixth copies this value for use with another fragment. The remaining instructions perform the SAXPY on the two fragments of the vector data in `x` and `y`. Note that the above code is not optimally scheduled due the aforementioned errors in the current cost estimation code.

4. Implementation of Compiler Optimizations For SWAR

Last year, at LCPC 98, we introduced and discussed several static compiler optimizations that apply to SWAR programming. These were based on tracking data, spacer, and mask values, and aggressively simplifying code dealing with spacers and masks. While some of these techniques can only be applied for particular types of targets and field sizes, others apply to all targets, and some can be implemented at both the vector and fragment levels.

In this section, we discuss how these optimizations have been, or will be, implemented in the Scc experimental compiler. We will briefly reintroduce these optimizations here, but refer you to [FiD98] for a more detailed discussion. Three such optimizations are: promotion of field sizes, SWAR bitwise value tracking, and enable masking optimization.

4.1. Promotion Of Field Sizes

In SWARC, the application programmer may specify the minimum number of bits of precision required for a value. The compiler may choose to use more bits for storage of these values in order to make use of specialized hardware on the target architecture. For example, 16 bit values are handled very well by HP's PA-RISC MAX [LeH96], but smaller sizes are not. Thus, a vector that was declared as containing 14-bit values will yield more efficient code sequences for MAX if the array's object type is promoted to 16-bit fields.

In general, for each SWAR target, there are certain field sizes that are less efficient than a somewhat larger field size, and thus none of these inefficient field sizes should be directly used. However, not all smaller field sizes are less efficient than a larger size; In some cases, the field size is small enough that the gain in parallelism outweighs the overhead of using an unsupported field size.

Currently, the Scc compiler targets only IA32 architectures with 64-bit MMX-based enhancements including MMX [Int99a], 3DNow! [AMD97], and Extended MMX [Cyr96]. These systems only support field sizes of 8-, 16-, 32-, and 64-bits directly. Data of any other field size must either be promoted, or operations on it will have to be emulated by the compiler. The Scc compiler emulates 1-, 2-, and 4-bit support, as these field sizes are reasonably efficient. All other unsupported field sizes are promoted to the next larger supported or emulated field size. One of our goals is to determine which unsupported sizes can be emulated efficiently, and add code to perform this emulation based on the techniques introduced in [FiD98].

Certain field sizes do not provide any added parallelism over the next larger size. This is true whenever the floor of the register size divided by the field size is equal for both field sizes. In this case, it is only beneficial to emulate the smaller field size if the extra bits can be used to increase the number of spacer bits between data fields for use with spacer value tracking [FiD98], or if the necessary emulation code can be significantly simplified. Otherwise, promoting the data to a higher precision allows the total amount of emulation code to be minimized without affecting the performance of the output code.

In Scc, promotion of data precision is performed in the back-end during the fragmentation phase, because this is the first point in the compilation process that the parameters of the target are known. The promotion depends not only on the size of the target's registers, but also on the set of supported field sizes and the set of instructions which operate on them.

4.2. Vector Algebraic Simplification and Bitwise Value Tracking

Last year, we introduced the topic of bitwise value tracking as it related to the optimization of compiler inserted masking operations. These are primarily composed of bitwise `AND` and `OR` operations and left and right shift operations, using constant-valued masks and shift distances. We now generalize this idea to apply to a larger set of operations and types of data. A masking example still suffices to convey the idea. Consider the following example in C in which the low byte of a 16-bit word is moved into the high byte with masking:

```
x = (( (x & 0x00ff) << 8 ) & 0xff00);
```

Simple constant folding will not improve the above code because no single operation has two constant operands. However, by aggressively applying the algebraic properties of the operations involved, we can restructure the code so that constant folding can be applied. Distributing the shift over the inner expression yields:

```
x = (( (x << 8) & (0x00ff << 8) ) & 0xff00);
```

which can be folded to:

```
x = (( (x << 8) & 0xff00 ) & 0xff00);
```

From here, we see that the `AND` operations can be folded because they are associative and each has a constant operand. In this particular example, they also happen to have the same value, although this is not true generally. The code is finally converted to the equivalent, but simpler, form:

```
x = ((x << 8) & 0xff00);
```

Note that unless we are able to fold the operations at each step, we will be simply replacing one set of operations with an equal number of different operations which are probably equally expensive. We have identified a strict set of conditions which must be met to make this optimization worthwhile:

- The top-level operation *op1* must have one operand which evaluates to a constant value, and another which is a tree rooted at an operation *op2*.
- *op2* must have one operand which evaluates to a constant, and a second which is a tree rooted at an operation *op3*, over which *op2* is distributive.
- *op3* must have one operand which evaluates to a constant, and be associative with *op1*. Note that *op1* and *op3* may differ. For example, in 1-bit fields, additions and exclusive-ORs are associative.
- Other restrictions may be imposed due to the exact form of the expression tree and the asymmetry of any of the operation's properties. For example, *op1* or *op3* may be required to be commutative so that operands may be reordered and associative combining of operations applied.

After ensuring that the above conditions are met, the algorithm to perform this optimization on an expression tree has four basic steps:

- Distribute *op2* over *op3*.
- Reorder the tree if necessary, depending on the commutative properties of *op1* and *op3*.
- Combine *op1* and *op3*.
- Perform constant folding on the tree.

After this last step, *op1* has been eliminated from the tree. This process can then be continued up the expression tree in the attempt to remove more operations.

In Scc, this optimization can be applied at the vector level to algebraically simplify vector operations, and again at the fragment level to optimize masking and spacer operations on the tuple trees for each fragment.

4.3. Enable Masking Optimizations

The individual fields of a SWAR register cannot be disabled. Thus, undesired field computations must be arithmetically nulled thereby incurring a significant cost. This cost can be avoided on a per fragment basis if the compiler can prove statically that one of two conditions hold:

- All fields of the fragment are enabled. In this case, no masking is done, and the corresponding fragment of the result is directly computed.
- All fields of the fragment are disabled. In this case, no masking is done, and the corresponding fragment of the result is copied from the original value.

Otherwise, the masking may still be avoided if the compiler can generate code that allows all fields to be active and later corrects the *inactive* field values.

In Scc, conditional statements, such as `ifs` generate multiple tuple lists for each word-sized fragment of the vector: one for evaluating the conditional value, one for the body, and one for each possible alternative such as an `else` clause. Because of this, Scc makes it easy to apply the above algorithm. During the fragmentation phase, when the conditional code is being generated, the compiler performs an analysis which reveals fragments fitting either of the two above cases. These fragments are noted, and enable masking operations are skipped during the generation of code for the fragment's body and alternatives.

5. Performance Evaluation

At the time of this writing, we have relatively few direct comparisons of performance — it is only recently that vectorized support has been added to the Intel C/C++ and MetroWerks Code Warrior compilers. Prior to these additions, Scc was the *only compiler* providing support for multi-word objects using the MMX and 3DNow! instruction set extensions. Prototype versions of the compiler have been available on the WWW since August 1998, and hundreds of uses of the WWW-interfaced version have been recorded. The most common comments are very positive, with a few very negative comments about Scc's early inability to handle very long vectors or to generate good error messages. In addition, we have data for two benchmarks, one integer MMX and one floating point 3DNow!

5.1. The Integer Benchmark — Subpixel Rendering

The pixels of color LCD (Liquid Crystal Display) panels are not square dots capable of displaying any color; rather, each pixel actually consists of three separate subpixels, one each for red, green, and blue. Generally, the subpixels are arranged as rectangular stripes that are 1/3 as wide as they are tall. Using this fact and rendering at 3 times the horizontal resolution, image quality can be dramatically improved.

The problem is that treating the subpixels like full pixels yields severe color fringing. To remedy this, we use a 5-point software filter that applies 1/9, 2/9, 3/9, 2/9, 1/9 weightings. Unfortunately, the filter is relatively expensive, partly because the weightings are awkward, but also because the memory reference pattern for subpixels has a stride of three bytes.

We constructed an optimized serial C version of this code for use with the PAPERS video wall library. However, we also assigned this problem to 16 individual students as part of a SWARC project in the "Programming Parallel Machines" course (Spring 1999 in Purdue's School of Electrical and Computer Engineering). The result was that at least a few of the students achieved more than 5x speedup over the optimized C code using Scc-generated MMX code. This was a surprisingly good result; in fact, even though some students wrote their own MMX code by hand, the fastest version used *unedited* Scc-generated code.

5.2. The Float Benchmark — Linpack

The Top 500 Supercomputers list (`http://www.top500.org/`) uses a version of Linpack to rank the floating point speeds of a wide range of machines. An updated version of the standard C source which corrects for problems with the timers on PC-compatible systems, is available via ftp from `ftp://ftp.nosc.mil/pub/aburto/linpackc/linpackc.c`. This version achieved 54 MFLOPS operating on 32-bit fields on our K6-2 400MHz test platform.

Rewriting just a few lines of code in SWARC (a 100-element chunk of the core DAXPY, DDOT, and DSCAL loops) and using Scc to generate 3DNow! code, the performance went to something around 80-100 MFLOPS, yielding a speedup of at least 48% over the C version. However, that performance was significantly hindered by our scheduler's overly-conservative estimations of load cost; by simply hand-tweaking the 3DNow! code schedule, we achieved more than 220 MFLOPS, a potential 307% speedup with corrected scheduling. Because memory bandwidth, etc., are also important factors, it is difficult to say how much closer to the machine's theoretical peak 1.6 GFLOPS could be achieved. Still, the performance we obtained is impressive by any standard— especially because our reference platform is a $2,000 laptop!

6. Conclusion

The SWARC language was developed to provide a portable, virtualized, high-level language for SWAR-based parallel processing. This language has been successfully targeted to IA32+MMX-based architectures via the experimental Scc module compiler. This compiler provides a framework for further research on SWAR processing such as emulation of unsupported operations and field sizes, optimization of vector and fragment operations, and instruction scheduling for SWAR-capable targets. In this paper, we have briefly described the SWARC language and the operation of the Scc compiler as it converts vector SWARC source code to scheduled, fragment-based C code modules.

We are currently expanding the targets of the Scc module compiler from MMX-based processors to generic 32-bit integer processors and processors enhanced with 128-bit extensions such as Intel's SSE [Int99a] and Motorola's AltiVec [Mot98]. At this writing, however, only basic libraries for using MMX, Extended MMX, and SSE with Gnu C have been released. These libraries, and a web interface to a recent version of the Scc compiler, can be accessed at `http://shay.ecn.purdue.edu/~swar/Index.html`.

References

[AMD97] Advanced Micro Devices, Inc., *AMD-K6 Processor Multimedia Extensions*, Advanced Micro Devices, Inc., Sunnyvale, California, March 1997.

[CaK98] Michael Van Canneyt and Florian Klampfl, *Free Pascal supplied units: Reference Guide*, `http://rs1.szif.hu/~marton/fpc/units`, December 1998.

[Cyr96] Cyrix Corporation, *Multimedia Instruction Set Extensions for a Sixth-Generation x86 Processor*, Cyrix Corporation, `ftp://ftp.cyrix.com/developr/hc-mmx4.pdf`, August 1996.

[FiD98] Randall J. Fisher and Henry G. Dietz, "Compiling For SIMD Within A Register", 11th International Workshop on Languages and Compilers for Parallel Computing, Chapel Hill, North Carolina, August 1998.

[HaL91] P. J. Hatcher, A. J. Lapadula, R. R. Jones, M. J. Quinn, and R. J. Anderson, "A Production Quality C* Compiler for Hypercube Multicomputers", *Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Williamsburg, Virginia, April 1991, pp. 73-82.

[Int99a] Intel Corporation, *Intel Architecture Software Developer's Manual: Vol. 1 Basic Architecture*, Intel Corporation, `http://developer.intel.com/design/pentiumII/manuals/24319002.PDF`, May 1999.

[Int99b] Joe Wolf, *Coding Techniques for the Streaming SIMD Extensions With the Intel C/C++ Compiler*, Intel Corporation, `http://developer.intel.com/vtune/newsletr/methods.htm`, July 1999.

[Int99c] Joe Wolf, *Advanced Optimizations With the Intel C/C++ Compiler*, Intel Corporation, `http://developer.intel.com/vtune/newsletr/opts.htm`, July 1999.

[LeH96] Ruby Lee and Jerry Huck, *HP Technical Computing — 64-bit and Multimedia in the PA-RISC 2.0 Architecture*, Hewlett-Packard Company, `http://hpcc997.external.hp.com:80/wsg/strategies/pa2go3/pa2go3.html`, June 1996.

[Mas91] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 2.2*, Document Number 9302-0001, Sunnyvale, California, November 1991.

[Met99] MetroWerks, Inc., *MetroWerks Desktop Products - CodeWarrior for Windows, Profession Edition*, MetroWerks, Inc., `http://www.metrowerks.com/desktop/windows/`.

[Mot98] Motorola, Inc., *AltiVec Technology Programming Environments Manual, Preliminary Rev. 0.2*, Motorola, Inc., `http://www.motcom/SPS/PowerPC/teksupport/teklibrary/manuals/altivec_pem.pdf`, May 1998.

[NiD90] A. Nisar and H. G. Dietz, "Optimal Code Scheduling for Multiple Pipeline Processors", 1990 International Conference on Parallel Processing, vol. II, Saint Charles, Illinois, pp. 61-64, August 1990.

[Oxf96] Oxford Micro Devices, Inc., *New Method for Programming Intel Multimedia Extensions (MMX)*, Oxford Micro Devices, Inc., `http://oxfordmicrodevices.com/pr040396.html`.

[Syb99] Sybase, Inc., *Watcom C/C++ 11.0 Datasheet*, Sybase, Inc., `http://www.sybase.com:80/products/languages/cdatash.html`.

[TMC90a] Thinking Machines Corporation, *C* Programming Guide*, Thinking Machines Corporation, Cambridge, Massachusetts, November 1990.