

A Vectorizing Compiler for Multimedia Extensions

N. Sreraman *

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
U.S.A.

sreraman@microsoft.com

R. Govindarajan

Supercomputer Education and Research Centre
Dept. of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012 INDIA

govind@{serc,csa}.iisc.ernet.in

Abstract

In this paper, we present an implementation of a vectorizing *C* compiler for Intel's MMX (Multimedia Extension). This compiler would identify data parallel sections of the code using scalar and array dependence analysis. To enhance the scope for application of the subword semantics, our compiler performs several code transformations. These include strip mining, scalar expansion, grouping and reduction, loop fission and distribution. Thereafter inline assembly instructions corresponding to the data parallel sections are generated. We have used the Stanford University Intermediate Format (SUIF), a public domain compiler tool, for our implementation.

We evaluated the performance of the code generated by our compiler for a number of benchmarks. Initial performance results reveal that our compiler generated code produces a reasonable performance improvement (speedup of 2 to 6.5) over the the code generated without the vectorizing transformations/inline assembly. In certain cases, the performance of the compiler generated code is within 85% of the hand-tuned code for MMX architecture.

KEY WORDS: MMX Instruction Set, Subword Parallelism, Vectorizing Compiler.

*The work reported in this paper was done when this author was at the Dept. of Computer Science and Automation, Indian Institute of Science, Bangalore, India.

1 Introduction

Multimedia is the integration of visual (video, images, animation), audio (music, speech) and textual information. It is basically information represented in different ways by these different media data-types. *Media processing* [1] is the decoding, encoding, interpretation, enhancement and rendering of digital multimedia information. This information is typically in the form of large volumes of low precision or short data-types. The large volume of data makes compression a necessary step before storage. This translates into *computational burden* for real-time retrieval of the data. Of late, media processing applications have been dominating the personal computing domain. They are characterized by [2]

- small native data types
- large data-set sizes
- large amount of inherent data parallelism
- computationally intensive features
- multiple concurrent media streams
- large I/O requirements

These applications have traditionally been supported by special-purpose chips, also known as media-processors, but with the rise in the fraction of such applications, it becomes necessary to enhance their performance preferably without an increase in the cost, and hence without the support of a special hardware.

This high computational demand on short data types for media applications has been effectively addressed by modern processors by the introduction of subword parallelism [3]. Subword parallelism is a specific instance of data parallelism in which a *data word* is the data-set. Subword parallelism is exhibited by instructions which act on a set of lower precision data packed into a word, resulting in the parallel processing of the data-set. Most of the current processors support 64-bit *words*. Although Intel x86 processors support 32-bit words, they have 64-bit floating-point units [4]. The word size of processors determines the width of general purpose registers and data-paths for integer and address calculations. The large word size facilitates higher precision arithmetic. Subword parallelism allows the processor to exploit this large word size even when not handling high precision data. Figure 1 illustrates an addition instruction exploiting subword parallelism. This instruction would take the same amount of clock cycles

as a traditional add instruction, and effectively performs 4 operations in parallel on 16-bit data operands. In order to support and exploit subword parallelism, modern processors extend their Instruction Set Architecture. These extensions, popularly referred to as the multimedia extensions, have now been accommodated in their processors by many vendors, e.g., Intel's MMX (MultiMedia eXtension) [5], Sun's VIS (Visual Instruction Set) [6], Hewlett Packard's MAX-2 (Multimedia Acceleration eXtension) [7] and PowerPC's AltiVec [8]. Since the same instruction applies to all data-elements in the word, this is a form of small-scale SIMD (Single Instruction Multiple Data).

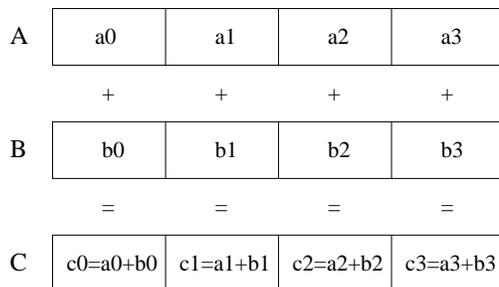


Figure 1: Packed Add $C = A + B$

An application written in a high-level language would not benefit from these extensions to the ISA, unless the compiler generates object code making use of these instructions. Unfortunately, this has not been the case for subword parallelism. Vectorization technique [9], which has traditionally been used by compilers for vector and SIMD machines, can be applied for this purpose. In simple terms, a vectorizing compiler identifies instructions in the loop, whose successive instances can be executed in parallel, without affecting the semantics of the program.

In the absence of compiler support for subword parallelism, the application programmer is currently forced to write his application at assembly level, which is both tedious and error prone. Some support [1] for these extensions come from the vendors in the form of:

Enhanced System Libraries: Selected library routines are hand-coded in assembly to exploit the extended set of instructions.

Macro Calls for Extended Set of Instructions: The system header files define a set of macros that provide a higher level interface to the extended set of instructions.

In case of hardware supported enhanced libraries, the programmer can make use of a system version of some function calls which exploits subword parallelism within the function. However, this loses out certain opportunity that a vectorizing compiler can achieve. For example inlining

a function may improve the parallelism and reduce the function overhead. A compiler may be able to exploit this enhanced parallelism while inlining would not be possible in hardware enhanced library functions since the source code would not be available. Using macro calls in program to exploit subword parallelism require the user to be aware of the code segment which can be optimized by the multimedia extensions and the macros provided. Further, the code transformations have to be performed manually. Lastly, programming with the macro calls is as hard as with the assembly equivalent.

The above reasons strongly motivate the need for a vectorizing compiler as a general approach for exploiting the subword parallelism. Further supporting different architectures as well as changes in the multimedia instruction set in this approach would require modifications only to the code generation module. This also allows easy portability of the application. Lastly, compiler support approach makes the process (of vectorizing) transparent to the user, reduce the errors associated with assembly coding and improve the performance of applications.

In this paper, we have designed and implemented a source to source vectorizing *C* compiler for Intel's MMX. The compiler takes a *C* source file as input. Various code transformations such as strip mining, scalar expansion, condition distribution are applied [10, 11, 12]. The output is a *C* source file, with the data parallel sections coded in inline assembly. This allows the rest of the code to be optimized by the native C compiler, thus leveraging the optimization techniques implemented in production compilers. The vectorizing compiler supports conditional constructs and performs expression reduction.

We used the Stanford University Intermediate Format (SUIF) [13], a public domain compiler tool, for our implementation. We evaluated the performance of the code generated by our compiler for a number of benchmarks (kernels and multimedia applications). Initial performance results reveal that our compiler generated code produces a reasonable performance improvement (speedup of 2 to 6.5) over the code generated without the vectorizing transformations/inline assembly. In certain cases, the compiler generated code is within 85% of performance of hand-tuned code code for MMX architecture.

The rest of the paper is organized as follows. In Section 2 we present the necessary background required on vectorization techniques and multimedia extension. Section 3 discusses the vectorization techniques as applied for MMX. In Section 4 we present the results obtained for some multimedia kernels. We discuss related works in Section 5. Concluding remarks and directions for future work are provided in Section 6.

2 Background

This section provides the background required to understand the rest of the paper.

2.1 Dependence Relations

The control flow in a program is represented by a *Control Flow Graph* [14, 15] which is a directed graph, where each node could be a statement or a sequence of statements, based on the level of abstraction, and an edge represents a control transfers between a pair of nodes. Control dependence, which can be derived from the control flow graph, restricts the order of execution of statements in a program. A statement S' may or may not be executed based on the execution of a statement S . This represents that statement S' is control dependent on S .

Two statements S and S' are said to be data dependent if there is one access each in S and S' to the same location and at least one of the two accesses is a write. Data dependences are represented by a *Data Dependence Graph* whose nodes are the statements of the program and directed edges represent dependences. Data dependences, henceforth referred to simply as dependences, can be classified as (i) true dependence, (ii) anti dependence, and (iii) output dependence [14]. The arcs of the data dependence graph are classified as *forward* and *backward* arcs. An arc or dependence from S to S' is said to be *lexically forward* when S' follows S in the program order and is said to be *lexically backward* when S follows S' in the program order. As long as the control flows along the program sequence, the dependence arcs will be lexically forward but control transfers against the program sequence, as in the case of a loop, can introduce lexically backward arcs.

Consider the example code¹ shown in Figure 2(a). In the dependence graph for this code, shown in Figure 2(b), solid lines represent data dependences and the dotted lines the control flow. It can be seen that the arc from S2 to S3 is lexically forward and the arc from S2 to S1 is lexically backward.

Array elements are typically *defined* and *used* by statements in a loop. These statements are usually executed more than once. It therefore becomes necessary to talk about *instances* of execution of the statement. The instances are identified by an *iteration vector* [11].

Index variable iteration vector: Iteration vector of the form (i_1, i_2, \dots, i_k) , where i_1, i_2, \dots are the values of the loop indices enclosing the statement, ordered from outer to inner. In

¹In all our discussion, we consider only *normalized loops* — loops whose indices run from 1 to some N , with a unit loop index increment or stride — for the sake of simplicity. The SUIF compiler framework which was used in our work takes care of normalizing the `for` loops.

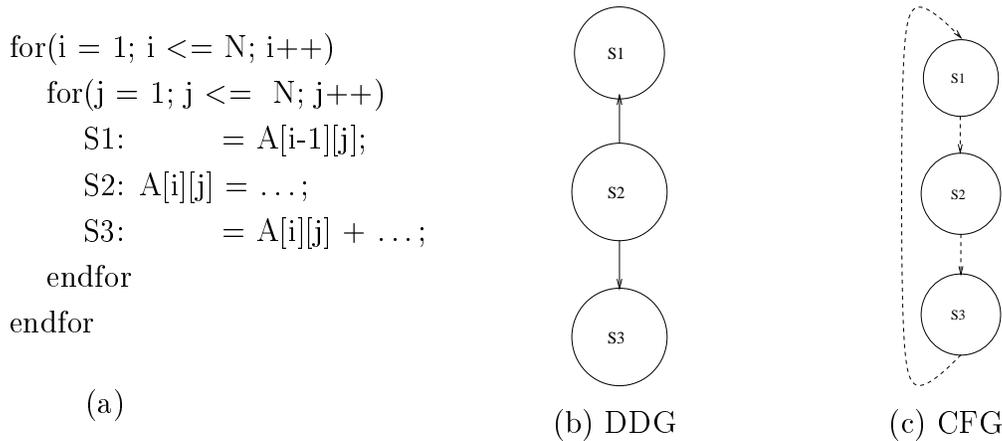


Figure 2: An Example Code (a) and its Data Dependence (b) and Control Flow Graphs (c)

the example in Figure 2(a), the (normalized) iteration vectors for statement S1 are $(1, 1)$, $(1, 2), \dots, (1, N)$, $(2, 1), \dots, (2, N)$, $(N, 1), \dots, (N, N)$.

Consider the data dependence from S2 to S1 in the example. It can be seen that the dependence would not have been present in the absence of the enclosing loops. Such dependence are said to be *loop-carried*. More precisely, a data dependence is said to be loop-carried if a value produced by a statement S in the current iteration is used by statement S' in a future iteration. The dependence arises because for at least a pair of different iteration vectors, say v_1 and v_2 , both S1 and S2 access the same memory location. Dependence analysis has been studied extensively in the literature [9, 11, 16, 17, 18, 19, 20], and the details are summarized in [10, 11, 12]. Lastly, the level of a dependence is defined as the loop nest level, numbered from outermost to innermost, that carries the dependence [10, 11].

Each dependence arc is associated with

kind : true, anti or output dependence,

direction : lexically forward or lexically backward,

level : the enclosing loop which is the cause of the dependence, and

distance : the separation distance at the level of dependence, valid only if it is a constant.

In our data dependence graph, the first three attributes are represented. The direction is represented by means of the directed arc, the other attributes are augmented to the arc by the symbol d_k^l , where l represents the level of dependence and k represents the kind of dependence and can be t(rue), a(nti), or o(utput).

Next, we briefly review Stanford University Intermediate Form (SUIF) which is used as the compiler framework in our implementation.

2.2 The SUIF Compiler Framework

SUIF (Stanford University Intermediate Format) compiler system is a platform for research on compiler-techniques for high-performance machines [21]. SUIF is a research compiler used for experimenting and developing new compiler algorithms. It fosters code reuse, sharing, and modularity. The compiler is structured as a small *kernel* plus a *toolkit* consisting of various compilation analysis and optimizations built using the kernel. The kernel performs three major functions:

- *Defines an intermediate representation of programs:* The program representation is designed to support both high-level program restructuring transformations as well as low-level analyses and optimizations.
- *Supports a set of program manipulation primitives:* These routines are used for performing several transformations.
- *Structure the interface between different compiler passes:* Compilation passes are implemented as separate programs that communicate via files, termed as SUIF files. SUIF files always use the same output format so that passes can be reordered simply by running programs in a different order. Different passes can communicate by annotating the program representation.

The SUIF kernel provides an object-oriented implementation of the SUIF intermediate format. The intermediate format is a mixed-level program representation. Besides the low-level constructs such as SUIF instructions, this representation includes three high-level constructs: loops, conditional statements and array access operations.

We have made use of the following SUIF passes in our work :

- `scc` is the driver for the SUIF ANSI C and FORTRAN 77 compiler.
- `porky` makes various transformations to the SUIF code. The purpose of the transformations could either be to allow subsequent passes to make simplifying assumptions, such as the assumption that there are no branches in a loop body or try to rearrange the code to make it easier for subsequent passes to get information without getting rid of any particular construct.

- `s2c` to read the specified SUIF file and print-out its translation into the Standard C language. We have augmented this pass to print inline assembly code for data parallel sections.

2.3 Intel MMX

In this section we present an overview of Intel's MMX (Multimedia Extension) and its different facets, namely the register stack, the data types supported and the instruction set [4].

2.3.1 Multimedia Registers

The MMX register set consists of eight 64-bit registers, aliased onto the registers of the floating-point register stack. MMX instructions access these registers directly using the register names MM0 through MM7. While operating in the MMX mode, the aliasing mechanism would ensure that accessing these registers as floating point units would result in NaNs (Not a Number).

2.3.2 Multimedia Data Types

Intel has introduced the following new 64-bit quantities

- Packed Bytes : eight bytes packed into the 64-bits.
- Packed Words : four 16-bit words packed into 64-bits.
- Packed Double-Words : two 32-bit double-words packed into 64-bits.
- Quad Word : one 64-bit quantity

2.3.3 MMX Instruction Set

The MMX instructions can be classified as

Data Transfer Instructions: The `MOVD` and `MOVQ` instructions move packed data (respectively 32 and 64 bit data) between MMX registers and memory or between MMX registers and themselves. The 32-bit data transfer instructions always move the low-order 32 bits of the MMX register. The register-to-register version of the `MOV` instruction implementation the operation of moving data between MMX and integer registers.

Arithmetic Instructions: These instructions include instructions to perform add, subtract, and multiply on packed operand types. These instructions exist for the three packed operand types, namely byte (8-bit), word (16-bit) and double word (32-bit), performing 8, 4 or 2 operations in parallel, as illustrated in Figure 1. Each operation is independent of the others and comes in three modes, unsigned saturation, signed saturation, and wrap-around. In the wrap-around mode, overflow or underflow results in truncation, and the carry is ignored. In the saturation mode, an overflow or underflow results in the clipping of the value to the data-range limit value. The result of an operation that exceeds the range of the data-type saturates to the maximum value of the range, and results less than the range saturates to the minimum of the range. For example, the upper and lower saturation limits are FFh and 00h for unsigned bytes, and 7Fh and 80h for signed bytes.

Comparison Instructions: These instructions independently compare all the respective data elements of the two packed data types in parallel. They generate a mask of 1's and 0's depending on whether the condition is true or false. These masks can then be used by logical instructions to select elements.

Logical Instructions: These perform logical operations on quadword registers.

Shift Instructions: MMX implements two versions of logical left, right and arithmetic right shift operations. The first is the regular packed shift that independently shifts each element in the packed word (16-bit) or double word (32-bit). The second version of shift operations is logical shift or right on the whole 64-bit MMX register. These shift operations are especially important and enable re-alignment of packed data.

Conversion Instructions: These convert data-elements in packed registers; while pack instructions perform integer demotion, unpack instructions perform integer promotion. Unpack instructions also interleave elements from the source registers. Figure 3 illustrates the pack and unpack low instructions.

The entire instruction set is summarized in Table 4 of Appendix A. Instructions are typically prefixed with a 'P' (for Packed) and suffixed with a 'B' (Byte), 'W' (Word), or 'D' (Double Word). The execution of the multimedia instructions, as shown in Figure 1 exploit data parallelism on the subwords in the multimedia registers. This is referred to as *subword parallelism* or *subword semantics* in this paper.

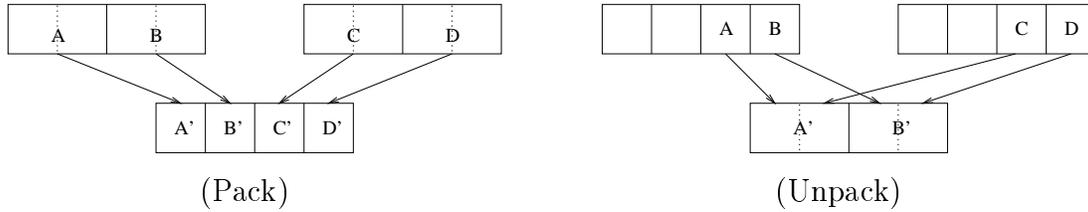


Figure 3: Pack and Unpack Instructions

3 Vectorization for Multimedia Extensions

Vectorization was one of the earliest development in the field of parallel compilation. It has traditionally been used for Vector and SIMD machines. Compilers for personal computers have never found a need for these techniques. The introduction of the subword model has however changed the situation and forced the review of vectorization techniques.

3.1 Domain Constraints and Limited Vectorization

The packed arithmetic instructions of the various multimedia extensions are basically vector operations. Therefore vectorization can be performed to exploit these instructions. These extensions have however been built on top of the scalar processors. This has resulted in constraining the domain of vectorization.

The packed instructions operate on multimedia registers which are aliased onto either the integer or the floating point registers. The size of a register is 8 bytes, which means that at most 8 data elements can be packed into a single register. The number of data elements that can be packed into a vector register is referred to as the *vector length* of the processor. We would like to perform as many instances of an instruction in parallel, as allowed by its dependence relations, for maximum speedup. It can however be seen that the former constraint on vector length would mean that at most the instances corresponding to the innermost loop can be performed in parallel. Taking this into consideration we resort to *limited vectorization*, i.e., we try to pull the instruction out of its innermost loop only.

The following section discusses our implementation of a vectorizing compiler for Intel's MMX.

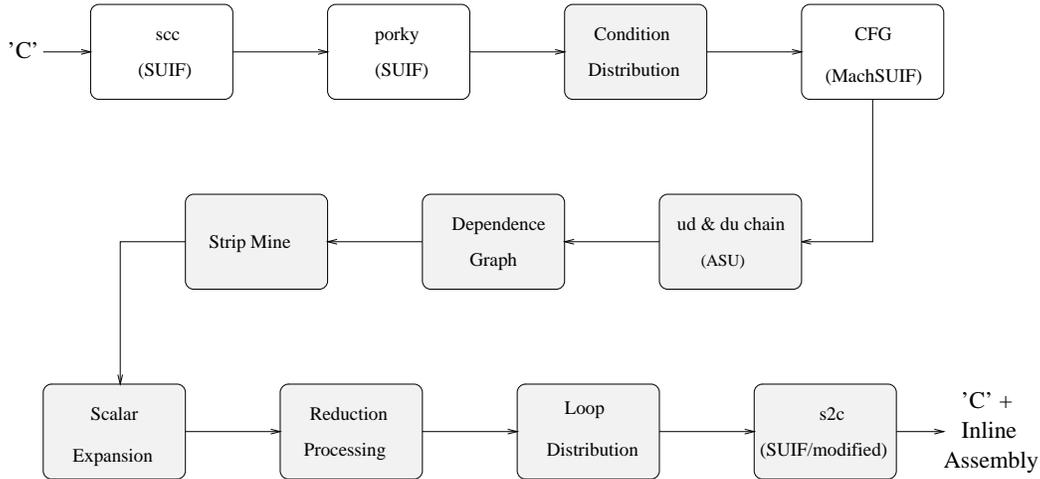


Figure 4: Process Overview

3.2 Vectorization for MMX

The compiler was implemented on the SUIF compiler framework. An overview of SUIF has been given in Section 2.2. The compiler has been structured as a set of passes. The application is converted into SUIF intermediate format and the passes are applied on the intermediate format. An overview of our implementation is given in Figure 4. In the figure, the rectangular boxes represent the different compilation phases, the shaded ones representing the passes written or modified by us.

The hypothetical example presented in Figure 5 will be used to illustrate the various passes of our compiler. Assume that all the array bounds are declared appropriately. Also, except for elements of array B , which was declared short, the other variables are declared as char data-type.

3.3 Identification of Data Parallel Sections

3.3.1 Motivation

What statements can be executed in parallel using the subword semantics? We answer this question with the help of our motivating example.

Assume $S1$ (in conjunction with $C1$) is executed using subword semantics i.e., operands of successive instances of $S1$ are packed in multimedia registers and are operated in parallel upon by multimedia instructions. When these operations are executed in parallel, it can be seen that

```

for( i = 1; i < N; i++)
  for( j = 1; j < N; j++)
    for( k = 1; k < N; k++)
      C1 : if (A[k-1]==...)
          S1 : A[k] = test + ...;
          S2 : test = ...;
      endfor
    S3 : B[i][j] = B[i-1][j] + test;
  endfor
endfor

```

Figure 5: Motivating Example

certain instances of C1 would make use of the wrong value of $A[k - 1]$. This is due to the fact that the k th instance of C1 is executed in parallel with the $(k - 1)$ th instance of S1, instead of waiting for it to complete and produce the required result, i.e., $A[k - 1]$. Basically this is a case of violation of the *true* dependence from S1 to C1. But had C1 waited for instances of S1 to complete it would have violated the *control dependence* from C1 to S1.

Clearly, S2 can not be executed in parallel using subword semantics since successive iterations write to the same location *test*, and hence when performed in parallel would result in an inconsistent state of the memory location. This is a case of *output* dependence between the successive instances of statement S2. On the other hand, S3 accesses the same memory location only in successive iterations of i . Hence instances involving successive iterations of j (and same iteration of i) can be executed in parallel. Thus the aim of this phase is to identify the statements which could be executed in parallel without violating the semantics of the the program.

3.3.2 Method

Loop distribution is the first step in our approach towards identification of data parallelism [10, 11, 12]. Loop distribution is a transformation that distributes the control of a **for** loop over groups of statements in its body. To extract maximum parallelism [22] in the presence of a dependence cycle, the loop control can be distributed over the strongly connected components of the dependence graph. A *strongly connected component (SCC)* of a dependence graph is a

maximal set of vertices in which there is a directed path between every pair of vertices in the set. A non-singleton SCC of a data dependence graph represents a maximum set of statements that are involved in a dependence cycle. Performing loop distribution over such SCCs is known as *loop fission* [10, 11, 12].

Only a singleton SCC that is not self-dependent is a candidate for exploiting subword parallelism. The presence of a self-dependence arc indicates that successive instances cannot be executed in parallel. In identifying SCCs in the dependence graph, and hence vectorizable loops, we must take into account the fact that due to the domain constraint mentioned in Section 3.1, we perform only limited vectorization which allows a statement to be pulled out of at most one level. This means that the execution order of the statement would not be affected with respect to the outer-loops. This allows us to ignore all the dependence arcs with the exception of the loop independent arcs and the ones carried by the innermost enclosing loop as for as extracting the parallelism and performing loop distribution are concerned. However, this may prevent exposing parallelism through transformation such as loop interchange [10, 11, 12]. As discussed in Section 3.7.3, the current implementation does not consider some of these loop transformations.

Statements in a conditional body are executed based on the *conditional test*. Such a statement S when pulled out of the loop, must be pulled out in conjunction with the *test* condition. Any *lexically backward* dependence between the statement S and the *test* would therefore be equivalent to a self-dependence. In order to facilitate the identification of such dependence loops, condition distribution is performed. As in the case of loop distribution, control is distributed over the statements of the *then*-part and the *else*-part. Conditional distribution is also known in the literature as IF-conversion [23]. There is one caveat however. This arises when a statement defines a variable (for example $A[k-1]$) that is used by the condition. To avoid this, any variable referenced in the test condition is replaced by a new variable initialized correctly [24]. This will be explained with the help of our motivating example in Figure 6.

Like loop distribution and condition distribution, there are a number of loop transformations such as loop splitting, loop interchange, and loop peeling that would enable and expose vectorizable loops [10, 11, 12]. In this work we have not considered these transformation and have left these as future work.

3.3.3 Implementation

Our algorithm for identifying data parallel sections involve the following steps. It is run for each function in the code.

1. The Control Flow Graph was constructed using the Mach-SUIF CFG library support [25].
2. Our module builds the use-def (UD) and def-use (DU) chains using the iterative technique described in [14].
3. The data flow module first builds the scalar component of the data dependence graph. Towards this end, arcs are added to the *use* references from the corresponding reaching definitions. Arcs are also added to *defining* references, from the set of *reached use* references of the definition it may kill. Output dependence arcs are added from *defining* references to their reaching definitions.
4. The module then builds the array data flow component of the data dependence graph. Dependence vectors are determined for each pair of references into the same array. Note that the dependence vectors are determined for each pair since array definitions are ambiguous. Based on the level of dependence, it can be determined if the dependence is either loop-independent or carried by the innermost enclosing loop, and in that case, an arc will be added between the pair of references.
5. For each outer for-loop, the module identifies the strongly connected component of the data dependence graph.
6. Single statement strongly connected components, which are not self-dependent are annotated as data parallel. They can therefore be executed using the subword semantics, provided the result type is sufficiently short.

3.3.4 Illustration

Conditional distribution is first performed on the code, transforming it as shown in Figure 6. Figure 7(a) shows the data dependence graph as constructed for the hypothetical example (in Figure 5) after conditional distribution. As mentioned earlier, we only show loop independent dependence arcs and loop dependent arcs carried at the innermost enclosing loop. For example, for S3, we are only concerned about the dependences carried at level 2 (there are none in this example) and are not concerned about the dependences carried by the outer loop (level 1). Figure 7(b) shows the *strongly connected component* graph where statements X1, S1, and S2 are contained in a strongly connected component. Since there are no singleton SCCs, statements in the innermost loop (level 3) as such are not vectorizable. Considering level 2 and the appropriate dependence graph for it, one can find out that statement S3 forms a singleton SCC (without a self-arc) at level 2. Hence S3 can be executed using subword semantics.

```

for( i = 1; i < N; i++)
  for( j = 1; j < N; j++)
    for( k = 1; k < N; k++)
      X1 : C1_temp = A[k-1];
      S1 : if (C1_temp==...) A[k] = test +...;
      S2 : test = ...;
    endfor
  S3 : B[i][j] = B[i-1][j] + test;
endfor
endfor

```

Figure 6: Condition Distribution

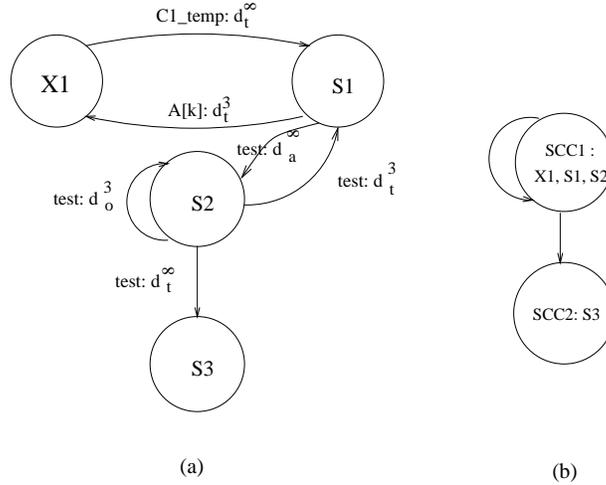


Figure 7: Dependence Graph and Strongly Connected Components

3.4 Scalar Expansion and Strip-Mining

3.4.1 Motivation

Consider the dependence graph shown in Figure 7(a). It can be seen that $S2$ can also be executed using subword semantics but for the output dependence arc on itself. This arc wouldn't have been there if $test$ had been an array reference indexed through k . Then successive iterations would write to different memory locations. This is the idea behind scalar expansion transformation [10, 11, 12].

3.4.2 Method

Definitions of a *scalar* variable in a loop results in an *output* dependence arc to itself. This arc can be broken by replacing the scalar variable by an array, essentially providing each iteration with a memory location to write on. This is known as *scalar expansion*. The scalar expansion of variable x is possible if it satisfies the following constraints:

- x is the target of at least one definition: If this is not the case, there is no gain as no self-arc is broken.
- x is not a target of a single statement recurrence: A single statement recurrence involves an anti-dependence and a true dependence arc in addition to the output-dependence arc. Again, there is no gain from expanding the scalar as true dependence would prevent vectorization.
- x is not an induction variable: In this case the iteration index must be expanded for correct results. This operation could be quite costly in our case.

As we discussed earlier, in MMX only a small number — at most 8 — operations can be executed in parallel. Hence not all the iterations of the vectorizable loop will be executed in parallel. These loops must therefore be partitioned into smaller sections which can be handled in parallel. This is known as *strip-mining* [10, 11, 12]. The number of iterations that can be handled in parallel is equal to the *vector length* of the processor. Strip-mining results in a nested loop, the outer loop with the same bounds as the original loop, and *vector length* stride, and the inner loop performing the iterations corresponding to the stride. Usually there is also a remainder section, similar to the inner loop, completing the final strip. We partition the loops into strips of length 8. When the vector length is 8, the inner loop, subject to satisfying dependence constraints, can be replaced by a single vector instruction which corresponds to 8 parallel operations. For vector lengths of size 4 and 2, our code generator handles them appropriately as discussed in Section 3.7.3.

Taking another look at scalar expansion in the context of strip-mining, it can be seen that instead of providing each iteration with a memory location it would be enough to provide just *vector length* memory locations. More specifically, a variable x would be scalar expanded as *expanded_x[vector length]*. This is because in our approach scalar expansion is performed after strip-mining on the inner loop. This reduces the memory overhead and at the same time allows for expanding the variables even when the loop bounds are not known. This can also be considered as *privatization* [10, 11, 12, 26] with respect to outer loop of strip-mining.

3.4.3 Implementation

For each variable x to be expanded, the module for scalar expansion would perform:

1. Declare an array whose elements are of the same type as x with size equal to N , the loop bound. After strip-mining this size would reduce to the *vector length* + 1. Note that the vector length, after strip-mining is always 8 in our implementation.
2. Replace all the references to x within the loop by an appropriate reference to the array. Suppose the loop index is i , and the lower bound of the loop is lb_i
 - uses before the definition of x are replaced by $expanded_x[i - lb_i]$.
 - the definition and subsequent uses of x are replaced by $expanded_x[i - lb_i + 1]$.
3. Adds an initialization instruction of the form

$$expanded_x[0] = x;$$

before the loop.

4. Adds a finalization instruction of the form

$$x = expanded_x[vector\ length];$$

after the loop.

Some explanation is needed for including Steps 3 and 4. A loop, say $L1$ may use x before *defining* it. To avoid inconsistencies, the value of x must be stored in an appropriate array location before the nested loop. The loop also defines x , which must be restored from $expanded_x$ after the loop. The module treats these operations as a form of *initialization* and *finalization* respectively.

Consider a loop, say $L2$, nested in the loop $L1$. $L2$ may refer to x . This would lead to an inconsistency. To handle this, we use an approach similar to finalization, i.e., restore x from the appropriate array location. Similarly $L2$ may define x , which is handled by storing x in the same array location from which x was restored. The array location can be identified by treating the case as an use of x and applying Step 2 of the algorithm.

3.4.4 Illustration

Application of the strip-mining of the j and k loops and scalar expansion transformation to our earlier example results in the following code. Here VL denotes Vector Length. The remainder loop has not been shown in this example for simplicity. Lastly, due to the fact that S3 is in a singleton SCC at the j -loop level, and in order to exploit subword parallelism on S3, the j loop is strip-mined.

```
for( i = 1; i < N; i++)
  for( Stride_j = 1; Stride_j < N; Stride_j += VL)
    for( j = Stride_j; j < Stride_j + VL; j++)
      for( Stride_k = 1; Stride_k < N; Stride_k += VL)
        X2 : Exp_C1_temp[0]=C1_temp;
        X3 : Exp_test[0]=test;
        for(k = Stride_k; k < Stride_k + VL; k++)
          X1 : Exp_C1_temp[k-Stride_k+1] = A[k-1];
          S1 : if (Exp_C1_temp[k-Stride_k+1]==...)
              A[k] = Exp_test[k-Stride_k] +...;
          S2 : Exp_test[k-Stride_k+1] = ...;
        endfor
        X4 : test=Exp_test[VL];
        X5 : C1_temp=Exp_C1_temp[VL];
      endfor;
    S3 : B[i][j] = B[i-1][j] + test;
  endfor
endfor
```

The data dependence graph for this code has been shown in Figure 8 for levels 4 and 5. Note that the dependence levels for the arcs have changed due to strip-mining. The broken arc on S2 and the broken arc from S1 to S2 show respectively the output and anti dependences broken by scalar expansion. Because of this, S2 is no longer in the SCC consisting of X1 and S1, and it (S2) forms a single SCC with no self-arc. Hence the instances of S2 can now be executed in parallel using the subword semantics.

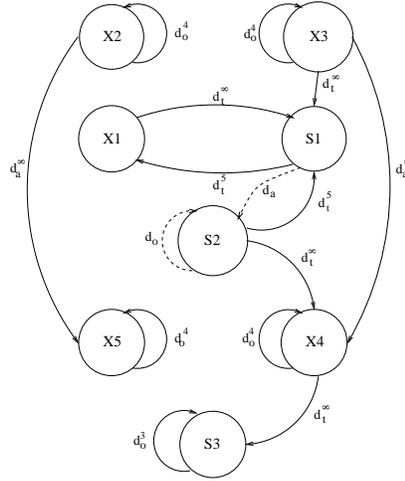


Figure 8: Dependence Graph after Scalar Expansion

3.5 Reduction Processing

3.5.1 Motivation

To further expand the scope of statements/loops that can be vectorized, *grouping* and *reduction* techniques are applied [10, 11]. We motivate the need for these techniques using vector dot product calculation as an example.

```

for( i = 0; i < N; i++)
  R1: dotproduct = dotproduct + A[i] * B[i];
endfor

```

The statement *R1* is an example of a single statement recurrence, with true and output dependence arc to itself. The data dependence graph for this example is shown in Figure 9(a). Scalar expansion removes the arc associated with the output dependence but leaves the true dependence arc and hence the self-loop. This would not help in increasing the scope of parallelism in this case.



Figure 9: Dependence Graph (a) before and (b) after Reduction Processing

Since addition is associative, the order of summation does not affect the result, except in case of overflow. So the data can be partitioned and partial sums can be determined. These partial sums can be later accumulated into the result. Strip-mining, which partitions the iterations of the loop can be used to partition the data. Each iteration of the inner loop computes different partial sums, such as

$$dotproduct[i] = dotproduct[i] + \dots$$

How does this help? Here, with strip-mining, the recurrence is not on the innermost loop but on the outer loop. This would have the effect of pulling the true dependence arc out of the inner loop, allowing us to ignore it as for vectorization of the inner loop is concerned. The code in Section 3.5.4 illustrates this transformation.

3.5.2 Method

While scalar expansion removes output dependence self-arcs associated with scalar variables, reduction processing aims to remove true dependence self-arcs. Addition and multiplication operations are associative. This allows for reordering and grouping of these operations and subsequently reducing to get the result.

A variable x can be expanded for reduction of the operation if

- x is the source and the target of a definition and the operation is a *reduction operation* [11], such as add, multiply, and maximum
- x is not the source of any other instruction in the loop: In such a case the expanded x would have to be collapsed to avoid inconsistency, and this operation is costly.

3.5.3 Implementation

For each variable x to be expanded, our module for reduction processing would

1. Declare an array whose elements are of the same type as x .
2. Replace the references to x within the loop, whose index is i and lower bound is lb_i , by $R_x[i - lb_i]$.
3. Adds an initialization of the form
for $0 < i < vector\ length$

$$R_x[i] = 0;$$

before the enclosing loop (in case of strip-mined loop, the initialization can be pushed one more loop outwards) after the *reaching definitions* of x , of course with the exception of the current definition.

4. Adds a finalization instruction of the form

for $0 < i < \text{vector length}$

$$x+ = R_x[i];$$

after the loop (in case of strip-mined loop, the finalization can be pushed one more loop outwards).

5. The initialization and finalization can be pulled out of the enclosing for-loops which do not contain either a reaching definition or a reached use of x .

3.5.4 Illustration

The vector dot-product computation is now transformed as:

```

for( Stride_i = 0; Stride_i < N; Stride_i += VL)
  for (i = Stride_i, Partial_i=0; i < Stride_i + VL; i++, Partial_i++)
    R1: R_dotproduct[Partial_i] += A[i] * B[i]; /* subword execution */
  endfor
endfor
for( Partial_i = 0; Partial_i < VL; Partial_i++)
  R2: dotproduct = dotproduct + R_dotproduct[Partial_i];
endfor

```

The data dependence graph for this example after the above transformation is shown in Figure 9(b). Though the level of true dependence is still 1, it must be noted that the strip-mining has introduced an inner-loop which can now be executed in parallel. It can be seen therefore that $R1$ can be executed in parallel, as now both the self-arcs have been removed. Statement $R2$ represents the accumulation of the partial sums into *dotproduct*.

The reduction transformation may produce a result that is different from the one produced by sequential execution in case of an *overflow*. Therefore it would be better to enable this transform through an user option. This option would indicate that overflow is an *exceptional* condition.

3.6 Loop Distribution

3.6.1 Method

After the code transformations, the loop control can be distributed. As mentioned earlier, to enable loop distribution in the presence of back arcs, the strongly connected components in the body of the loop are identified and topologically sorted. This would result in a graph in which all the arcs are lexically forward. The loop control can then be distributed over the strongly connected components. As mentioned earlier, the single statement strongly connected components, which are not self-dependent can be annotated as data parallel instructions.

3.6.2 Implementation

For each outer for-loop:

1. The strongly connected components are identified from the data dependence graph.
2. The strongly connected components are ordered using topological sort.
3. Statements are now reordered so that
 - statements belonging to the same strongly connected components are grouped together in the program order.
 - strongly connected components are in the topologically sorted order.
4. Loop control is now distributed over each strongly connected component.
5. Single statement strongly connected components which are not self-dependent and whose result type is conducive to subword execution are annotated as data parallel statements.

3.6.3 Illustration

Applying loop distribution to the hypothetical example results in the following code:

```

for( i = 1; i < N; i++)
  for( Stride_j = 1; Stride_j < N; Stride_j += VL)
    for(j = Stride_j; j < Stride_j + VL; j++)
      for( Stride_k = 1; Stride_k < N; Stride_k += VL)
        X3 : Exp_test[0]=test;
        for(k = Stride_k; k < Stride_k + VL; k++)
          S2 : Exp_test[k-Stride_k+1] = ...;
        for(k = Stride_k; k < Stride_k + VL; k++)
          X1 : Exp_C1_temp[k-Stride_k+1] = A[k-1];
          S1 : if (Exp_C1_temp[k-Stride_k+1]==...)
              A[k] = Exp_test[k-Stride_k] +...;
        endfor
        X4 : test=Exp_test[VL];
      endfor;
    endfor
  for(j = Stride_j; j < Stride_j + VL; j++)
    S3 : B[i][j] = B[i-1][j] + test;
  endfor
endfor

```

Figure 10 shows the dependence graph, where the strongly connected components are grouped together as a single node, after the above transformations. It can be seen from the graph that $S2$ and $S3$ can be executed using the subword semantics. The statements $X2$ and $X5$ have been omitted from the above code as they play no significant part.

3.7 Code Generation

The extensive use of the C programming language for system applications has made *performance* a necessity rather than a luxury. Commercial, as well as open-source compilers, therefore perform a wide variety of machine-dependent and machine-independent optimizations. Therefore it makes sense to use such a compiler to leverage the scalar optimizations performed by it, while the vectorizable sections are handled by our modules.

Hence in this paper we propose to generate the inline assembly code only for the vectorizable

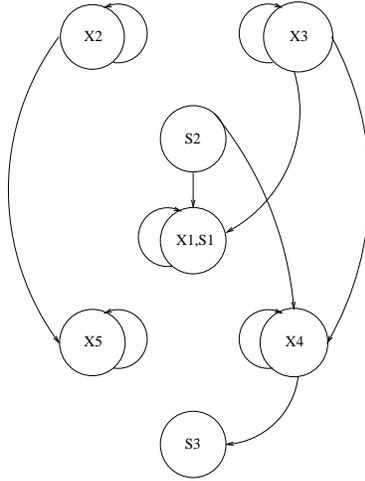


Figure 10: Strongly Connected Components Graph

sections of the code. Inline assembly allows the programmer to introduce *assembly* instructions in a *C*-code. The code generator takes the SUIF abstract syntax tree file as input and emits the *inline assembly* equivalent of the data parallel code fragments. Details on inline assembly can be found in [27]. The SUIF-C converter, *s2c*, has been modified for this purpose.

Let us now take a brief look at the issues involved in code generation.

Register Management: Basically this involves the management of two sets of registers, 8 general-purpose and the 8 multimedia registers. The native compiler/assembler provides support for management of the general-purpose registers. This also allows it to perform optimizations during register allocation.

The multimedia registers have to be handled by our code generator. A simple stack based register assignment scheme has been used for this purpose, i.e., the register-set is treated as a stack. On request, the top of stack is returned, and on release the register is pushed back onto the stack.

Subword Size: As discussed earlier, MMX allows subword sizes of 1, 2 and 4 bytes. The subword size for the execution is chosen to be the result (of the operation) size. Once the subword size is chosen, the operands would have to be promoted or demoted based on their respective sizes.

For integral promotion, the *punpck* (unpack) instruction is used. There is also an added issue of the *signedness* of the operand. If signed, it would be necessary to perform signed extension of the operand. For integral demotion, the *pack* operation is used. Let us now

look at the integral promotion (say byte to word) operation on a register mm , which currently holds 4 *byte-subwords*.

1. Create a zeroed register say $mm1$, usually performed by subtracting the register from itself.
2. Unpack the lower bytes of $mm0$ with $mm1$. This would result in the register $mm0$ holding the byte subwords of $mm0$ left-shifted to the word-boundary (refer to Figure 11(b)).
3. If the operand using $mm0$ is signed, perform arithmetic right shift (sign-extended) on $mm0$, otherwise perform logical right shift on $mm0$ (refer to Figure 11(c)).
4. Copy $m0$ into mm .

This sequence has been illustrated in Figure 11.

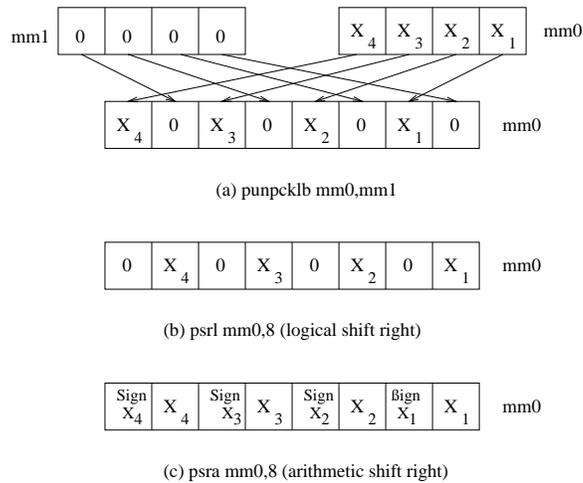


Figure 11: Integral Promotion for Subwords

Constants: An operation involving constants cannot be performed using subword semantics. It would therefore be necessary to replicate the value on all the subwords. That is, similar to scalar expansion, we also expand constants. This is performed using the *punpck* (unpack) instruction. Basically the register is unpacked with itself the required number of times.

Alignment: Memory alignment deals with the access of multi-byte operands that cross word boundary. Basically, the data-bus would be able to fetch word-length bytes of data in one access, provided the bytes are word-aligned. In most processors, movement from and

to memory must be word-aligned addresses. Specifically, a quad-word is expected to be aligned on a 8-byte boundary. It may result in an error otherwise. Hence the vectorizing compiler must ensure that in performing packed memory operations the addresses must be aligned; if they are not, additional work is involved in loading word-aligned data and shifting them to get the required non-aligned data. However, Intel's Pentium processors, on the other hand, support access to non-word aligned multi-bytes. In this case, the processor performs more than 1 memory access, and hence the operation may be slower. The compiler therefore does not have to deal with alignment issues for *correctness*. The performance would however deteriorate as shown in Section 4.3.

The issue of alignment has not been dealt with in our work.

3.7.1 Implementation

The code generation phase is based expression-tree traversal and involves the following steps.

1. Perform post-order traversal of the expression tree.
2. For each node do
 - if it is a variable symbol, emit instruction to load the variable in to general-purpose register. Pop a register out of the stack. Move the contents of the register to the multimedia register. The multimedia register is the destination register.
 - if it is an array reference instruction, emit instruction to load the address of reference into a general-purpose register. Pop a register out of the multimedia stack.
 - if it is not an array instruction, emit code corresponding to the instruction, and register value returned by the children nodes. The register corresponding to the left child is the destination register.
 - if not left child of its parent, push destination register back onto stack.
 - return destination register.

3.7.2 Illustration

Applying the code generation pass for a small fragment, the statement $S3$ of the example in Figure 5, where the elements of the array B are *short* and $test$ is a char data-type, results in the code shown in Figure 12.

```

asm(
    "movq (%1) , %%mm6"           // Load qword from 0(GP1) to MM6
    "movd %2 , %%mm5"           // Move doubleword from GP2 to MM5
    "psubb %%mm0 , %%mm0"       // Zero out MM0
    "punpcklbw %%mm5 , %%mm0"   // Unpack lower bytes of MM0 and MM5
                                // into MM0
    "movq %%mm0 , %%mm5"       // Copy MM0 into MM5
    "psraw $8 , %%mm5"         // Arithmetic Right Shift MM5 by 8
                                // Integral Promotion of test
                                // completed
    "punpcklwd %%mm5 , %%mm5"   // Replicating test onto all
    "punpckldq %%mm5 , %%mm5"   // subwords
    "paddw %%mm5 , %%mm6"       // Add MM5 into MM6
    "movq %%mm6 , (%3)"         // Store quadword at 0(GP3)
    :                             // Clobber List
    : "r" (0) , "r" (&B[i - 1][j]),
      "r" (test) , "r" (&B[i][j])

                                // GP register loading instruction,
                                // 0 is loaded in GP0 or %%0, address of
                                // B[i-1][j] in GP1 or %%1 and so on.
);

```

Figure 12: The Inline Assembly Code

The subword size for the above execution is 16 bits. This example illustrates *integral promotion* of *test* and the replication of the scalar value onto all the subwords. It can also be seen that the general-purpose register assignment is left to the compiler. This code performs 4 successive instances of S2.

3.7.3 Remarks

A couple of remarks on our implementation are in order. First, our method does not directly handle *conditionals*. Instead a transformation is performed. Conditionals after condition distribution transformation are of the form:

```
if (test)
  x = expr;
```

This can be further transformed as:

```
x = ((not) test and x) or (test and expr);
```

Now the original method can be applied as is. The operations *and* and *and not* are included in the extensions. Further, *test* is evaluated once and stored in a register for further operations.

Second, for reasons of simplicity, our implementation always strip-mines a loop with a vector length of 8. When the subword size does not match this vector length, for example for word size operands, instead of re-folding the loop to match the vector length 4, we duplicate the vector operation, subject to satisfying all dependence constraints, and retain the original strip-mining. In other words, the strip-mined inner loop with a vector length 8 accommodates 2 vector operation each of vector length 4. This is handled appropriately by our code generation algorithm, by re-emitting the instructions corresponding to the expression tree multiple times.

3.8 Implementation on the SUIF Framework

The techniques for identification of data parallel sections and the different code transformations were implemented as the *passes* of the SUIF framework. As part of constructing the dependence graphs, we augmented the *CFG* library [28] to compute the UD and DU-Chain using the iterative algorithm. We developed the array dependence analysis module which uses the SUIF dependence library. Lastly, we modified the SUIF-C converter, *s2c*, to generate the (inlined assembly) code. The different passes and the order in which were applied are shown in Figure 4. We retained SUIF’s interface between different compiler passes. That is, these compilation passes are implemented as separate programs that communicate via SUIF files. Different passes can communicate by annotating the program representation.

The code transformation passes involved extensive reordering and cloning of SUIF structures. The SUIF hierarchy facilitated reordering the structures considerably. Reordering would typically require that the node be removed from its parent and inserted before or after another node as per the requirement. The internal mechanism for such a change would be handled by SUIF. Cloning of a structure was also handled similarly. Cloning a node which would be inserted in a different scope would require that the scopes be reconciled. This involves identifying the symbols referenced by the node which are not present in the other scope, and cloning these symbols also. The low-level implementation of these operations is again handled by SUIF.

Our vectorizing compiler with SUIF infrastructure currently runs on Intel's Pentium II processor-based workstation running Redhat Linux.

A few limitations of our implementation are:

- The compiler considers only *for*-loops as candidates for vectorization. *Do-while* and *while-do* loops are ignored.
- Loop transformations such as loop splitting, loop interchange etc. are not performed. These transformations can enhance the vectorizability of the loop.
- The compiler generates some overhead in the form of unnecessary strip-mining and scalar expansions. This overhead will be incurred in the absence of a subsequent pass which can reverse the effect of unnecessary strip-mining and scalar expansion. It is known that such reversing would be difficult to implement.
- During code generation the alignment issues have been ignored. This will definitely have an impact on the performance.
- Function calls and the presence of pointers hamper vectorization. Inlining the function is a possible solution to overcome the former limitation. However, the compiler does not currently support such function inlining.
- The language itself poses some barriers to exploiting the extensions fully. For example, though saturating arithmetic is supported by the vendor extensions, there is no natural way for the programmer to specify the need for such an operation. Also some other instructions such as multiply and add are not supported by this implementation.

4 Results and Discussion

In this section we present the initial performance results obtained by using our vectorizing compiler. In our experiments we compile kernels from media processing applications and run them on Intel MMX architecture and measure their performance. The reasons for considering kernels rather than the multimedia application itself, are as follows. Media processing applications typically spend a major fraction of the execution time on a few small data parallel kernels. Studying the performance improvement in the kernel gives us a direct measure of how much of the subword parallelism is being exploited by vectorizing compiler. The complete applications typically contain sequential and non-vectorizable code as well as code which operate on full

words which do not contribute to any improvement in performance in the MMX architecture, whether they are hand-tuned, or exposed by an optimizing vectorizing compiler.

4.1 Benchmarks

We consider the following kernels. Dissolve is a video processing application [29]. It is typically used during the editing of a video sequence. The application takes two video frames as input, computes a weighted average of the pixels in the frame and outputs a new frame. The new frame can then be used as a filler between the input frames. For our experiments, we have considered 128×128 pixel frames and compute just the average of the pixels. As can be seen, dissolve is an extremely data parallel application.

Chroma-keying is an image processing utility. It basically replaces the background of the given image with an alternate background (also an input). In our experiments, we have considered a 128×128 image for Chroma-keying. It replaces the pixels having the background color in the first frame, by the corresponding pixel of the second frame.

Vector dot product is an algebraic computation common in signal processing applications. The inputs are two 1-dimensional vectors and the output is the sum of the product of individual elements of the vector. For our purpose, we compute the dot product for 64 element vectors. Both dot product and the following sum of absolute differences kernel require grouping and reduction transformations to be performed to exploit subword parallelism.

Sum of absolute difference (SAD) is the processing kernel for motion-estimation algorithm. Motion estimation is used in MPEG-encoder to compress the input stream using the implicit temporal coherence between successive frames of a video. This (SAD) kernel is a key target for performance improvement through MMX. As a matter of fact, VIS includes an instruction to perform this operation. We have considered the sum of absolute difference of a 16×16 block. This SAD kernel is then integrated into the motion estimation code and its performance is evaluated².

4.2 Experimental Setup

The kernels listed above are not computationally intensive, but are usually invoked several times. As mentioned earlier, this usually results in the application spending a significant fraction

²To be fair, however, it should be mentioned that the MMX version of the code does in fact benefit to some extent due to the reduction of loop control overheads.

of the execution time on these kernels. This means that optimizing these kernels can translate into a significant improvement for the applications.

We compare the performance of the kernel compiled by the native compiler (*gcc*) and that of the kernel compiled on our vectorizing compiler. The former is referred to as non-MMX code as the *gcc* compiler does not generate MMX extension instructions, while the latter is referred to as the vectorized code or MMX code. To generate the MMX code, our compiler passes are applied to the kernel to get the source code augmented with the inline assembly instructions. This code is then compiled with the native compiler, *gcc*, and linked to the main routine. Similarly, for obtaining the non-MMX code, the kernel is compiled on the native compiler and linked with the main routine. In generating the MMX and the non-MMX code using the native *gcc* compiler on Pentium II, we obtained both the optimized (`-O2`) and unoptimized versions. Although we report the performance of both optimized and unoptimized versions of MMX and non-MMX code, in our discussion we concentrate mainly on the unoptimized versions for the following reasons: (i) The optimizations of the *gcc* compiler has only minor impact on the MMX code, since the inlined assembly code is left untouched by the compiler; and (ii) the register allocation in the optimized MMX code with the inlined assembly turned out to be poorer than in the non-MMX code. A full fledged code generation approach, rather than our inline assembly code generation, would have eliminated some of these differences. It can be seen from the results that the performance of the non-MMX code improves much more significantly, when the optimizations were turned on, than compared to the MMX code.

The main routine times (using the `times()` routine) the execution of these kernels on an Intel II processor running at 266 MHz. To make the accurate measurement of execution time, it was required to call the kernel functions several times (1 to 50 Million times), and the average time for executing the kernel once was computed. The kernels dissolve and chroma-key were run respectively 1M times, while dot-product and SAD were run for 50M and 10M times respectively. Speedup is then measured as

$$\text{Speedup} = \frac{\text{Exec. Time for non-MMX Code}}{\text{Exec. Time for Vectorized Code}}$$

4.3 Results and Discussion

Table 1 presents the speedup obtained by our compiler over a traditionally compiled (non-MMX) code. For each kernel, we report the major data type used by the kernel (byte (i8) or short word (i16)), the execution time (in microseconds) for the unoptimized and optimized versions of MMX and non-MMX code, the Speedup achieved, and the expected speedup in the

architecture for the data type used by leveraging just subword parallelism. Although it may be possible to improve upon the expected speedup through a more optimal use of MMX features such as saturation logic, and instructions such as multiply-add, etc., we do not consider them here.

Kernel	Data Type	Unoptimized Code			Optimized Code			Theoretical Speedup
		Exec. Time (in μ S)		Speedup	Exec. Time (in μ S)		Speedup	
		MMX code	non-MMX code		MMX code	non-MMX code		
Video Dissolve	i8	225.680	1460.300	6.47	156.520	305.300	1.95	8
Chroma-Keying	i8	904.520	4751.120	5.25	512.740	1020.540	1.99	8
Byte Dot Product	i8	1.020	2.591	2.54	0.901	0.866	0.96	8
Short Dot Product	i16	2.187	2.706	1.24	1.511	1.046	0.69	4
Sum of Absolute Difference (SAD)	i8, i16	8.856	47.030	5.31	7.168	13.943	1.95	4/8

Table 1: Speedup for the Kernels

It can be observed that the speedups achieved on the unoptimized code is significantly higher than that achieved on the optimized code. As discussed earlier, this may be due to the poor code and register allocation in the presence of inlined assembly code. This is also partly due to the overheads generated by our aggressive strip-mining and scalar expansion. It is interesting to observe that for the dot product kernel, the optimized non-MMX version in fact performs better than the optimized MMX version. However, it can be noticed that, except in the case of dot product, even in the case of optimized version the vectorized MMX code achieved roughly half the speedup achieved on the unoptimized code. Let us now turn our attention to the performance of unoptimized MMX and non-MMX code.

It can be seen that our compiler is able to fully harness the data parallelism in the case of the *dissolve* and *chroma-keying* and the *SAD* kernels. We present the speedup obtained by integrating the SAD kernel into the motion-estimation module and compare it against a hand-tuned version in Table 2. The kernel was hand-tuned using:

$$SAD(\mathbf{A}, \mathbf{B}) = (\mathbf{A} \text{ Saturated Subtract } \mathbf{B}) \text{ bitwise-or } (\mathbf{B} \text{ Saturated Subtract } \mathbf{A})$$

which does not involve any conditionals and is the common way of computing SAD using the MMX instructions. We refer to this version as hand-tuned SAD (H-SAD). We compare the speedup achieved by the hand-tuned version (measured as the ratio of the execution time of the original code compiled by *gcc* to the execution time of the hand-tuned version compiled by *gcc*) with that obtained by our vectorized code. It can be seen that the speedup obtained by our compiler is comparable to that obtained by the hand-tuned version (within 85%).

Kernel	Data Types	Exec. Time (in mS)		Speedup
		MMX (unopt.)	NON-MMX (unopt.)	
Sum of Absolute Difference(SAD)	i8, i16	0.0088	0.0470	5.31
Motion Estimation(using SAD)		2630.0	11436.0	4.34
Sum of Absolute Difference(H-SAD) (alternate <i>hand-code</i> approach)	i8, i16	0.0079	0.0470	5.94
Motion Estimation(using H-SAD)		2267.0	11436.0	5.04

Table 2: Speedup for Motion Estimation

What surprised us was the performance *dot product* kernel. The MMX instruction set supports packed *short-word multiply* only. This meant that the byte length vector elements had to be promoted to short before the multiply operation. Initially the result was attributed to the overhead involved in this expansion. This led us to try the dot product for short-word quantities. As can be seen, the short dot product performs better, but not significantly so.

An inspection of the dot product code revealed that the array introduced by our grouping and reduction module was not quad-word aligned. As discussed in Section 3.7, alignment does not cause any problem (in terms of correct execution of the code) although it may incur performance penalties in Intel’s processors. We enforced the array to be aligned and measured the performance of kernel. These results are reported in Table 3. Further, we suspect that the overhead due to the *sequential code* is significant in the dot product kernel. To confirm this, we measured the performance of dot product for different lengths of input vectors.

As can be seen from the table, for small vector lengths the sequential section dominates, causing a loss of speedup. As the vector length increases the speedup also increases as the fraction of execution time spent in the sequential section comes down. The significance of alignment is reflected in the speedup gained over the unaligned version.

Input Size	Exec. Time (in μ S)			Speedup	
	Aligned MMX	Un-Aligned MMX	Without MMX	Aligned MMX	Un-Aligned MMX
16	0.658	0.802	0.802	1.21	1.00
32	0.911	1.269	1.481	1.62	1.16
64	1.481	2.248	2.776	1.87	1.23
128	2.459	3.891	5.457	2.21	1.40
256	4.336	7.271	10.802	2.50	1.48

Table 3: Effect of Input Size and Alignment on the Performance of Dot Product

5 Related Work

Vectorizing techniques have been discussed in detail in literature. This includes a comprehensive coverage by Zima and Chapman in [10] and by Wolfe in [11].

SUIF vectorizing compiler [30] is an implementation of vectorization techniques on the SUIF platform. The target architecture is UCB’s Torrent architecture [31], which is a traditional vector architecture. An optimizer for VIS extension was proposed in [32]. The optimizer makes use of the SUIF vectorizing compiler as its backbone. Code generation had been completed only for parallel add and parallel conditional copy. The focus of this work was on alignment issues during code generation.

Operation packing was proposed in [33]. The idea is to support the packing operation in the hardware, for the instructions in the reservation station. Such a technique may provide an improvement for almost all applications, but the speedup may not measure up to static compiler analysis for vectorizable applications. In [34], multimedia extensions are exploited in a Java JIT compiler. A vectorizing compiler for *VIS* has been proposed in [35]. In addition to traditional vectorization techniques, loop unrolling and instruction scheduling has been employed to exploit subword parallelism. Further, their work also deals with memory alignment, which is essential in *VIS* for the sake of correctness, while it is only of a performance issue in the case of Intel processors.

Lastly we compare our approach of providing a vectorizing compiler with the support [1] provided by vendors in the form of enhanced system libraries and macro calls. If we consider

the Sum of Absolute Differences example, our vectorizing compiler is able to perform necessary code transformation, such as scalar expansion and reduction. In case of hardware supported enhanced libraries, the programmer can make use of a system version of *absolute()* function, say *Vabs()*. The function *Vabs()* would be performed using the subword semantics. While the function may exploit subword parallelism in computing the absolute difference of N elements, the sum is calculated sequentially, on each call, thereby losing out on some parallelism. Further, these system enhanced functions cannot be in-lined since the source code would not be available. On the other hand, using macro calls requires the user to be aware of the code segment which can be optimized by the multimedia extensions and the macros provided. Further, the code transformations have to be performed manually. Lastly, programming with the macro calls is as hard as programming with assembly.

6 Conclusion

This paper presents our implementation of a vectorizing compiler for Intel's Multimedia Extension [5]. This extension is targeted at the data parallel kernels of media processing applications [1]. The goal of this work was to provide compiler support for this enhancement to the instruction-set, and make it transparent to the programmer. Vectorization techniques [10, 11], which have traditionally been used by compilers for vector and SIMD processors, are used by our compiler to extract subword parallelism from a sequential code. To enhance the scope for application of subword semantics, our compiler performs several code transformations. These include strip-mining, scalar expansion, grouping and reduction, and loop distribution. We made use of SUIF, a compiler tool, for implementing the various passes of our compiler [21]. Our compiler generates inline assembly code for the data parallel sections identified by it. It must be noted that our compiler also performs conditional distribution during vectorization and code generation. Further, though the target architecture is Intel, the techniques discussed could be applied for the multimedia extensions of the other architectures as well.

We also report initial performance results of the code generated by our vectorizing compiler. Four data parallel kernels from multimedia application when compiled by our compiler and run on an Intel Pentium II processor result in a significant speedup compared to the performance of the code compiled by the native compiler. Further the performance improvement achieved by our compiler is within 85% of the performance of the hand-tuned code in one of the applications, viz., sum of absolute differences. We have also reported the performance achieved by a multimedia application (motion estimation) by linking the compiled code for a kernel used in

that application.

6.1 Future Work

The performance of our compiler when applied to the various multimedia kernels has been encouraging. The scope for vectorization can be enhanced through other loop transformations such as loop interchange, loop splitting, and loop-peeling [10, 11]. Our compiler currently does not perform any instruction scheduling [36]. Instruction scheduling and software pipelining [36, 37, 38, 39] — an instruction scheduling technique for iterative computation — would definitely lead to improved performance due to efficient usage of available resources. Also our compiler does not perform global register allocation [36]. Such an allocation may not be trivial as the floating point and multimedia registers are overlapped. Further, extending our compiler to exploit the features of such as multiply-and-add instructions would further increase the instruction-level parallelism exploited.

In the presence of pointers, the conservative assumptions made on aliasing limit vectorization. Alias analysis can improve the situation, but the maximum benefit in terms of performance can be obtained only by better coding practices. The proposed introduction of the *restrict* keyword as a part of the C-9x standard would allow the programmer to provide some guarantees regarding the memory locations referenced by a pointer. This provides the compiler with more freedom for optimizing in the presence of *restricted* pointers.

Function inlining would enhance the scope of vectorization considerably. This would enable our compiler to perform more optimizations and reduce the function call overhead. Considering that the data parallel kernels are called very frequently, this could lead to a significant improvement. Hence function inlining, which is currently not supported in our compiler, would be an useful future extension.

Acknowledgments

The authors are thankful to the anonymous reviewers for their constructive comments which helped improve the quality of presentation and organization of this paper. The authors acknowledge V. Santhosh Kumar for his help in running the experiments.

References

- [1] R. B. Lee and M. D. Smith. Media processing: A new design target. *IEEE Micro*, pages 6–10, July/Aug 1996.
- [2] T. M. Conte, P. K. Dubey, M. D. Jennings, R. B. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, and A. Wolfe. Challenges to combining general-purpose and multimedia processors. *IEEE Micro*, pages 33–37, December 1997.
- [3] R. B. Lee. Subword parallelism. *IEEE Micro*, July/Aug 1997.
- [4] Intel. *Intel Programmers User Manual*, 1996.
- [5] U. Weiser and A. Peleg. MMX technology extension to Intel architecture. *IEEE Micro*, pages 42–50, July/Aug 1996.
- [6] M. Tremblay. VIS speeds new media processing. *IEEE Micro*, pages 10–20, July/Aug 1996.
- [7] R. B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, pages 51–59, July/Aug 1996.
- [8] P. K. Dubey. Architectural and design implication of media processing, 1998. HIPC'98 Tutorial Lecture.
- [9] K. Kennedy and R. Allen. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and System*, 9(4):491-554, Oct. 1987.
- [10] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley Publishing Company, Reading, MA, 1991.
- [11] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, Reading, MA, 1996.
- [12] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformation for high-performance computing. *ACM Computing Surveys*, 26(4):345-420, Dec. 1995.
- [13] Suif Compiler Group *SUIF Manual*. Stanford University Compiler Group, 1994.
- [14] A. V. Aho, J. D. Ullman, and R. Sethi. *Compilers, Principles, Techniques and Tools*. Addison-Wesley Publishing House, Reading, MA, 1986.

- [15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, July 1987.
- [16] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.
- [17] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In Proc. of the ACM SIGPLAN '86 Symp. on Compiler Construction, Palo-Alta, CA, July 1986.
- [18] D. Kuck, Y. Muraoka, and S. Chen On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293-1310, Dec. 1972.
- [19] G. Goff, K. Kennedy, C-W. Tseng. Practical dependence testing. In Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation (PLDI-91), pages 15-29, Toronto, Ontario, Jun. 1991.
- [20] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102-115, Aug. 1992.
- [21] Suif Compiler Group *An Overview of the SUIF Compiler System*. Stanford University Compiler Group, 1994.
- [22] A. Darté and F. Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. *Journal of Parallel Algorithms and Applications*, 12(1- 3):83-112, 1997. Special Issue on Optimizing Compilers for Parallel Languages.
- [23] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In Proc. of the 10th SIGACT/SIGPLAN Conf. on Principles of Programming Language (POPL-83), pages 177-189, Austin, TX, Jan. 1983.
- [24] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proc. of the Supercomputing'90*, pages 407-416, New York, NY, Nov. 1990.
- [25] M. D. Smith. Extending SUIF for Machine-specific Optimizations. Technical Report, Harvard University, Cambridge, MA, July, 1997.
- [26] R. Cytron and J. Ferrante. What's in a name? -or- the value of renaming for parallelism detection and storage allocation. In Proc. of the Intl. Conf. on Parallel Processing pages 19-27, 1987.

- [27] B. Underwood. Brennan's guide to inline assembly.
http://www.rt66.com/~brennan/djgpp/djgpp_asm.html.
- [28] C. Young. *The SUIF Control Flow Graph Library*. Harvard University, Cambridge, MA, 1996.
- [29] M. Thekaulp. *Digital Video Processing*. Prentice Hall Inc., Englewood Cliffs, NJ, 1995.
- [30] D. DeVries. SUIF vectorizing compiler. *IEEE Micro*, pages 51–59, July/Aug 1996.
- [31] K. Asanovic and D. Johnson. Torrent architecture manual. Technical Report, ICSI, 1996.
- [32] M. Lam and G. Cheong. An optimizer for multimedia instruction set. SUIF Workshop Preliminary Report.
- [33] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proc. of the 5th Intl. Symp. on High Performance Computer Architecture*, pages 51–59, Jan. 1999.
- [34] A.J.C. Bik, M. Girkar, and M.R. Haghighat. Incorporating Intel MMX technology into a Java JIT compiler. *Scientific Programming*. 1999
- [35] A. Krall and S. Lelait. Vectorizing techniques for VIS. Dagstuhl Seminar on Instruction and Loop-Level Parallelism, Report # 237, April 1997.
- [36] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.
- [37] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.
- [38] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Microprogramming Workshop*, pages 183–198, Chatham, Massachusetts, October 12–15, 1981.
- [39] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22–24, 1988.

A MMX Instruction Set

The entire instruction set of MMX is summarized in Table 4.

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB,PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	Subtraction	PSUBB,PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication Multiply and Add	PMULL,PMULH PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for greater than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
	Unpack high	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		
Logical	And, And Not Or, Exclusive Or	No Distinction		
		PAND,PANDN POR,PXOR		
Shift	Shift Left Logical	PSLLW,PSLLD,PSLLQ		
	Shift Right Logical	PSRLW,PSRLD,PSRLQ		
	Shift Right Arith.	PSRAW,PSRAD		
Data Transfer		MOVD,MOVQ		
Empty MMX State		EMMS		

Table 4: MMX Instruction Set Summary