Automatic Vectorization Using Dynamic Compilation and Tree Pattern Matching Technique in Jikes RVM

Sara El-Shobaky Ahmed El-Mahdy Ahmed El-Nahas
Department of Computer and Systems Engineering
Faculty of Engineering
Alexandria University
Alexandria 21544, Egypt
E-mail: {sara.elshobaky, ahmed.elmahdy, ahmed.nahas}@alex.edu.eg

ABSTRACT
Modern processors incorporate SIMD instructions to improve the performance of multimedia applications. Vectorizing compilers are therefore sought to efficiently generate SIMD instructions. With the existence of different families of SIMD instruction sets, the task of compiler writers is more complex. Moreover virtual machines, such as JVMs, are currently widely used for increasing the portability of programs across different platforms; performing SIMDization on these virtual machines would further require 'fast' compilation. This paper selects an efficient retargetable compilation technique, based on tree-pattern matching, which generates efficient SIMD code on static compilers, and studies its utility on the Jikes RVM. The paper extends BURS system used in Jikes optimizing compiler accordingly, and adds new rules for manipulating subword data for the IA-32 architecture. Initial experimental results show an overall speedup at runtime despite dynamic compilation overheads.

Categories and Subject Descriptors
D.3.4 **[Programming Languages**]: Processors---Run-time environments, Translator writing systems and compiler generators; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)---Single-instruction-stream, multiple-data-stream processors (SIMD).

General Terms
Languages, Design

Keywords
Vectorization, Dynamic Compilation, Java

1. INTRODUCTION
Vectorization is the process of converting a computer program from a scalar form into a vector form. A scalar program has arithmetic and logic instructions that operate on scalar operands (usually in pairs); whereas corresponding instructions in the vector program operates on 'vector' operands, each consisting of multiple scalar sub-operands.

Current processors incorporate a form of vector processing usually called SIMD processing, where multiple scalar operands are packed into a single register. Such hardware allows for fast computation especially for homogenous data arrays. Achieving such performance relies on the quality of vectorization technique.

Currently, vectorization is generally manually done by the programmer using compiler intrinsic. While such technique allows for utilizing the underlying SIMD architecture, it increases the complexity of programming and more importantly decreases the portability of the vectorized program.

Automatic vectorization techniques, on the other hand, tackle those issues, however, at the expense of being platform dependent. That increases the cost of retargeting the compilers for different SIMD processors.

A notable technique in the literature that provides for retargeting is that proposed by Leupers [8,9]. The technique relies on using tree-pattern matching to generate SIMD code; that potentially reduces the task of retargeting to merely writing a set of 'rules' for the target machine. However, the technique has been demonstrated on 'static' compilers, allowing portability of programs at the source-level.

This paper investigates the case of dynamic vectorization, where vectorization happens 'on-the-fly' while the program executes. Such model is currently used in Java and .Net systems and has the benefit of providing portability at the binary-level (not at the source level as traditional compilation techniques). The investigation focuses on Leupers' technique being highly retargetable, and assesses the run-time compilation complexity and achievable speedups.

This paper chooses the Jikes Research Virtual Machine (RVM) [2] as the underlying dynamic compilation system; Jikes RVM provides a flexible open test-bed for developing novel dynamic compilation optimizations. It also provides facilities for tracing, and measuring various compilation aspects. Moreover, it utilizes a tree-pattern code generator BURS (Bottom-Up Rewriting System) [4, 5] which fits nicely with Leupers' technique. It also includes many optimizations such as loop unrolling and data dependence graph analysis which simplifies our task.

ICOOOLPS '09 Genova, Italy

The paper is organized as follows: Section 2 shows related work; Section 3 gives an overview of the Jikes Research Virtual Machine; Section 4 describes briefly the code selection technique proposed by Leupers. Section 5 contains description of the proposed method. Section 6 lists the limitations of the capabilities of the method. Section 7 presents the results of experiments with the implemented system. Section 8 concludes the paper.

2. RELATED WORK

Distinct optimization techniques have been proposed to exploit SIMD instructions. In Krall and Lelait's [1] technique, a loop is unrolled and instructions from successive iterations that have adjacent memory references and independent are packed into one SIMD instruction [1]. This technique relies on the knowledge of the target instruction set which limits the unroll factor.

In the Intel C++/Fortran compiler [12], vectorization is decomposed into three phases: analysis, restructuring, and vector code generation; with a strong interaction between the first two phases. The program analysis phase performs control-flow, data-flow and data-dependence analysis to provide the compiler with useful information on where implicit parallelism in the source program can be exploited. Program restructuring phase focuses on converting the source program into a form that is more amenable to analysis and, eventually, vectorization. Then finally, vector code generation phase generates vector code for all vectorizable loops.

The GCC vectorizer [13,14] implements a loop-based vectorization technique with the adoption of a tree SSA (Static Single Assignment) optimization. The vectorizer applies a set of analyses on each loop, followed by the actual vector transformation for the loops that had successfully passed the analysis phase.

The SUIF vectorizer [15] implements a two phase source-to-source optimizer for multimedia instructions. In the first phase parallel loops are identified and instructions in the loop bodies are converted into vector instructions working on infinite length vectors. In the second phase, the code generation is applied to transform the vector operations into function calls. And vector operations that cannot be transformed will be converted back to parallel loops.

None of the above techniques uses tree pattern matching to generate SIMD instructions. Using tree pattern matching has the benefit of easier retargetability which is an advantage with the many different SIMD instruction sets in the literature. We therefore use Leupers' tree pattern matching in our dynamic compilation investigation.

3. THE JIKES RESEARCH VIRTUAL MACHINE (RVM)

The Jikes RVM is designed for use in research on fundamental virtual machine design issues. It provides a flexible testbed to prototype new virtual machine technologies and to experiment with a large variety of design alternatives.

There are three different types of compilers in Jikes RVM: one of them is the optimizing compiler which is a dynamic compiler; it compiles methods while an application is running to generate optimized code. Therefore, this optimizing compiler is the dynamic compiler used to implement the proposed vectorization technique.

The structure of the optimizing compiler is shown in Figure 1. The optimizing compiler begins by translating Java bytecodes into a *high-level intermediate representation (HIR)*. HIR is a register-based intermediate representations which provides greater flexibility for code motion and code transformation than do tree or stack-based representations.

After performing some high-level optimizations, HIR is converted to a *low-level intermediate representation* (LIR) whose operations are specific to the virtual machine's object layout and parameter-passing conventions. A *dependence graph* is constructed for each basic block. The dependence graph is used for instruction selection. Each node of the dependence graph is an LIR instruction, and each edge corresponds to a dependence constraint between a pair of instructions.

After low-level optimization, the LIR is converted to *machine-specific intermediate representation* (MIR). The dependence graphs for the extended basic blocks of a method are partitioned into trees. These are fed to a *bottom-up rewriting system* (BURS), which produces the MIR. Then symbolic registers are mapped to physical registers. A *prologue* is added at the beginning, and an *epilogue* at the end, of each method. Finally, executable code is emitted.

BURS, a code-generator generator, is a tree pattern matching system for instruction selection. Instruction selection for desired target architecture is specified by a *tree grammar*. Each rule in the tree grammar specifies the tree pattern to be matched, an associated cost (reflecting the size of instructions generated and their expected cycle times), and code-generation action (flags for the operation and the code to emit).

The tree-pattern matching performed uses dynamic programming to find a least-cost parse for any input tree. The rules are used in generating a parser which transforms the LIR into MIR.

The *JBURG* Program is a program in the Jikes RVM optimized compiler system that generates a fast tree parser using BURS. JBURG allows good instruction selection. However, it cannot be used to write productions for vector operations that need matching coverings in an entire tree of operations. Leupers has shown how with a modified *BURS* system one can achieve this result.

4. CODE SELECTION TECHNIQUE

Leupers [8, 9] presented a new code selection technique capable of exploiting SIMD instructions when compiling plain C code. It permits taking full advantage of SIMD instructions for multimedia applications, while still using machine independent source code. Most compilers use tree pattern matching with dynamic programming for code selection. This technique uses an intermediate program representation consisting of data flow trees. However, tree pattern matching with dynamic programming is not directly applicable for generation of SIMD instructions. In general SIMDization requires to simultaneously covering multiple Data Flow Trees (DFTs) instead of processing one DFT after another. This means that code selection has to be performed on full data-flow graphs (DFGs) instead of only DFTs as in traditional compiler technology.

Leupers showed a traditional technique of code selection for Media Processor with SIMD Instructions. He used SIMD instructions by considering the 32-bit data registers to be composed of either two 16-bit sub-registers or four 8-bit sub-registers, So that any full register in C programming language may store either four “byte” data type, two “short data type, or a single “integer” at a time.

The solution that Leupers prefers and then used in the current work is to generate SIMD instructions already in the code generation process during the code selection phase. It maps the machine independent intermediate representation of a program into machine specific instruction. However, The new generated code operates only on symbolic 32-bit registers, in a manner that make the existing instruction scheduling and register allocation techniques can still be used.

Code selection is concerned with mapping an intermediate representation (IR) of the source program to machine instructions of the target processor. This task can be viewed as covering the IR by machine instruction pattern. Most current code selection techniques are based on tree covering and operate on DFT based on IRs of basic blocks; where the basic block is a straight-line piece of code without any jumps. Tree covering in general produces suboptimal covers for basic blocks. Since basic blocks generally appear in the form of data flow graphs (DFGs), DFG have to be split into DFTs. This is performed by cutting DFGs at nodes representing multiple uses of values.
In the process of DFG covering, the given DFG is partitioned into multiple DFTs by cutting the DFG at the common sub-expressions (CSE) edges and computing optimal covers for each single DFT. This traditional approach is not directly capable of generating SIMD instructions, because this in general requires the consideration of multiple DFTs at a time.

Leupers overcome this problem by permitting the generation of alternative solutions during tree pattern matching. This approach is used instead of annotating only single optimal rule to each DFG node. All optimal rules are annotated including those for SIMD instructions, and only later determine the best rule globally for whole DFG. In order to achieve this, a dedicated nonterminal symbols is introduced in the tree grammar, which denote the different possibilities of using register. By applying this approach, these registers can be used as a full 32-bit register or as two separate 16-bit registers.

5. PROPOSED IMPLEMENTATION

The proposed implementation uses loop unrolling to duplicate the loop body into a certain number of instances. Corresponding instructions from different iterations are then, when possible, packed together into SIMD instructions using the tree pattern matching technique. The proposed technique is implemented using Jikes RVM and generates SIMD instructions, capable to run on the IA-32 architecture.

The automatic vectorization is performed as follows: Loop unrolling is done by the Jikes RVM optimizing compiler during the phase for converting the bytecode into HIR. All we need here is to flag the basic blocks which contains the unrolled loops. This flag is then used during the phase for conversion of LIR to MIR to detect parts of code that are candidates for SIMDization and need further processing. Thus, the phase for converting from LIR to MIR must then be modified to enable generation of SIMD code.

Tree pattern matching with *dynamic programming* is not directly applicable to generate *SIMD* instructions as this requires matching coverings of multiple *DFT*s. This means that code selection has to be performed on full data-flow graphs (*DFG*s) instead of separate *DFT*s as in traditional compiler technology.
To solve this problem, the modified BURS of Leupers is used. This requires the writing of new rules for matching similar BURS trees of nodes that may be packed into SIMD instructions, saving alternative coverings of some DFG’s instead of just one as in original Jikes RVM, as well as adding procedures for checking constraints that has to be enforced if some instructions can be SIMDized. Figure 1 shows the structure of the Jikes RVM optimizing compiler; the figure highlights the phase where such modifications are made.

Next sub-sections describe the modifications done in the phase of conversion from LIR to MIR in the Jikes RVM optimized Compiler.

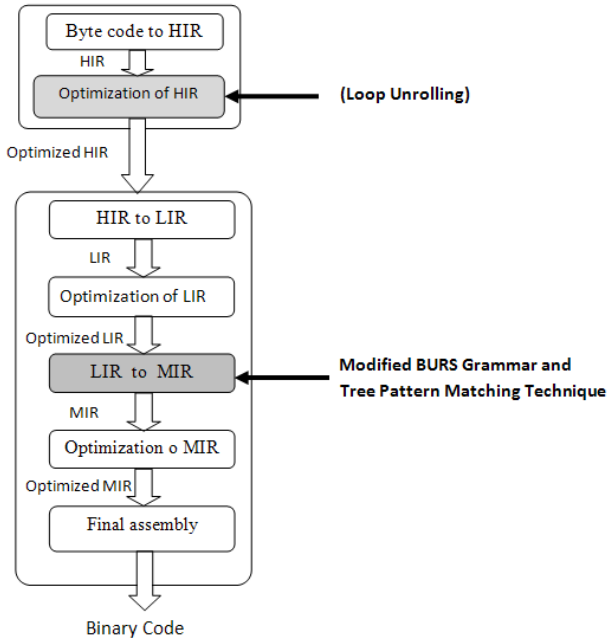

Figure 1. Modifications on the internal structure of Jikes RVM optimizing compiler

5.1 Adding New Rules to the IA-32 BURS Rules to Perform SIMDization

During the conversion from LIR to MIR, a basic block is transformed into DFGs. Then, those DFGs are transformed into DFTs [10].

During DFG covering process, each node is covered by the best cost rule that emits instructions to generate the appropriate MIR code. In the case of basic block that contains an unrolled loop, some similar DFTs will be generated corresponding to the unrolled iterations of the loop body. Consider, for example, the following unrolled loop that loads two short values from memory,

performs '*and*' operation, then *stores* resulted values into memory.

```
for (int i=0 ; i<N ; i+2){
    A[i]   = (short) ( B[i]   & C[i]   );
    A[i+1] = (short) (B[i+1] & C[i+1]);
}
```

The DFG representation of the basic block of the unrolled loop will contain two similar BURS trees each having the same rules that will cover the tree nodes during the DFG covering process. Figure 2 shows one such tree.

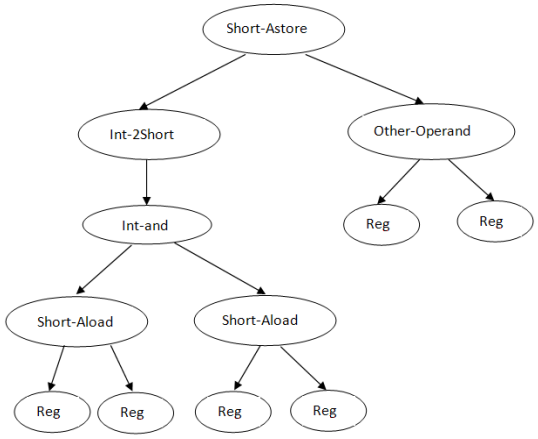

Figure 2. BURS Tree for 'And' Operation on Short data type

The goal of the proposed technique is to detect those similar trees, then, when possible, pack them together into one tree that generates SIMD code.

The packing can be achieved by naming the first tree 'the high tree' and the second symmetric tree 'the low tree'. The 'high tree' is to be covered by special rules called 'high rules', and the 'low tree' by another symmetric rules called 'low rules'. Only 'high rules' are to generate SIMD code. The 'low rules' are used to keep information on the operands and used in matching of covering without emitting any code. This mapping is described in Figure 3.

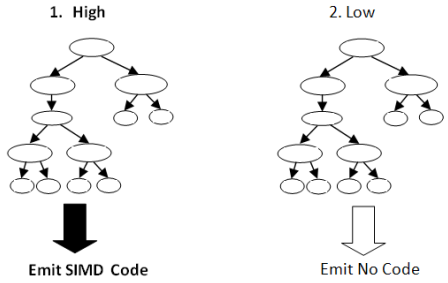

Figure 3 Generated Code from High and Low trees in Case of Short data type

As an example, consider an instruction for addition in LIR code such as "INT_ADD". This instruction adds two 32-bit registers and also writes the result into a 32-bit register. This addition can be expressed by the following rule:

- r: INT_ADD(r, riv)

Where the nonterminal symbols "r" and "riv" denote a full 32-bit registers and "INT_ADD" is a terminal symbol. Furthermore, when a SIMD instruction performs two 16-bit additions, two separate rules for modeling this behavior are described by

- r_hi: INT_ADD(r_hi, riv_hi)
- r_lo: INT_ADD(r_lo, riv_lo)

Where 'r_lo' and 'r_hi'' are nonterminals that denote the lower and upper 16-bit subregisters of a full register. Also, the same is applicable for the other nonterminals 'riv_hi' and 'riv_lo'. These two new rules are used to cover such nodes in the high and low trees respectively. The goal here is to make the high rule generate SIMD instructions while the low rule emits no code. Consequently, all other SIMD instructions are modeled in the same manner.

In general, selected trees must have the following characteristics:

- The root node is covered by three rules cover; which are: the normal rule cover, the high, and low rules which are used for the vectorization purpose.
- Operands of load and store operations must be vectors not scalar.
- Those loads/store operations perform load/store operations of 16-bit or 8-bit operands from/to memory.

Also, to identify that two trees are symmetric and guarantee the correctness of this packing process both trees are traversed in parallel to see if the following list of constraints are satisfied :

A pair of nodes N_i, N_j in a *DFG* (instructions) can be packed into one *SIMD operation*, if:

- There is no scheduling precedence between N_i and N_j
- N_i and N_j have same operator.
- According to the tree grammar rules, N_i may be located in an upper sub-register , N_j may be located in a lower sub-register.
- If N_i and N_j represents *LOAD* / *STORE* operation then they load / store values from adjacent location of memory.

Similar processing is to be done, in case of four symmetric trees with operands of 8-bit data object. In this case, the first tree will be the 'high tree' which emits the SIMD instructions, while the remaining other three 'low trees' will be used only in matching the coverings and do not emit any code. This is shown in figure 4.

In this work, new rules are written to perform vector operations for load, store, addition, negation, logic operations, etc. These rules are appended to the IA-32 BURS rules.

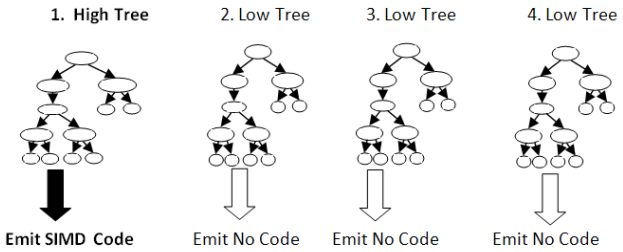

Figure 4 Generated Code from High and Low trees in Case of Byte Data Type

5.2 Modifications in JBURG Program

JBURG is a program in Jikes RVM Optimizing Compiler that generates parser from BURS rules.

In this section, we will describe the modification done in the JBURG program to allow annotation of multiple optimal rules instead of only single rule.

This JBURG program generates Java code that parses BURS trees. During the tree parsing process, it annotates the optimal rule cover for each DFG node. Therefore, the modified part allows annotation of up to three optimal solutions for each node. Those rules must have the same cost which is the lowest one, where the first rule must be the normal rule used without the implementation of vectorization then the second and the third are its corresponding high and low rules respectively.

Those three covers are described by:

- First cover is the normal optimal solution with the best cost cover. Example of such cover:

  r: SHORT_ALOAD(r, riv) → Cost =20

  Where the cost number is the cost of matching the pattern that reflects the size of instructions generated and their expected cycle time.

- Second cover is the optimal High rule which must have the same cost of the first cover. Example of such cover:

  r_hi: SHORT_ALOAD (r, riv) → Cost =20

- Third cover is the optimal Low rule which should have the same cost as the first and second cover, while also it has a symmetric template of the High rule except for the nonterminal extensions. Example of such cover

  r_lo: SHORT_ALOAD(r, riv) → Cost =20

This modification has been applied in the JBURG program to generate java code parser with a suitable structure to save all the three optimal rules solution. As well, preserve the order of the selected rules as shown in the above list for future consideration.

5.3 Modifications in Code Selection Technique

The code selection phase comes after DFG covering process with the most optimal rules. During this phase, the best node cover is determined to be selected from the available three alternatives. At this point the unrolling flag already set during the loop unrolling phase is checked to make a decision whether to continue through the goal to maximize the use of SIMD instructions or not.

To ensure a valid packing of each pair of trees, the constraints mentioned in section 5.1 are examined. Each pair of trees can be packed in a SIMD instruction if the following conditions are satisfied:

- The parallel loads and stores of sub-registers implemented by SIMD instructions actually refer to adjacent data in memory.
- There is no scheduling precedence between nodes of each tree pair.

The data dependence graph already constructed in the optimized compiler of Jikes RVM is used; where each node in the graph represents an LIR operation that can be mapped easily to a BURS tree node, and each edge corresponds to a dependence constraint between a pair of instructions. This graph contains different types of dependency edges that are traversed to find dependences between different tree pairs.

5.4 Generation of SIMD Code

If all constraints for correct vectorization are satisfied, SIMD code for such pair of symmetric trees can be generated. To allow this SIMD code generation, rules cover of such tree should be changed. Since JBURG code was earlier modified to save three different rule cover, then SIMD rule will be chosen from those rules. For example the three covers for both root node of the above tree in Figure 2 are as follows:

- stm:SHORT_ASTORE(INT_2SHORT(r), OTHER_OPERAND(r,riv))
- stm: SHORT_ASTORE(INT_2SHORT(riv_hi), OTHER_OPERAND(r, riv))
- stm: SHORT_ASTORE(INT_2SHORT(riv_lo), OTHER_OPERAND(r, riv))

At this point, the root node of the first tree will change its rule cover by the second option which is the ‘high rule’. As well, the second tree root will change its rule cover by the third option which is the ‘low rule’. After that, both trees are relabeled to make sure that all children nodes are covered by the appropriate new rules cover.

6. Limitations

The current implementation has some limitations that can be implemented in the future work. These limitations are described by:

- The current technique allows packing of two short data objects or four byte data objects only into one 32-bit data object.
- The current work is applied only into the IA-32 machine architecture.
- Supported operations includes: load, store, add, subtract, and, or, not and the array_copy operation, while the add and subtract operations are implemented assuming that a special hardware eliminate the carry propagations.
- Each SIMD tree must contain at least a load and/or store operations.

- To limit the checks needed to ensure that there is no dependency, the loop body must contain one operation. Besides, if the loop reads and modifies values of the same array, array element inside the loop must be in the form of affine array access (A[x*i+k] where x is always equal to one).
- All memory access are aligned or handled by the hardware.
- No support for constants and invariants (e.g., x [i]=6 or y).
- Supports only operations on one dimensional array.
- All memory accesses are consecutive (stride=1)

7. RESULTS

For initial evaluation of the system, we consider a simple loop and a more complex loop that are typical in multimedia applications. The first performs a simple logic operation on two arrays, and the second performs an image composting operation. To assess performance, we chose the static count of generated MIR instructions and the total run time as performance metrics. The former metric sheds light on the maximum speed up possible, whereas the latter gives the actual overall speed up, taking into consideration compilation time and various overheads.

The first loop performs 'and' operation on two short arrays as follows:

```
for(int i=0; i<N ; i++)
    A3[i] = (short)(A1[i] & A2[i]);
```

The total number of MIR instructions decreases from 44 to only 27 instructions when SIMDization is used, giving a theoretical maximum speed up [6] of 1.6. In case of the same loop but with bytes data types, the number of instructions decreases from 78 to only 29 instructions. In such case theoretical maximum speed up is 2.68.

Table 1 shows the run time required for executing 10^8 'anding' loop iterations. This large number is intentionally chosen to decrease the effect of compilation time. The two columns represent the results for anding two short data objects and two bytes data objects respectively. This loop was tested using three different compilation options: The first is compiling the loop without unrolling optimization; the second is setting the unrolling factor to one in case of short data objects, and two in case of byte data objects; the third is applying the new vectorization technique, which implies using the loop unrolling feature and generating SIMD code.

Run time(ms)	Short data-type	Byte data-type
Simple Loop	378	378
Unrolled Loop	331	318
SIMD Loop	213	128

Table 1: Run time Comparisons of Vector Anding

As expected the run time required to execute unrolled loop is better than the original loop due to decreasing the number of loop overheads. On the other hand, generating SIMD code improves performance significantly. In case of the short data-type, speed up gained by using loop unrolling is around 1.14. But when using SIMD code the speed up ratio from unrolled loop to the SIMD one is 1.6, which is the same as when comparing static instruction counts. On the other hand, in case of byte data-type the speed up ratio between unrolled loop and the SIMD one is 2.5, which is close to the static instruction count result (2.68).

Figure 5 presents the run time speedup when using our implementation with different data types. As shown, the speed up of anding 8-bit data objects is greater than anding 16-bit data objects, which is expected due to fixed machine word size.

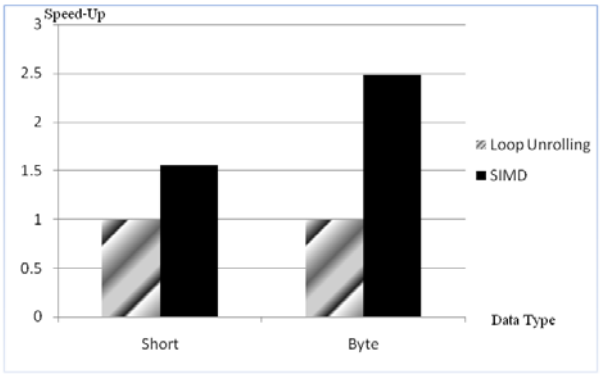

Figure 5 Speed up graph for short and byte data-types

Decreasing the compilation time is highly recommended since it affects the overall execution time. Figures 6 and 7 represent the total execution time of the 'anding' loop in both short and byte data-types respectively. The two curves represent executing the loop in two different conditions: Firstly, by applying the new vectorization technique, and secondly, without applying it. The overhead of compilation with the new technique is amortized by the run time reduction due to SIMDization. However, achieving the maximum potential speed up requires substantial number of loop iterations (in the order of 10^7 loop iterations).

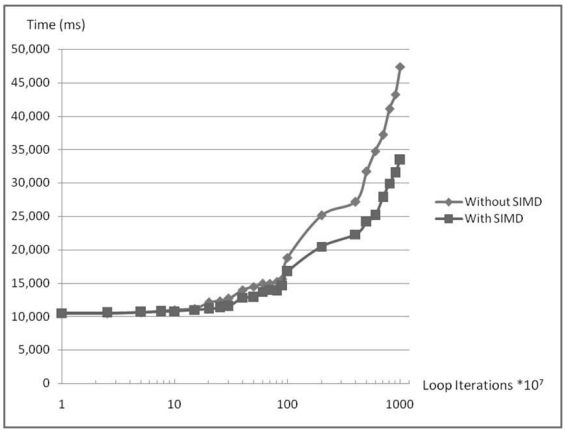

Figure 6 Graph of total execution time of vector and with short data-type

The worst case would appear in case of loop carried dependence. In such case an extra compilation time using the new vectorization technique will occur. While, the run time of the program does not change since there is no exploitation of SIMD code. For example in case of figure 6 and 7 loops, if a dependency exists, no runtime reduction will occur, but only a small compilation overhead which is no more than 20 ms will be introduced.

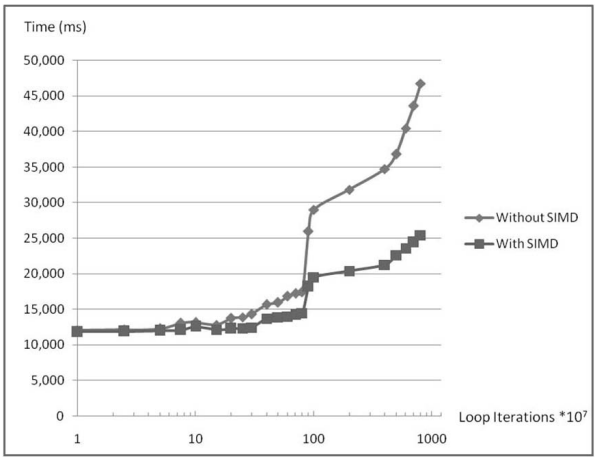

Figure 7 Graph of total execution time of vector and with byte data-type

The second loop we analyzed is the 'image compositing' loop. Image composting generally combines two images to create special effects. The loop we considered performs 'fade-in-fade-out' effect. The computation performed between images A and B represented as:

C = fade * (A-B) + B

Where A, B are two input images, and C is the resultant blended image.

SIMD instructions are exploited by packing four byte instructions into one SIMD instruction. Table 2, below, shows a run time breakdown for executing the loop; the run time speed up resulted is 1.14. This small speed up is due to the overhead of other non SIMDized instructions, loop branching and boundary checks. As a result the total compilation time increased. The new vectorization technique has larger compilation time compared to original unmodified Jikes RVM. That is due to the modifications in the instruction selection phase. The modified selection phase took 47ms more; 7 ms for finding tree pairs and checking SIMD constraints, 40 ms overhead of covering all DFGs nodes with multiple rules.

Time	SIMD	Without SIMD
Total compilation time (ms)	11665	11601
Instruction selection (ms)	1394	1347
Run time (ms)	18.179	20.703

Table2: Compilation time and runtime of the composite loop

8. CONCLUSION

This paper presents initial results for a dynamic compilation scheme that systematically vectorizes loops in the absence of memory alignment constraints. Experimental results indicate speedup factors 1.6 for 2 data objects per vector and 2.5 for 4 data objects per vector for a simple loop. And a speed up of 1.14 for a more complex loop example. Compilation time required to perform SIMDization slightly increases the overall execution time required to run a program in Jikes RVM. This overhead in addition to normal dynamic compilation overhead can be negligible with large number of loop iterations leading to significant improve in performance (by exploiting SIMD instruction). However, SIMDization overhead still occurs if conditions required to pack a SIMD pair is not satisfied.

REFERENCES

[1] Krall, A. and Lelait, S. 2000. Compilation Techniques for Multimedia Processors. Int. J. Parallel Program. 28, 4 (Aug. 2000)

[2] IBM. Jikes Research Virtual Machine (RVM). http://jikesrvm.org/User+Guide, 2005.

[3] Michael G. Burke , Jong-Deok Choi , Stephen Fink , David Grove , Michael Hind , Vivek Sarkar , Mauricio J. Serrano , V. C. Sreedhar , Harini Srinivasan , John Whaley, The Jalapeño dynamic optimizing compiler for Java, Proceedings of the ACM 1999 conference on Java Grande, p.129-141, June 12-14, 1999, San Francisco, California, United States

[4] Fraser, C. W., Hanson, D. R., and Proebsting, T. A. 1992. Engineering a simple, efficient code-generator generator. ACM Lett. Program. Lang. Syst. 1, 3 (Sep. 1992)

[5] Christopher W. Fraser , Robert R. Henry , Todd A. Proebsting, BURG: fast optimal instruction selection and tree parsing, ACM SIGPLAN Notices, v.27 n.4, p.68-76, April 1992

[6] Zhao, J., Rogers, I., Kirkham, C., Watson, I.: Loop parallelization for the Jikes RVM. In:. Proc. of the 6th International Conference on Parallel and Distributed computing, Applications and Technologies, 2005

[7] Hennessy, J. L. and Patterson, D. A. 2002 Computer Architecture: a Quantitative Approach. 3rd. Morgan Kaufmann Publishers Inc.

[8] Rainer Leupers, Code selection for media processors with SIMD instructions, Proceedings of the conference on Design, automation and test in Europe, p.4-8, March 27-30, 2000, Paris, France

[9] Rainer Leupers, Steven Bashford, Graph-Based code selection techniques for embedded systems, March 2004.

[10] Steven Bashford , Rainer Leupers, Constraint driven code selection for fixed-point DSPs, Proceedings of the 36th ACM/IEEE conference on Design automation, p.817-822, June 21-25, 1999, New Orleans, Louisiana, United States

[11] Steven S. Muchnick, Advanced compiler design and implementation, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1998

[12] Bik, A. J., Girkar, M., Grey, P. M., and Tian, X. 2002. Automatic intra-register vectorization for the Intel architecture. *Int. J. Parallel Program.* 30, 2 (Apr. 2002).

[13] Dorit Naishlos. Autovectorization in gcc. In the GCC Developer's summit, pages 105-118, June 2004.

[14] P. Lesnicki, M. Cornero, A. Cohen, G. Fursin, A. Ornstein, and E. Rohou. Split compilation: an application to just-in-time vectorization. In Workshop on GCC for Research in Embedded and Parallel Systems (GREPS'07), Brasov, Romania, September 2007.

[15] An Optimizer for Multimedia Instruction Sets (A Preliminary Report) – Cheong, Lam – 1997