Incorporating Intel® MMX™ technology into a Java™ JIT compiler[1]

Aart J.C. Bik *, Milind Girkar and Mohammad R. Haghighat

Micro-Computer Research Labs., Intel Corporation, 2200 Mission College Blvd. SC12-303, Santa Clara, CA 95052, USA
E-mail: aart.bik@intel.com

Intel® MMX™ technology can be exploited by a Java™ JIT compiler to speedup the execution of integer operations. While translating bytecode into Intel machine code, the compiler identifies innermost loops that allow the same integer operations to be applied to multiple data elements in parallel and generates code that uses Intel® MMX™ technology to execute these loops in SIMD fashion. In the context of JIT compilation, compile-time directly contributes to the run-time of the application. Therefore, limiting program analysis-time and synthesis-time is even more important than in a static compilation model. The compiler must also ensure that arithmetic precision and the exception handling semantics specified by the JVM are preserved.

1. Introduction

The architectural neutrality of the Java™ Programming Language is obtained by compiling Java source programs into bytecodes, which are instructions for the JVM (Java™ Virtual Machine) [6,9]. A Java compiler first translates a source program into JVM bytecode that is embedded in a class file. Subsequently, the compiled program can run on any platform that provides an implementation of the JVM. The implementation may provide a simple interpreter for bytecode or, alternatively, a JIT (just-in-time) compilation can be done, consisting of a conversion of JVM bytecode into native machine code directly prior to execution. Obviously, this latter approach may substantially improve the performance of the application, while preserving the architectural neutrality.

In this paper, we focus on the techniques used by Java™ JIT compiler developed at Intel Corporation [5] to further speedup performance by exploiting MMX™ technology [3,7]. While translating bytecode into machine code for Intel 32-bit Architectures, the compiler first identifies vector loops. Such loops are innermost loops that allow the same integer operations to be applied to multiple data elements in parallel. Subsequently, the compiler converts these loops into constructs that exploit Intel® MMX™ technology to execute these loops in SIMD fashion.

Program analysis-time and synthesis-time in the JIT compiler must be kept limited because compile-time actually contributes to the run-time of the application. Therefore, our JIT compiler uses simple methods to detect and generate vector loops, rather than relying on more advanced, but also more expensive methods. Another concern for the compiler is that loop vectorization must preserve the exception handling semantics specified by the JVM. The approach taken by our JIT compiler is to generate multi-version code for each vector loop. If run-time tests indicate that no exception can be thrown by the loop, then the vector loop will be executed. Otherwise, a serial loop that precisely deals with all potential exceptions will be executed. Obviously, vectorization must also preserve the precision of all arithmetic operations.

Section 2 gives some preliminaries on Intel® MMX™ technology. In Section 3, the detection of vector loops is discussed, followed by a presentation of code generation in Section 4. The results of some preliminary experiments are presented in Section 5. Finally, conclusions are stated in Section 6. For a detailed presentation of the JIT compiler, the reader is referred to the documentation [5].

2. Intel® MMX™ technology

Intel® MMX™ technology [3,7] provides three new extensions to the Intel Architecture.

*Corresponding author.

[1]Other brands and names than mentioned here are the property of their respective owners.

Scientific Programming 7 (1999) 167–184
ISSN 1058-9244 / $8.00

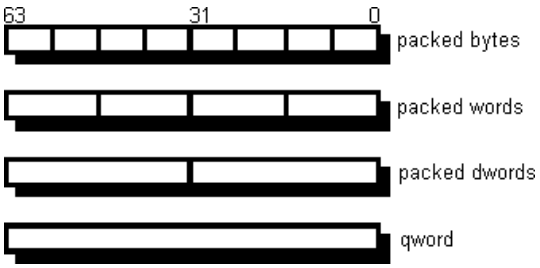

Fig. 1. MMX™ 64-bit data types.

- Eight 64-bit registers: mm0 through mm7.
- Four 64-bit data types: packed bytes/words/dwords and qwords.
- Instructions operating on the new data types.

The new register set consists of eight 64-bits registers that are aliased to the FPU data registers. As a result, MMX™ and floating-point code should not be mixed at the instruction level. Each floating-point code section should be exited with an empty FPU stack and instruction emms (empty MMX™ state) should be executed after each MMX™ code section.

As illustrated in Fig. 1, the new 64-bit data types consist of eight packed bytes (8 × 8-bits), four packed words (4 × 16-bits), two packed dwords (2 × 32-bits), or a single qword (1 × 64-bits). The MMX™ instructions implement operations on these data types (e.g., instruction paddb adds 8 bytes in the source operand to 8 bytes in the destination operand). In addition to common wrap-around arithmetic, where results that overflow or underflow are truncated, MMX™ technology also supports saturation arithmetic, where results that overflow or underflow are saturated to the maximum or minimum value of a particular data type.

The idea explored in this paper is to speedup Java array operations of type byte (8-bit), short, char (both 16-bit), or int (32-bit), using MMX™ instructions that simultaneously operate on 8 signed bytes, 4 signed words, 4 unsigned words, or 2 signed dwords, respectively.

Because all arithmetic in the JVM is done using the 32-bit type int, the JIT compiler must ensure that vectorization does not result in a loss of precision. If, however, all array stores in a loop (and all array loads to obtain a uniform vector length) have the same lower precision type (i.e., byte, short, or char), then additions, subtractions, shift-left instructions (but not shift-right instructions), and logical operations may be done using MMX™ instructions in corresponding lower precision wrap-around arithmetic. Eventually this yields the same result in the truncated part. Integer comparisons may only be done in lower precision if an array element is *directly* compared with immediate data or another array element of the same type (viz. `b[i]>a[i]`, or `b[i]<-2` for byte arrays `a` and `b`, but not `a[i]+b[i]>=3` because the addition yields an expression of type int).

3. Vector loop detection

The JIT compiler uses *control flow analysis* [1,4,8] to identify innermost loops. Each innermost loop is examined by means of *loop analysis*, *data dependence analysis* [2,10,11], and *exception analysis* to identify vector loops.

3.1. Control flow analysis

The *flow graph* of a Java method is a triple $\langle V,E,s\rangle$ consisting of a directed graph with a set of vertices V representing the basic blocks and a set of edges $E \subseteq V \times V$ representing normal transfer of control between these basic blocks (not counting potential transfer of control after exceptions). An initial vertex $s \in V$ represents the entry of the method.

A vertex $w \in V$ *dominates* another vertex $v \in V$ if every directed path from the initial vertex of the method to v contains w. The set of dominators for a vertex v is denoted by $Dom(v)$. An edge $(f,e) \in E$ is called a *back-edge* if $e \in Dom(f)$. Each back-edge in a method gives rise to a *natural loop*, consisting of all vertices that can reach f without going through the loop-entry e (including both vertices of the back-edge). Given a back-edge $(f,e) \in E$, the natural loop $L \subseteq V$ defined by this back-edge is computed as follows.

```
L := {e};
Level(e)++;
comp_natural_loop(f);
```

The procedure used in this fragment is defined below, where $Pred(v)=\{p|(p,v) \in E\}$.

```
comp_natural_loop(vertex v)
  if v ∉ L then
       L := L ∪ {v};
       Level(v)++;
       for each p ∈ Pred(v) do
         comp_natural_loop(p);
  endfor
endif
end
```

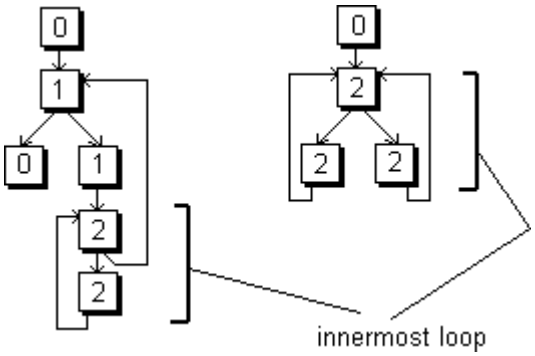

Fig. 2. Innermost loops.

Note that if we initially set `Level(v)=0` for all `v` $\in$ `V`, then after applying this algorithm to each back-edge in a method, it is straightforward to identify all innermost loops of the method. Each loop `L` $\subseteq$ `V`, where for some fixed `k`, `Level(v)=k` holds for all `v` $\in$ `L`, is an innermost loop.

In Fig. 2, we give two control flow graph examples. In the first example, there are two back-edges that define an innermost loop (consisting of vertices with level 2) and an outermost loop (consisting of vertices with level 1 and 2). In the second example, two natural loops share the same loop-entry, while neither loop is contained in the other. In this case, all vertices in this loop have the same level, so that effectively the two back-edges give rise to one innermost loop.

3.2. Loop analysis

Given a method with a set of local variables `J`, the compiler determines the set `I` $\subseteq$ `J` of *induction variables* for each innermost loop in the method. Our JIT compiler marks a local variable `i` $\in$ `I` as a stride-`c` induction variable for a loop `L` defined by a back-edge `(f,e)` $\in$ `E`, if the only instruction that modifies this local in the loop is an instruction `iinc i c` appearing in a vertex `v` $\subseteq$ `L` where `v` $\in$ `Dom(f)` (which implies that the increment is executed in each iteration), and this local variable is not further referenced by any instruction on the execution path from the increment instruction to the back-edge. The local may be referenced before the increment, though.

Because no other loops can be contained within an innermost loop, this constraint can be verified by visiting the vertices of a loop in a *reverse post-order* (i.e., a topological sort of the dominance relation). Consequently, the induction variables of a natural loop can be found in a *single* pass over the vertices in the loop. The front-end of the JIT compiler translates explicit additions and stores to `iinc` instructions when applicable, which enhances the detection of induction variables.

Subsequently, an innermost loop `L` that is defined by a back-edge `(f,e)` $\in$ `E` is marked as a vector loop candidate if the following constraints are satisfied:[2]

- The only loop-exit `(v,w)` $\in$ `E`, where `v` $\in$ `L`, `w` $\notin$ `L`, occurs for `v`=`e`, and this edge is taken on failure of a *loop-condition* that can be expressed as either `i`<`expr` or `i`>`expr` for loop-invariant `expr` and induction variable `i` $\in$ `I` with stride ± 1.
- The operand stack is empty on entry and exit of the loop.

The loop-body consists only of integer array stores, conditional statements, and induction or accumulation statements, all operating on integer expressions, where:

- All *integer* array load and store instructions have the same element type (byte, short, char, or int), referred to as the *loop-type* of the loop.
- The induction variables of the loop induce a *uniform* access stride of ± 1 on all integer array load and store instructions.
- All *reference* array load instructions (required to implement multi-dimensional integer array store or load instructions) as well as all scalar reference load or get-field/static instructions are invariant in `L`.

The latter two constraints ensure that arrays and, hence, memory are accessed contiguously. The first constraint ensures that the loop is *well-behaved* [8], which implies that the compiler can generate a run-time expression for the number of iterations of the loop.

The loop-type determines the type of MMX™ instructions (i.e., packed bytes, packed words, or packed dwords).

Consider, for example, the following fragment.

```
int dest, src, val, a[][], b[];
...
for (int i = 0; i < N; i++)
 for (int j = 0; j < i; j++, val -= 10)
     a[i][dest++] = b[src++];
```

[2] Our JIT compiler attempts to express loop-conditions in the appropriate form using some rewriting rules, including negating conditions in loops that iterate-while-false and making inclusive bounds exclusive. In addition, a simple conversion of repeat-loops into while-loops is performed to increase the number of loops that satisfy these constraints.

In the corresponding bytecode, the JIT compiler marks the local variables `j, dest, src, val` (but not `i`) as induction variables (with strides $+1$, $+1$, $+1$, and -10 respectively) of the innermost loop. An upward direction on the integer array store and load instructions is induced, whereas the reference array load (viz. an `aaload` instruction corresponding to `a[i]`) is loop-invariant with respect to the innermost loop. Hence, this loop can be marked as a vector loop candidate of type int with loop-condition `j<i`.

Note that although the induction variable in the following loop has a negative stride, still an upward direction through the array is induced.

```
for (int i = N; i >= 1; i--)
   a[N-i] = 0;
```

3.3. Data dependence analysis

Before a vector loop candidate is marked as a vector loop, the compiler must ensure that vectorization of the loop preserves the semantics of the original serial loop. Because the constraints of the previous section ensure that object creations, method invocations, and field stores do not appear in candidate vector loops, the only concern of the compiler (ignoring exceptions for the moment) is that all data dependences on arrays [2, 10,11] are preserved.

Since the Java™ Programming Language implements multi-dimensional arrays as reference arrays to other arrays, data dependence analysis must focus on the *last* dimension of all byte-, short-, char-, and int-arrays. In this manner, we correctly account for the fact that different rows of one multi-dimensional array may actually refer to an identical vector. Likewise, this approach also accounts for the possibility that some rows of several arrays are mapped onto the same vector. Since Java disallows pointer arithmetic and because all arrays begin at index 0, we can safely assume that different subscript values in the last dimension also refer to different elements within the integer vectors. To limit analysis time, our JIT compiler relies on simple tests to detect data dependences, rather than more advanced, but potentially more expensive techniques.

For each vector loop candidate, a pair-wise comparison of each integer array store with *every* integer array load or store is done. Suppose that the subscripts in the last dimension of two compared array occurrences can be expressed as `expr+i+c` and `expr+i+d`, respectively, where `expr` denotes a loop-invariant expression, `i` an induction variable with stride ± 1, and `c,d` $\in \mathbb{Z}$ two constants. Then, the references may be involved in a data dependence with distance `|c-d|`. If this distance is either zero, or if the distance is greater than or equal to the vector length (i.e., $+8$, $+4$, and $+2$ for loop-type byte, short/char, and int, respectively), then vectorization does not change the semantics of the code.

In all other cases, the JIT compiler simply resorts to disabling vectorization of the loop. Alternatively, the compiler may decide to extend the run-time tests that decide between serial or vector execution of the loop (note that our compiler always generates multi-version code, as further explained in the next section). For each two reference expressions that may give rise to vectorization-preventing data dependences, the compiler adds a run-time test that checks whether the reference expressions, which must be loop-invariant, actually contain a reference to the same vector.

Consider, for example, the following loop.

```
int a[][], b[][], c[];
...
for (int i = 0; i < N; i++)
 for (int j = 1; j < N; j++)
 a[i][j] = b[i][j] + c[j-1];
```

The bytecode for the loop-body of innermost loop is shown below.

```
aload_0
iload_3
aaload     ; load a[i]
iload 4
aload_1
iload_3
aaload     ; load b[i]
iload 4
iaload     ; load b[i][j]
aload_2
iload 4
iconst_1
isub
iaload     ; load c[j-1]
iadd
iastore    ; store a[i][j]
= b[i][j] + c[j-1]
```

The two `aaload` instructions yield loop-invariant reference expressions that merely serve to implement the multi-dimensional array operations Hence, pair-wise comparisons between the array store `iastore` of `a[i][j]` with both the array loads `iaload` of `b[i][j]` and `c[j-1]` are made. For the first com-

parison the compiler sees that, although `a[i]` and `b[i]` may refer to the same vector of int-elements, there can only be data dependences with distance 0 with respect to the innermost loop. The second comparison, however, reveals that there may be data dependences with distance +1 (which is less than the vector length +2 for int). Hence, the compiler may decide to keep the loop serial, or to generate multi-version code of the loop that, at source level, has the following form.

```
if (<potential-exception-tests> ||
    a[i] == c)
   <serial-loop>
else
   <vector-loop>
```

3.4. Exception analysis

Because the order in which operations are executed is affected by vectorization, the JIT compiler must ensure that the precise exception handling semantics imposed by the JVM specifications [9] are preserved. The approach taken by our JIT compiler is to always generate multi-version code for a vector loop. If run-time tests indicate that data dependences and exceptions cannot occur, the vector loop will be executed. Otherwise, a serial loop that precisely deals with all potential exceptions will be executed.

Because not all instructions are allowed in vector loops, our compiler only has to be concerned about the possibility of the following run-time exceptions.

(1) ArithmeticException (integer division and remainder instructions).
(2) NullPointerException (array references, array-length, and get-field/static instructions).
(3) ArrayIndexOutOfBoundsException (all array references).

Situation (1) is simply avoided by only allowing immediate nonzero divisors in vector loops. The JIT compiler generates a run-time comparison with `null` for each reference expression in the loop to handle situation (2). Sub-expressions in each reference expression are examined before the expression itself is tested (viz. expression `f.a[i]` gives rise to the test "`f==null && f.a==null`").

Situation (3) is handled by generating the following range checks, depending on the kind of array reference. Here, variable `i` denotes a stride-`c` induction variable, `i0` denotes the initial value of `i` on entry of the loop, and `n` denotes the number of iterations. Each individual range check is implemented by a single unsigned compare.

- `a[F(i)]` with `F(i)` affine: `0 <= F(i0), F(i0+(n-1)*c) < a.length`
- `a[E]` with `E` loop-invariant: `0 <= E < a.length`

Note that this kind of multi-version code generation to hoist range checks (and other loop invariant expressions) to a pre-header is also useful if it is not followed by vectorization. Therefore, these transformations are part of the standard set of optimizations applied by our JIT compiler [5].

4. MMX™ code generation

During code generation, the JIT compiler uses MMX™ technology to implement all loops that have been marked as vector loops in the previous phase. A general framework for a loop is constructed, followed by code generation for each of the constructs that may appear in the loop-body of the vector loop.

4.1. Preliminary discussion

For each vector loop, the JIT compiler generates the following general framework, where `old_entry` denotes the label of the serial implementation of the loop (alternatively, the compiler may use `dec ecx` and `jne Back` to avoid the complex instruction `loop`), see Program Code 1.

Code for each run-time test, arising from data dependence and exception analysis, is generated first. If any of these tests indicates that data dependences or exceptions could occur, a serial implementation of the loop, precisely dealing with potential exceptions will be executed. Next, code is generated that computes the number of iterations. This number is evaluated as either `expr-i0` or `i0-expr` in register `ecx` for the initial value `i0` of the loop-index, depending on whether the loop-condition is expressed as either `i<expr` or `i>expr`. Subsequently, a shift is used to divide this number by +8, +4, or +2, in case the loop-type is byte, short/char, or int, respectively. The result determines the number of iterations of the vector loop. Reaching definition analysis and constant folding are used to simplify this computation.

Code generation for the prelude, loop-body, and postlude are explained in the next sections. In the induction part, each induction variable with a former stride `c` now is given the stride `c*s`, where `s` is +8, +4, or +2 for loop-type byte, short/char, or int, respectively. The framework is terminated with an `emms` in-

```
     ...                ; evaluate a run-time test
     jne old_entry      ;     (done for all tests)
     ...                ; evaluate #iterations into ecx
     sar ecx, SHIFT_FAC ; shift by either 3, 2, or 1
     jle old_entry      ; jump if no vector work
     ...                ; prelude
Back:...                ; loop-body
     ...                ; induction code
     loop Back
     ...                ; postlude
     emms
     jmp old_entry
```

Program Code 1.

struction and a jump to the code for the serial loop to handle remaining iterations in case the number of iterations is not a multiple of the vector length.

In the remainder of this section, we use object-oriented pseudo-code to explain MMX™ code generation. Given a program construct represented by a pointer `e` to an instance of a certain class, we assume that the JIT compiler generates code for this construct by invoking the method `mmx_gen()` on the object as follows.

```
<mmi,free>:= e -> mmx_gen(F);
```

Here, `F` denotes the set of free MMX™ registers that may be used by the code generator. The method returns an MMX™ register in which the corresponding result will reside when the generated code is executed at run-time. The boolean flag `free` is set if the returned register may be freely used for further code generation afterwards, or reset if the contents of this register should be preserved (this situation can only occur for `mmi` $\notin$ `F`).

We use the following notation to denote the implementation of code generation for a syntactical construct represented by a class `Expr`.[3]

```
Expr :: mmx_gen(mreg_set F)
  ...   returns <mreg,boolean>
end
```

Because our focus here is on code generation for MMX™ technology, in many cases we simply use comments to denote code generation for particular constructs. The reader is referred to [5] for the details of this code generation.

[3]To simplify the presentation, we assume that there is a class for each of the constructs considered in the following sections, even though in the actual internal representation of our compiler such a direct correspondence may not exist.

For each vector loop, the compiler first sets aside a set of MMX™ registers for some specific tasks discussed later in this section. This set is denoted by `M`. Subsequently, the vector loop is generated by invoking method `mmx_gen(F)` on all objects that form the loop-body of the original loop, where `F`=`{mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7}-M`.

4.2. Integer array load instructions

For each integer array load instruction in a vector loop, the subscripts are either loop-invariant or induce a unit-stride in the last dimension. In the latter case, we require that access is uniformly upward or downward in all array operations of the loop and code generation for this situation proceeds as follows,[4] see Program Code 2.

Here, the value of `SCALE` is $+1$, $+2$, or $+4$, depending on whether the loop-type is byte, short/char, or int, respectively. Furthermore, the value of `DISPL` is 0 if the loop-direction is upward, or if this direction is downward, again depending on the loop-type, -7, -6, or -4, respectively. In Fig. 3, we illustrate why the displacement –7, for example, is required for downwardly accessed byte elements. Subsequently accessed elements are below the first accessed byte. Therefore, this byte must be stored in the most significant byte of the MMX™ register. For upward access, the element is simply stored in the least significant byte, which gives rise to displacement 0.

To obtain a consistent correspondence between elements in different MMX™ registers, this is also the

[4]If the set of free registers is empty when a new register is required, our JIT compiler simply resorts to keeping the loop serial. For brevity, such details are omitted.

```
Loop_Var_ALoad:: mmx_gen(mreg_set F) returns <mreg,boolean>
     ... // evaluate reference and subscript expression
         // in two different integer registers ref and sub
    let mmi ∈ F;
    emit("movq mmi, DISPL[SCALE * sub + ref]");
    return <mmi,true>;
end
```

Program Code 2.

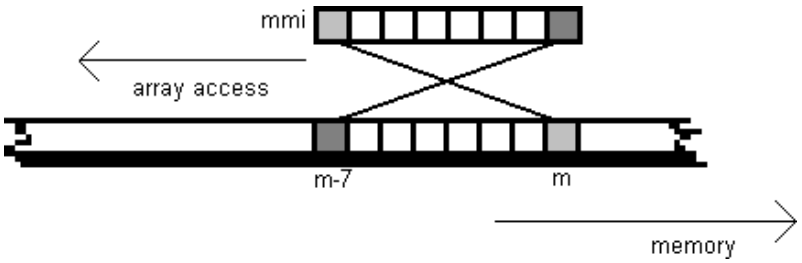

Fig. 3. Mapping memory to an MMX™ register.

reason that access must be uniformly upward or downward in one vector loop (viz. `a[i]=b[N-i]` is not allowed). Since displacements are more likely to cause unaligned memory references, generally, upward access is preferred.

4.3. Loop-invariant scalar expansions

Each loop-invariant integer scalar value, arising from constant or local variable load instructions, getfield/getstatic instructions, and loop-invariant array load instructions, must be expanded into an MMX™ register before it can be involved in a vector operation. Assuming that an integer value is stored in eax, this value is expanded (and possibly truncated to the loop-type) into MMX™ register `mm0` by using one of the three code sequences shown in Table 1.

The expansion of a byte value after it has been moved into the least significant byte of an MMX™ register using three punpcklbw instructions is illustrated in Fig. 4.

Certain integer constants allow an alternative, more efficient expansion, as illustrated in Table 2. Since MMX™ technology does not support byte shifts, not all the alternative expansions into a byte are possible.

For each different loop-invariant scalar value that is referred to within a vector loop, the compiler may decide to either pre-expand the scalar in the prelude, or to simply expand the scalar in each iteration of the vector loop. In the former case, the compiler adds a new reserved MMX™ register to the set `M`. In the latter case,

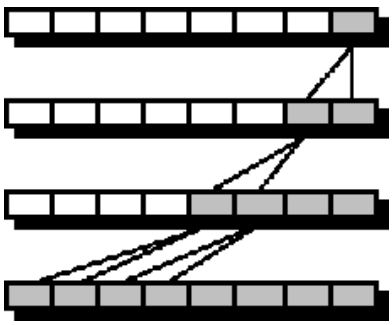

Fig. 4. Scalar expansion.

redundant expansions are performed to reduce register pressure within the loop-body. The current implementation eagerly adds registers to the set `M` for local variables and constants until the working set `F` becomes too small.

Pseudo-code for the code generation of loop-invariant integer scalar values is given in Program Code 3.

4.4. Induction variable expansions

The expansion of a stride-`c` induction variable (viz. `a[i]=i`) is implemented as follows. In the prelude, code is generated that expands (and possibly truncates) the initial value of the induction variable into a reserved MMX™ register. This register is added to the set `M`. Subsequently, the register is set to the appropriate initial value by adding one of the vector values show in Table 3, depending on the type and the direction of the vector loop. Refer to Fig. 3 to see why

Table 1
Loop invariant expansions

Expand into packed bytes	Expand into packed words	Expand into packed dwords
`movq mm0,eax`	`movq mm0,eax`	`movq mm0,eax`
`punpcklbw mm0,mm0`	`punpcklwd mm0,mm0`	`punpckldq mm0,mm0`
`punpcklbw mm0,mm0`	`punpcklwd mm0,mm0`	
`punpcklbw mm0,mm0`		

Table 2
Constant expansions

CONST	Packed bytes	Packed words	Packed dwords
0	`pxor mm0,mm0`	`pxor mm0,mm0`	`pxor mm0,mm0`
-1	`pcmpeq mm0,mm0`	`pcmpeq mm0,mm0`	`pcmpeq mm0,mm0`
2^n-1		`pcmpeq mm0,mm0`	`pcmpeq mm0,mm0`
		`psrlw mm0,16-n`	`psrld mm0,32-n`
-2^n		`pcmpeq mm0,mm0`	`pcmpeq mm0,mm0`
		`psllw mm0,n`	`pslld mm0,n`

```
Inv_IntScalar_Expr:: mmx_gen(mreg_set F) returns <mreg,boolean>
  if (pre-expanded into mmk) then
                       // no code required in loop-body
    return <mmk,false>; // but register is not free
  else
     ... // generate code that moves scalar
        // value into integer register r
     let mmi ∈ F;
     emit_expansion_sequence(mmi, r); // see tables above
     return <mmi,true>;
  endif
end
```

Program Code 3.

the increments must be reversed in case memory is accessed downward.

In the induction code, instructions are generated that increment the individual data elements of the MMX™ register with the new stride `c*s`, where `s` is $+8$, $+4$, or $+2$ for loop-type byte, short/char, or int, respectively. This vector induction step can be done using either a pre-expanded constant in another MMX™ register in `M`, or a constant that is expanded in each iteration.

Consider, for example, the following initialization of a byte array:

```
byte a[];
...
for (int i = 0, k = 0; i < N; i++, k-=7)
    a[i] = (byte) k;
```

A fragment of the bytecode of this loop is shown below.

```
Back: aload_0
      iload_1
      iload_2
      i2b
      bastore
      iinc 1 1
      iinc 2 -7
      iload_1
      getstatic N
      if_icmplt Back
```

Part of the MMX™ code generated for this fragment is shown below, where `ebx`=`i`, `ebp`=`k` and `mm6` is used to implement the vector induction. Register `eax` is used as scratch register in the prelude, and it contains the base of array a within the loop-body, see Program Code 4.

Note that although the precision of the induction variable is actually an int, exploiting the knowledge about the finally stored range of values has enabled

Table 3
Vector values for induction

	Packed bytes	Packed words	Packed dwords
upward	c*<7,..,0>	c*<3,..,0>	c*<1,0>
downward	c*<0,..,7>	c*<0,..,3>	c*<0,1>

```
      mov       eax, -56
      movd      mm7, eax
      punpcklbw mm7, mm7
      punpcklbw mm7, mm7
      punpcklbw mm7, mm7       ; expand -7 * 8
      movd      mm6, ebp
      punpcklbw mm6, mm6
      punpcklbw mm6, mm6
      punpcklbw mm6, mm6       ; expand initial value of k (in ebp)
      mov       eax, 824845084
      movd      mm4, eax
      mov       eax, 353240832
      movd      mm5, eax
      punpckldq mm5, mm4
      psubb     mm6, mm5       ; subtract 7 * <7,6,5,4,3,2,1,0>
      ...
      ...                      ; more prelude
Back: pmovq     [ebx+eax], mm6
      paddb     mm6, mm7       ; vector induction
      add       ebx, 8
      add       ebp, -56
      loop      Back
      emms
      jmp       old_entry
```

Program Code 4.

the JIT compiler to use a vector of bytes instead of integers. In the resulting byte array, the values simply wrap-around, as illustrated in Fig. 5.

4.5. Arithmetic operations

For arithmetic operations within a vector loop, first code is generated for the operands, followed by the appropriate MMX™ instructions. Code generation for an integer addition, for example, proceeds as follows, where <T> denotes byte, word, or dword, as defined by the loop-type, see Program Code 5.

Similar implementations are used for the logical operators AND, OR, XOR, and integer subtraction (both binary and unary, but without commutativity). The corresponding MMX™ instructions are `pand`, `por`, `pxor`, and `psubb/psubw/psubd`.

Logical shift-left/right and arithmetic shift-right operators are handled similarly. In these cases, however, the second operand is simply moved into an MMX™ register (i.e., without expansion), or an immediate form of `psslq/pslld, psrlw/psrld/psrlq,` or `psraw/psrad` is used when applicable.[5] To preserve the final precision, however, shift-right instructions are only allowed in case the loop-type is int. Note that because the MMX™ instruction set is not orthogonal, not all integer operations can be actually implemented (e.g., byte shifts are not supported). In such cases, the loop is left serial.

Consider the following loop that operates on an array of type int:

```
for (int i = 0; i < 100; i++)
   a[i] += 1;
```

[5] Our JIT compiler applies strength reduction and rewrites selective integer multiplication and division (e.g., by a power of 2) into equivalent shift and add operations, hereby enhancing vectorization opportunities.

Fig. 5.

```
AddOp:: mmx_gen(mreg_set F) returns <mreg,boolean>

   // Evaluate operands

   <mmi,freei>:= operand1 -> mmx_gencode(F);
   <mmj,freej>:= operand2 -> mmx_gencode(F-{mmi});

   // Place result in free register

   if (freei) then
     res1:= mmi; res2:= mmj;
   else if (freej) then
     res1:= mmj; res2:= mmi; // add is commutative
   else
     let res1 ∈ F-{mmi,mmj};
     emit("movq res1, mmi"); // move to free register
     res2:= mmj;
   endif

   // Perform operation

   emit("padd<T> res1, res2");
   return <res1,true>;
end
```

Program Code 5.

The instructions that result for the bytecode equivalent of this fragment are shown below. The set `M={mm7}` is used to pre-expand the constant, see Program Code 6.

4.6. Conditional statements

Conditional statements in a loop-body are handled as follows. First, the compiler identifies all *guards* in the loop, which are the conditions that control conditional statements. For each basic block, the compiler determines the guards that control this basic block. Subsequently, for each guard, MMX™ code is generated that computes a corresponding *bit-mask*. Finally, the compiler generates code for all basic blocks in the loop-body in *reverse post-order* to ensure that guards have been evaluated when needed. Here, conditional branches are eliminated by replacing all array stores and accumulations in a basic block that is under control of guards by the appropriate masked instructions.

Code to compute the bit-mask for each guard is obtained as follows. By means of simple rewriting rules (e.g., making an exclusive integer comparison inclusive, swapping a true- and false-branch), integer comparisons can be handled similar to the other binary integer operations. We use one of the MMX™ instructions `pcmpeqb/pcmpeqw/pcmpeqd` for all integer types, or `pcmpgtb/pcmpgtw/pcmpgtd` for signed integers. If the loop-type is byte, short, or char, then integer comparisons may only be done in the corresponding lower precision if an array element is *directly* compared with immediate data or another array element of the same type (viz. `a[i]==3` or `a[i]>a[i-1]`, but not `a[i]+b==3`).

For example, the code to compute a bit-mask for condition `a[i]==3`, where a is a byte array, is shown below. We assume that the constant 3 has been expanded as a byte value into MMX™ register `mm7`.

```
pmovq   mm0, [ebx+eax]; load a[i]
pcmpeqb mm0, mm7
```

For each guard `g` that controls a conditional statement at the end of a basic block `v ∈ V`, there is a branch that is taken when the guard is true (denoted by

```
      pcmpeqd mm7, mm7
      psrld   mm7, 31     ; expand 1
      ...                 ; rangechecks on 0, N-1
      ...                 ; base address to eax
      mov     ecx, N[esp]
      sar     ecx, 1      ; #iterations / 2
      jle     old_entry
Back:
      pmovq   mm0, [4*ebx+eax]
      paddd   mm0, mm7
      pmovq   [4*ebx+eax], mm0
      add     ebx, 2
      loop    Back
      emms
      jmp     old_entry
```

Program Code 6.

the positive guard g+) and a branch that is taken if the guard fails (denoted by the negative guard g-). Hence, if we assume that the basic block was already under control of a condition C(v), the condition associated with the true-branch e^t and false-branch e^f are $C(e^t) = C(v) \wedge g+$ and $C(e^f) = C(v) \wedge g-$, respectively. If a basic block does not end in a conditional, we simply set C(e) = C(v) for the only outgoing edge e.

Since the loop-body of an innermost loop is acyclic, we can compute the condition of each basic block in a single reverse post-order pass over all basic blocks in the loop as the disjunction of all conditions associated with all the incoming edges, where $g- \vee g+$ is rewritten into true. To simplify code generation, currently our JIT compiler only continues with vectorization in case the condition that is associated with each basic block consists of a *conjunction* of guards.

In Fig. 6, we illustrate this process for a loop-body consisting of the basic blocks B1, B2, B3, B4, and B5 (B0 evaluates the loop-condition). For example, the negative guard g- is associated with B3, which means that all state-changing instructions in this basic block must be masked using the negation of the bit-mask computed for guard g.

If the vectorization of an innermost loop with conditional statements is feasible, the compiler reserves an MMX™ register in M for each guard in the loop-body. Moreover, it also computes next-use information for each guard, i.e., the number of subsequent instructions in each iteration that depend on the guard. In the following sections, these instructions use the following function to generate code that evaluates the bit-mask for a set of positive and negative guards in a set G.

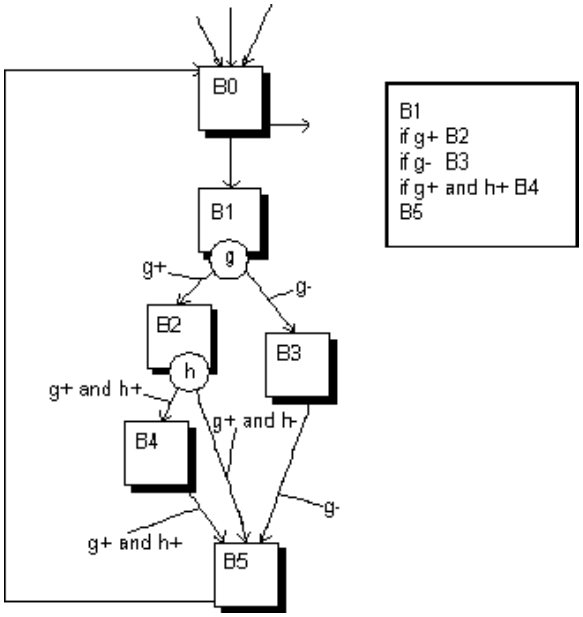

Fig. 6. Conditional statements.

This set represents the conjunction of guards associated with the basic block in which the instruction resides (see Program Code 7).

The code for a positive guard is relatively simple (see Program Code 8).

The code for a negative guard is slightly more elaborate (see Program Code 9).

4.7. Array store instructions

For each array store, code is generated that evaluates the reference and index in two integer registers, and

```
mmx_conjunction_guards(guard_set G, mreg_set F) returns mreg

  first:= true;

  let mmi, mmj ∈ F; // free registers

     for each g ∈ G do

       nxt_use(g)--;    // update next-use information of g
       mmg:= reg_of(g); // obtain register in which g resides

       if (is_positive(g)) then
          ...                    //       positive guard g

     else

          ...                    // negative guard g

     endif

     first:= false;

  endfor

  return mmi; // return bit-mask of conjunction of guards
end
```

Program Code 7.

```
if (first) then

 if (nxt_use(g) == 0) then
   mmi:= mmg;              // simply use this register directly
 else
   emit(''movq mmi, mmg''); // first move of a guard
 endif

else

   emit(''pand mmi, mmg''); // mask with guard

endif
```

Program Code 8.

the right-hand-side expression into an MMX™ register. Subsequently, if the store is guarded, the conjunction of guards is computed. In this case, the final result is obtained by combining the computed result masked on true and the old value residing in memory masked on false. In any case, eventually the result is stored into memory using a `movq` instruction. The values of `DISPL` and `SCALE` are defined as for load instructions (see Program Code 10).

Consider, for instance, the following Java source code fragment that operates on two byte arrays:

```
for (int i=0;i<N;i++)
   if (a[i] > 0 && a[i] < 100)
      a[i] = b[i];
```

Application of the method outlined above to the bytecode for this fragment yields the MMX™ instructions that are shown below, where we assume that the

```
if (first) then

  if (nxt_use(g) == 0) then
    mmi:= mmg;              // use this register
    emit(''pcmpeq mmj, mmj     // and obtain negation
          pandn mmi, mmj'');   // of first guard
  else
    emit(''pcmpeq mmj, mmj
          movq mmi, mmg
          pandn mmi, mmj'');   // negate first guard into mmi
  endif

else

  emit(''movq mmj, mmi
        movq mmi, mmg
        pandn mmi, mmj'');  // negate guard into mmi

endif
```

Program Code 9.

```
AStore:: mmx_gen(mreg_set F) returns <⊥,⊥> // no result

  ... // evaluate reference and subscript expression
    // in two different registers ref and sub
  <mmi,free_rhs>:= rhs -> mmx_gen(F);

  if (guards != ∅) then // store is guarded

  if (! free_rhs) then // make sure we can overwrite
       let mmk ∈ F;
       emit("movq mmk, mmi");
       mmi:= mmk;
  endif

  mmj:= mmx_conjunction_guards(guards, F-{mmi});

  emit("pand mmi, mmj
         pandn mmj, DISPL[SCALE * sub + ref]
         por mmi, mmj");

  endif

  emit("movq DISPL[SCALE * sub + ref], mmi"); // store result

end
```

Program Code 10.

set `M={mm6,mm7}` has been used to pre-expand the constants 100 and 0, and that the integer registers `eax` and `edx` are used to store the base address of arrays `a` and `b`, respectively (see Program Code 11).

4.8. Accumulations

Accumulations of array elements can be vectorized if the accumulator has the same precision as the loop-

```
Back:
      pmovq   mm0, [ebx+eax]
      pcmpgtb mm0, mm7        ;  bit-mask a[i] > 0
      pmovq   mm2, [ebx+eax]
      movq    mm1, mm6
      pcmpgtb mm1, mm2        ; bit-mask 100 > a[i]
      pmovq   mm2, [ebx+edx]  ; load b[i]
      pand    mm0, mm1        ; mm0 and mm1
      pand    mm2, mm0        ; mask new value
      pandn   mm0, [ebx+eax]  ; mask old value
      por     mm2, mm0        ; combine for store
      pmovq   [ebx+eax], mm2
      add     ebx, 8
      loop    Back
```

Program Code 11.

type. Roughly speaking, the following four kinds of accumulations can be dealt with in a vector loop with corresponding loop-type:

```
b   = (byte)  (b±u[i]); // byte u[], b;
s   = (short) (s±v[i]); // short v[], s;
c   = (char)  (c±w[i]); // char w[], c;
i ± = x[i];             // int x[], i;
```

For each accumulator in a vector loop, the compiler reserves an MMX™ register in the set `M`, and code is generated in the prelude that resets this register. The code to implement the actual accumulation itself strongly resembles the code for array store instructions. First, the expression that is added to the accumulator is evaluated into an MMX™ register. If the accumulation instruction is guarded, this register is masked with the conjunctions of guards next. Finally, the accumulating expression is added to the MMX™ register `mma` $\in$ `M` that is reserved for the accumulation. Pseudo-code for this code generation is shown below, where <T> denotes byte, word, or dword, as defined by the loop-type (see Program Code 12).

A similar approach is taken by our JIT compiler to implement unguarded *mixed-type* accumulations of the following two forms (with an implicit `s=1` as special case) using the MMX™ instruction `pmaddwd`.

```
acc1 ±= s * w[i];   // short v[], w[], s;
acc2 ±= v[i] * w[i]; // int acc1, acc2;
```

The core loop to implement the former accumulation, for instance, is shown below, where we assume that `mm6` contains the expanded constant `s`, and `mm7` is used as accumulator, see Program Code 13.

After any of the previously discussed accumulations has been done in a vector loop, eventually the accumulator contains either `n=8`, `n=4`, or `n=2` partial sums for data type byte, short/char, and int, respectively. Assuming that these partial sums are stored in MMX™ register `mm0`, we can move the total sum into the 32-bit integer register `eax` using one of the sequences shown in Table 4. For packed words, eventually either a movsx or movzx instruction is required, depending on whether the loop-type is short or char.

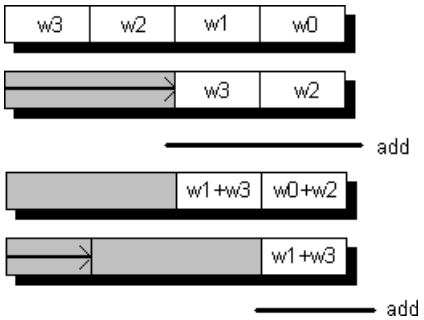

Fig. 7. Addition of partial sums.

In Fig. 7, we illustrate the accumulation of partial sums for words. The appropriate sequence of instructions is generated in the postlude. In addition, the postlude is further extended with code that adds/subtracts the contents of the integer register to/from the original accumulator and, after possibly a conversion into the appropriate type has been done, stores this sum back into the accumulator.

Consider, for instance, the following conditional accumulation into a short accumulator `s`:

```
short s, a[];
...
for (int i = 0; i < N; i++)
 if (a[i] > 5)
 s += a[i];
```

```
Accum:: mmx_gen(mreg_set F) returns <⊥,⊥> // no result

  <mmi,free_rhs>:= accum_expr -> mmx_gen(F);

  if (guards != ∅) then // accumulation is guarded

    if (! free_rhs) then // make sure we can overwrite
      let mmk ∈ F;
      emit("movq mmk, mmi");
      mmi = mmk;
    endif

    mmj:= mmx_conjunction_guards(guards, F-{mmi});

    emit("pand mmi, mmj"); // mask accumulation

  endif

  emit("padd<T> mma, mmi"); // add result to mma ∈ M

end
```

Program Code 12.

```
Back:
        pmovq   mm0, [2*ebx+eax]  ; load w[i]
        pmaddwd mm0, mm6          ; accumulate s * w[i]
        paddd   mm7, mm0          ; into 32-bit accumulator
        add     ebx, 4
        loop    Back
```

Program Code 13.

Table 4
Accumulation of partial sums

Add packed bytes	Add packed words	Add packed dwords
movq mm1,mm0	movq mm1,mm0	movq mm1,mm0
prslq mm1,32	prslw mm1,32	prslq mm1,32
paddb mm0,mm1	paddb mm0,mm1	paddd mm0,mm1
movq mm1,mm0	movq mm1,mm0	
prslq mm1,16	prslw mm1,16	
paddb mm0,mm1		
movq mm1,mm0		
prslq mm1,8		
paddb mm0,mm1		
movd eax,mm0	movd eax,mm0	movd eax,mm0
movsx eax,al	movs/zx eax,ax	

Part of the MMX™ instructions that are generated for the bytecode implementation of this accumulation are shown below, where constants 0 and 5 have been pre-expanded into `mm6` and `mm7`, respectively. Note that the current code generation naively re-loads element `a[i]` from memory into an MMX™ register, see Program Code 14.

```
Back:
 pmovq   mm0, [2*ebx+eax] ; load a[i]
 pcmpgtw mm0, mm7         ; evaluate guard
 pmovq   mm1, [2*ebx+eax] ; re-load[i]
 pand    mm1, mm0         ; mask accumulation
 paddw   mm6, mm1         ; accumulate
 add     ebx, 4

 loop    Back

 movq    mm0, mm6         ; postlude
 psrlq   mm0, 32
 paddw   mm6, mm0
 movq    mm0, mm6
 psrlq   mm0, 16
 paddw   mm6, mm0
 movd    eax, mm6
 movsx   eax, ax          ; sum of partial sums
 add     eax,    s[esp]
 movsx   eax,    ax
 mov     s[esp], eax      ; add eax to s
```

Program Code 14.

4.9. Special constructs

Some special cases are handled differently by our compiler. For example, although the methods described above can be used to generate MMX™ code for block-fills (viz.`a[i]=c`) and block-moves (viz. `a[i]=b[i]`), such operations are handled more efficiently using the rep stos and rep movs string operations of the Intel Architecture.

Although method invocations may not occur in vector loops, invocations of the static methods abs, min, and max of the class `java.lang.Math` are allowed in array operations. Provided that vector values are stored in `mm0` and, for `min/max`, in `mm1` as well, the MMX™ instructions in Table 5 can be used to implement these operations [3,7]. Here, <T> denotes byte, word, or dword, as defined by the loop-type. Note that the given implementation of abs leaves the most negative representable integer value unaffected, as is required by the Java specification.

5. Preliminary experiments

In this section, we present preliminary results of integrating a prototype MMX™ technology vectorization tool in our JIT compiler. In the experiments, the JIT compiler is invoked from within the Intel Research Virtual Machine on a Pentium® II 300 MHz system.

We have conducted the experiments with the following loops for `N=1024` and `T` $\in$ `{byte, short, int}`. The run-time of each individual loop is obtained by running that loop many times and dividing the total run-time accordingly.

```
    T a[], acc;
    ...
L1: for (int i = 0; i < N; i++)
       a[i] = (T) i;
L2: for (int i = 0; i < N; i++)
       a[i] = (T) (i \& 0x0f);
L3: for (int i = 0; i < N; i++)
       a[i] = (T) (4 * b[i]);
L4: for (int i = 0; i < N; i++)
       acc += a[i];
L5: for (int i = 0; i < N; i++)
       if (a[i] > 0)
         acc += a[i];
```

In Table 6, we show the serial execution times in micro-seconds of these loops with all default optimizations of our JIT compiler enabled (including range check hoisting), and the vector execution times when MMX™ code generation has been enabled. The corresponding speedup is shown in brackets.

From the table it becomes clear that loop L3 remains serial for byte operations, due to the lack of shift operations for bytes (required to implement the multiplication). For the remaining loops, however, we see

Table 5
Fast Abs(), Min(), and Max() operations

Abs(mm0)		Min/Max(mm0,mm1)	
pxor	mm1,mm1	movq	mm2,mm0
pcmpgt<T>	mm1,mm0	movq	mm3,mm1
pxor	mm0,mm1	pcmpgt<T>	mm0,mm3
psub<T>	mm0,mm1	pxor	mm1,mm2
		pand	mm0,mm1
		pxor	mm0,mm2 / mm0,mm3

Table 6
Execution times and speedups of preliminary experiments

	Byte	Short	Int
L1	6.9 1.7 (4.1)	10.3 3.4 (3.0)	7.0 7.6 (0.9)
L2	13.8 2.3 (6.0)	13.8 4.4 (3.1)	13.8 8.9 (1.6)
L3	13.8 13.8 (1.0)	10.3 6.7 (1.5)	13.8 9.4 (1.5)
L4	13.8 1.8 (7.7)	13.8 3.8 (3.6)	7.3 7.7 (0.9)
L5	15.3 3.4 (4.5)	15.3 6.7 (2.3)	13.8 13.8 (1.0)

that using a naïve MMX™ code generator to expose the 8-way and 4-way SIMD parallelism for byte and short data types can already help to improve the performance. Unfortunately, exposing the 2-way SIMD parallelism for 32-bit integers only yields some speedup for loops L2 and L3. These results clearly suggest that there is potential to obtain more speedup by means of a more advanced (but also more expensive) code generator. Balancing the corresponding increase in compile-time with these potential gains is a topic of ongoing research.

6. Conclusions

In this paper, we have shown how a JIT compiler can utilize Intel® MMX™ technology to improve the performance of loops that may be executed in SIMD fashion. The exception handling semantics of the original loop are preserved using multi-version code, where run-time tests decide between execution of either a serial loop that precisely deals with all potential exceptions, or an optimized vector loop in the case that the tests guarantee that exceptions cannot occur. A similar approach is taken to ensure that potential data dependences in the original loop are not violated. To limit analysis-time and synthesis-time, which actually contributes to run-time in the context of JIT compilation, our JIT compiler relies on simple methods for data dependence analysis and code generation, rather than on more accurate but potentially more expensive methods. A loss of precision of arithmetic operations is avoided by only allowing the vectorization of loops where the final result can also be obtained using the lower byte or word precision of MMX™ technology.

We have shown that a naïve translation of integer operations into MMX™ instructions already can obtain some speedup. The methods presented in this paper, however, are open for many improvements. First, here we assumed a simple MMX™ register allocation scheme in which registers are naively assigned to consecutive instructions in a loop. Using more sophisticated register assignments could reduce the number of times memory must be accessed. More careful instruction selection could combine memory load instructions that are followed by register-register instructions into single register-memory instructions to reduce code size and register pressure. Second, currently no attempts are made to schedule the resulting MMX™ instructions to minimize latency stalls. As is shown in [3], however, software pipelining is essential to fully exploit the potential of MMX™ technology. Further improvements could be obtained by improving the vector loop analysis, supporting vectorization on arrays of type long (64-bit), and providing more support for mixed-type loops. Finally, the techniques of this pa-

per may be used to speedup a wider range of numerical applications once MMX™ technology support for floating-point operations becomes available.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] U. Banerjee, *Dependence Analysis*, A Book Series on Loop Transformations for Restructuring Compilers, Kluwer, Boston, 1997.

[3] D. Bistry et al., *The Complete Guide to MMX™ Technology*, McGraw-Hill, New York, 1997.

[4] C.N. Fisher and R.J. LeBlanc, *Crafting a Compiler*, Benjamin-Cummings, Menlo Park, CA, 1988.

[5] M. Girkar, M.R. Haghighat and A.J.C. Bik, *Jaguar: A Java™ JIT Compiler for the Intel Architecture*, Intel Corporation, document under construction, 1998, 1999.

[6] J. Gosling, B. Joy and G. Steele, *Java Programming Language*, Addison-Wesley, Reading, MA, 1996.

[7] Intel Corporation, *Intel Architecture MMX™ Technology – Programmer's Reference Manual*, Intel Corporation, Order No. 243007-003, 1997.

[8] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan-Kaufman, 1997.

[9] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading, MA, 1996.

[10] M.J. Wolfe, *High Performance Compilers for Parallel Computers*, Addison-Wesley, Redwood City, CA, 1996.

[11] H. Zima, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, 1990.