1. Introduction

The use of vector instructions, also called SIMD instructions (Single-Instruction Multiple-Data), may dramatically improve the performance of some programs through data parallelization. Compilers are able to generate them by means of an optimization technique called vectorization. According to our measurements shown in Fig. 1, it is by far the most efficient optimization technique among loop tiling, loop nest interchange, unroll and jam and vectorization for our dataset. However, in some cases, vectorization may actually dramatically reduce performance. As shown in Fig. 2, this is the case for 6.17% of our benchmark programs, down to a speedup of 0.25, that is, four times slower. In order to avoid such a situation, modern compilers try to vectorize only when they predict it to be profitable. To do so, they leverage ad-hoc performance-prediction algorithms. However, our measurements show that they tend to be very conservative.

In the remainder of this article we focus on the tensor contraction kernels (TC). Those are very common computation kernels, at the hot path of a significant portion of the scientific programs executed on the Supercomputer Center at Kyushu University (e.g. molecule-structure simulations). Moreover, we focus on the Intel Compiler over the GCC and LLVM compilers for two reasons: (1) GCC and LLVM almost invariably fail to optimize the TC kernels, (2) the Intel Compiler provides a way to force vectorization to vectorize through the command line, thereby bypassing its performance-prediction heuristics. We measure in Sect. 4 (Fig. 2) that, when we force vectorization, 64.18% of our dataset is at least 5% faster that when we use the default of the Intel Compiler. In other words, even though it best optimizes the TC kernels over GCC and LLVM, the Intel Compiler still misses a vectorization opportunity for more than half of them. To mitigate this situation, we could always force vectorization; that would, however, strongly negatively impact the performance of the 6.17% of TC kernels that become much slower with this strategy (at the left...
memory-access intensive computation kernels. This consti-
tates the first contribution of this paper. It is an important
step forward for two reasons. First, it makes our prediction
method easier to embed into current compilers as it faster (it
does not require to compile once), and does not require any
feedback loop from the output to the input of the compiler.
Second, it potentially opens the way to more complex prob-
lems, where one has to choose between thousands of optimi-
ization strategies (two in this paper). In such configuration
indeed, the time to extract the software characteristics is a
fatal bottleneck to all methods that require to generate the
assembly of the compiled program. When designing soft-
ware characteristics, there is no rule of thumb to make sure
that we are actually capturing anything useful about the pro-
grams to compile, with respect to a given goal. To address
this situation, we introduce a technique that enables us to
qualitatively assess this property by means of dimension re-
duction and visualization. It is the second contribution of
this paper.

This article is organized as follows. We start with an
overview of the important techniques and notions used in
our work in Sect. 2. Next, in Sect. 3, we introduce the re-
lated work, especially why the existing techniques are not
appropriate for the TC kernels. In Sect. 4 we explain how
we model the problem, especially our set of software char-
acteristics. The next three sections present our experimental
results: in Sect. 5 we detail the experimental setup; in Sect. 6
we detail the prediction accuracy of our technique, as well
as the performance improvement it enables; in Sect. 7 we
explore the reason why our set of software characteristics
performs better than the related work by means of visual-
ization. We conclude the article by a discussion about the
potential application of our technique (Sect. 8).

2. Background

2.1 Tension Contraction Kernels

Our dataset consists of tensor contraction kernels. They
are very common scientific-calculation kernels, especially
in molecule-folding simulations. They consist of $k$ perfectly
nested loop with a single multiplication-accumulation state-
ment inside the innermost one. This innermost statement
features three memory loads and one memory store that may
result in irregular memory access patterns: this makes such
kernels challenging to optimize. There is no specific limita-
tion in the depth of tensor contraction kernels, but we only
consider the ones of depth 4 in this article$^1$; we refer to them
with the abbreviation TC4. TC4 kernels operate on 3 matri-
ces $A$, $B$ and $C$, two of them being of dimension 3, and one
dimension 2. Each dimension has the same size, which we
call $N$. In the case of TC4, the innermost statement is
executed $N^4$ times. An example of such kernel for $N = 64$
is given in Fig. 3 using the C language.

One can generate many TC4 kernels by varying: (1)
the dimensions of \( A \), \( B \) and \( C \), 3 or 2; (2) the order of the array indices at line 8, providing that each induction variable appears exactly 2 times, at most once per access; (3) \( N \), the size of each dimension of the arrays. It is possible to fully characterize a TC4 kernel with a short description string with the following format: AAA-BBB-CCC, where AAA, BBB and CCC are the indices used to access arrays \( A \), \( B \) and \( C \) respectively. We call the string the workload id, or wid in short. For example, the wid of the kernel in Fig. 3 is \( ij-ikl-jkl \). We always consider that the induction variables corresponding to the loop nests are \( i \), \( j \), \( k \) and \( l \) in this order. Therefore, the wid characterizes the source code of a kernel, independently from the size of the arrays.

2.2 SIMD Performance of Our Dataset

We have measured that Intel Compiler version 12.1.5 applies vectorization differently depending on the alignment of the memory accesses with the SIMD width\(^1\). Indeed, in the SSE3 instruction set, the SIMD arithmetic instructions only operate from SIMD registers, whose width is 128 bits (4 single-precision-floating-point scalars). One has to load and store data between memory and these registers: we call such operations SIMD load/store operations. SIMD load/store operations are more efficient when the locations of scalars in memory are consecutive and aligned with 128 bits. In this case it is possible to move the 4 single-precision-floating-point scalars at once using an aligned packed instruction. When locations are consecutive but not aligned, it is still possible to use a single different non-aligned packed instructions: its execution is however far slower. In other cases, one has to load/store scalars independently and pack/unpack them into the SIMD registers using dedicated instructions. This is the slowest situation and we want to avoid it as much as possible. However, TC kernels idiosyncratically feature at least one such case along one of the four loop indexes \( i, j, k, l \).

Figure 4 shows the speedup of the TC kernels forcibly vectorized, compared to when compiled with vectorization forced off. The highest performance is achieved for \( N \) multiple of 4, and the lowest is achieved when \( N \) is an odd number. For \( N = 4k+2 \) Intel Compiler generates two versions of the inner loop body, one vectorized and one scalar, and uses an if statement to execute the appropriate one. In the vectorized loop body, it uses exclusively aligned SIMD load/store.

---

\(^1\)This only applies for when the dimension size of the input arrays is known statically, which is the case in our experiments.
to our dataset\footnote{Although they proved to be efficient for the problems defined in the articles that introduced them.}.

3. Related Work: Use Machine Learning to Improve Compilation

Using machine learning to improve compilers is not a new idea, and the literature already contains several works that apply support vector machine (SVM)\cite{4, 14}, nearest neighbor (NN)\cite{13, 14}, artificial neural networks (ANN)\cite{3, 5}, and logistic regression\cite{9}. They however make different types of predictions: Stephenson and Amarasinghe\cite{14}, Park et al.\cite{4} and Agakov et al.\cite{13} determine parameters of code optimizations; Kulkami et al.\cite{3} order optimization passes in the middle end; Pekhimenko and Brown\cite{9} use machine learning to focus search algorithms.

Our work aims at predicting if whether or not we should set the Intel Compiler compiler’s option `vec-threshold0`. This boils down to a simple optimization-space exploration (OSE) problem. Given a program to compile, OSE consists in finding the best optimization strategy inside a given optimization space. The definition of an optimization strategy depends on the problem. For example in our work, we consider two strategies, depending on whether or not we set `-vec-threshold0`. From this point of view, Stock et al.\cite{5} is a superset of our work because it considers a larger optimization space. Still, our work exhibits four major differences that makes it more scalable. The first difference is that Stock et al. express the OSE problem in term of regression while we use classification. The second difference is that we propose a very different software characteristic set, as explained below. The third difference is that we vary the size of the input arrays, thereby trusting the machine learning model into understanding the variations of the performance with \(N\) explained in Sect. 2.2. The last difference is that we extract the software characteristics before actually optimizing. The approach from Stock et al.\cite{5} requires to compile the program once in order to extract the software characteristics. This is less practical because (1) it requires more time to extract the software characteristics (2) it makes it more complex to integrate the technique into the compilation flow. Still, our method achieves the same speedup numbers, thereby being superior overall.

The set of software characteristics is critical to ensure the success of such machine-learning driven compilation techniques. It is however challenging to design such set, and many have been proposed. First, the software characteristics may be measured dynamically at runtime or statically from sources. Hoste and Eeckout\cite{12} propose a comprehensive example of the former. Such program characteristics make it possible to gather far more information. Their scheme however requires to actually execute the program: this is not something we can afford inside a compiler for obvious time-related constraints. This is why we only consider static software characteristics (SSC) in our work. SSC can be further extracted from the optimized program\cite{5} or from the non-optimized one\cite{2–4, 6, 13, 14}. Those often consist of a vector of numbers that express important properties of the program. The most popular of these sets is GCC milepost from Fursin et al.\cite{6}, which consists of 46 numbers extracted from the intermediate representation of GCC. The work from\cite{8} is original: it leverages genetic algorithm and predictive performance modeling in order to automatically determine the important numbers to be included into the SSC set. However, the set it generates is very similar to the one proposed by the other works previously cited; to make things worse, the performance improvement they exhibit is very low: around 5% only. Park et al.\cite{4} propose a different, original approach to define SSC. They leverage graph mining techniques to directly feed the program’s data-flow graph to a SVM, and predict the best optimization scenario. They compare this approach with Milepost GCC and yield better prediction accuracy.

All these approaches however share a common weakness: they merely consider the important characteristics of the control flow graph, or the static counts of instructions per type. The programs we consider in this article exhibit the exact same control flow graph and instruction sequence, making all these approaches inappropriate. This has motivated us to design new SSC sets: proposal1 and proposal2.

Finally, G. Fursin et al.\cite{7} have proposed the notion of iterative compilation; they get rid of software characteristics altogether, and consider (among other) the performance of a program for a given set of optimization strategies in order to predict for the same program. While robust, this approach is obviously only relevant for large optimization spaces, which is not the case in our study.

4. Modeling of the Problem

4.1 The Baseline Software Characteristics

Software characteristics consist of numerical values that describe the program to compile. They are fed as input to our prediction models. In order to be integrated before the compiler, we only consider software characteristics that are measured statically, that is, without actually executing, and before compiling. We consider as baseline the three sets below selected from the related work:

**Milepost.** We consider GCC Milepost\cite{6} in our experiments, although it does not grasp any information about our benchmark as explained earlier. It is representative of the methods that rely on information on the control flow graph or the static instruction count including the one from Park et al.\cite{4}.

**Assembly.** This corresponds to the static instruction count of the vectorized assembly, as proposed by Stock et al.\cite{5}. Because it requires to actually compile the code it breaks our constraints detailed in Sect. 1. Yet we consider this set in our experiments as a target.

**Random.** Additionally, we consider a set of software char-
characteristics made of 3 random numbers; it constitutes an important baseline for two reasons. First, if a given set of software characteristics yields a similar accuracy as random, it means that this software characteristics does not characterize the TC4 kernel. Second, if random yields high accuracy, it means that our experimental setup is flawed.

4.2 Our Software Characteristics

As discussed earlier, the sets of software characteristics proposed by the related work are ill-adjusted to our problem. Therefore, we have to devise our own. We propose to follow two sets of software characteristics: Proposal1. For our first set of software characteristics, proposal1, we propose to feed the machine learning device with important properties of the memory access pattern of TC4: the sequentiality of memory accesses; the re-use of data; and whether or not the memory accesses are sequential along the innermost loop nest (index l). It is inspired from the observations made in Sect. 2.2. It consists of three positive integer values per memory access, as detailed in Fig. 5, that is, 9 in total. We consider the load and store from/to A as a whole to avoid repetition; therefore we need 9 positive integer values to describe one TC kernel. This is detailed in Fig. 5 a for an access $x$ to a 2-dimension array. On the bottom is also given the exact value of this software characteristics for the TC4 kernel of Fig. 3, with the wid $ij$-$ikl$-$jkl$. We decided to evaluate these properties after thoroughly analyzing the important properties of TC4 with respect to automatic vectorization.

Proposal2. Our second set of software characteristics, proposal2, starts from the assumption that the compiler processes the kernels that exhibit similar properties modulo nest interchange in a similar fashion. There is no need to explicit these properties, we only need to teach the machine learning device which parts of the array accesses can be transformed from one to the other by means of loop nest interchange. To do so we define proposal2 so that it describes the repetition pattern in the wid. For instance, the first value is 1 if and only if the rightmost index of the access to A is the same as the leftmost index of the access to B; it is 0 otherwise. An example is given in Fig. 5 b.

4.3 Expression as a Classification Problem

In this section we propose a method to decide whether or not to force automatic vectorization on behalf of Intel Compiler and the compiler user. We use the setup described in Fig. 6: our classifier takes into input the set of software characteristics detailed previously as well as the size of the array dimension, and decides the option of the compiler depending on the predicted class, as detailed in Sect. 6. To this end, we set an arbitrary threshold of 5% in order to define the notion of significant speedup difference. With this definition, we can divide the data set into 3 classes: class 0 when the vectorization speedup is more than 5% lower for vectorization forced than non forced; class 2 when it is more than 5% larger; and class 1 in between. Ideally, we want to predict each class so that we set the option -vec-threshold0 for elements of class 2, and not for the ones of class 0. The prediction does not matter for class 1 (the option does not have any effect on performance in this case). It is important to notice that we directly feed $N$ to the machine learning device, and not $N/4$ as it might seem natural from our study of Sect. 2.2. Indeed, not only the latter approach would be less generic\(^4\), our experiments (not detailed) show that it does

---

\(^4\)The value 4 would need to be adjusted to the SIMD width of the target processor.
not raise the accuracy of the trained models anyway.

Figure 2 shows for our dataset the speedup compared to the program compiled without SIMD instructions with the command line:

```
icc -novec
```

the grey line corresponds to the default behavior of the compiler:

```
icc -vect
```

and the black line is when we force vectorization forced:

```
icc -vec-threshold0
```

The respective position of the lines tells us which option is most profitable performance-wise: when the grey line is above the black one, this means that the default behavior of the compiler is conservative and optimal (class 0); on the other hand, when the black line is above, this means that the compiler is missing a vectorization opportunity (class 2). The TC kernel for which both lines are approximately at the same position are in class 1. We can see that upon applying automatic vectorization, Intel Compiler takes the correct decision in 29.63% of the cases. It is too conservative for 64.18% of our data: by failing to vectorize, it misses speedups up to 2.5 times; it is on other hand not a good idea to always force vectorization for 6.17% of the data: this might slow down the compiled program by up to 4 times.

The grey line is under 1 for less than 1% of our dataset. It corresponds to the situation where the compiler is actually too aggressive, and vectorizes although it is not profitable. We do not focus on this situation in this article.

5. Our Experimental Setup

In the remainder of this article, we consider the same 1500 randomly generated TC4 kernels. We vary $N$, the dimension of array dimensions, from 10 to 100. This represents in total $1500 \times 91 = 136500$ points. We focus on single-precision-floating-point calculation (float type in C). Our test machine is an Intel Core2 Extreme based on the architecture Merom. It features 4 processing cores, we however only use one. The processor supports the SSE3 SIMD instruction set, that is, 128-bit vectors; this corresponds to 4 float scalars. This is the same vector instruction set as Intel Silvermont, the state-of-the art micro-architecture from Intel for embedded systems. Our kernels are implemented in C language. The arrays are declared globally, and their size is set with a constant. We make sure the compiler aligns their starting address with the SIMD width. We compile using Intel C Compiler version 12.1.5, using the default optimization level. The command line is:

```
icc -xSSE3 -O2
```

In particular, Intel Compiler carries out automatic vectorization at this level. The documentation of Intel Compiler suggests to use -03 for computation intensive programs; our measurements (not detailed here) however show that -02 is by far the best performing option in the specific case of TC4 kernels, never being the worse. When specified, we force vectorization by means of an extra option, -vec-threshold0; the command line becomes:

```
icc -xsSSE3 -vec-threshold0
```

With this option, Intel Compiler applies vectorization even if it predicts the probability for it to be profitable to be 0%. Each data point is the resulting geometric mean of 5 measurements; data is discarded and re-measured if the standard error of these five measurements is larger than one fifth of the arithmetic mean, that is, if the coefficient of variation is larger than 0.2. We do not use other compilers such as GCC or LLVM because they do not provide such options.

As for the machine learning technique, we use support vector machine (SVM) with a Gaussian Kernel\(^1\), as provided by the R package e1070 [1]. All the accuracy numbers are calculated by means of a 3-fold cross validation procedure on the wid. In particular, we make sure that all the data taken on kernels with the same width do not span across both the training and test sets. This is very important because this reproduces the situation where the compiler is faced with a never-encountered program. Therefore, our predictions are the ones a compiler user would obtain in real situations.

6. Experimental Results

6.1 Classification Accuracy

For the data set shown in Fig. 6, we predict the class by

---

\(^1\)We have observed that the accuracy does not significantly change with the predictor, therefore we only show our results with SVM. This situation is coherent with the current common wisdom in the field of machine learning [15].
means of support vector machine for the sets of software characteristics introduced in Sect. 4, under the setup described in Sect. 5.

Figure 7a shows the prediction accuracies for each class and each set of software characteristics. First, random yields low accuracy overall and for each class as expected, around 50%. Then, our two proposed sets of software characteristics, proposal1 and proposal2, exhibit significantly higher accuracy; this means that they indeed provide useful information for our predictions. On the other hand, milestone yields notably low accuracy figures, very close to random overall. Finally, assembly yields the highest accuracies for each individual class as well as in general. This is because performances is largely decided by the vector instructions generated by compiler for memory load/store. This result is coherent with the results obtained by Stock et al. [5]. Still, proposal2 is almost as accurate as assembly: 91.48% and 97.37% respectively. However, proposal1 does not achieve as high accuracy as expected; as we have observed from the generated assembly files, this is because Intel Compiler carries out loop interchange internally, changing the value of the properties measured by this software characteristics set. Additionally to Stock et al. [5], we vary \( N \), the size of the dimensions of the input arrays. Still, the machine learning model figures out by itself the variation of the performances with respect to \( N \), as studied in Sect. 2.2.

We notice that the accuracy for class 0 is invariably significantly lower that for the other classes. After investigation, we have determined that most of the mis-predictions for class 0 are actually predicted as class 1. This is not a concern with respect to our final target which is to decide if whether or not the option vec-threshold0 should be set. Indeed, it is enough to decide not to set the option for kernels predicted classes 0 and 1, and to set it for class 2. This corresponds to the Fig. 7b. Classes 0 and 1 are merged into class 0+1. After re-training the SVM, it yields very high accuracy figures for class 0+1, especially for proposal2 and assembly: 98.45% and 99.5% respectively. The overall accuracy is also slightly up for all the software characteristics.

6.2 Effects on Performances

This section shows the effect of the results of Sect. 6.1 on the performance of programs. We use the results as follows: given the predictor of Fig. 6, we compile with the command line

\[
icc \ -xsse3 \ file.c
\]

if it predicts class 0 or 1, or

\[
icc \ -xsse3 \ -vec-threshold0 \ file.c
\]

if it predicts class 2.

With our method, we expect to get speedup for class 2 and to avoid speed-down for class 0. It has no effect on class 1. The distribution of the speedups due to our method for the whole dataset is shown in Fig. 8. We only show the results for kernels originally in classes 0 and 2, that is, 50.63% of the data (see Fig. 6). As a reference, we further display the results for the perfect predictor; it sets the upper limit for the other sets of software characteristics.

The ranking of median speedups approximately matches the ones obtained for overall accuracy. The best sets of software characteristics are assembly and proposal2: both yield 1.69 time speedups. This is very close to the theoretical maximal of 1.7, shown by the target. On the other hand, milestone performs as bad as random; this can be explained by its dramatically low accuracy for class 2, as shown in Fig. 7 (36.2%). Finally, proposal1 stands in the middle. In conclusion, we can say that proposal2 provides the best set of software characteristics: not only it is the best suitable for compilation, but it also yields very high speedup figures, very close to the maximum.

Yet, the worse case for proposal2 is a dramatic 3-time slowdown. It is reached for the wid \( li\)-\( kj\)-\( kij \) and \( N = 12 \), which corresponds to a data predicted as class 2 although it is in class 0. It is a singularity: other values of \( N \) for the same wid achieve profitable vectorization. There are exactly 9 such miss-predictions in the whole data set for proposal2 (0.006% of the dataset). A close study reveals that
The main contribution of our work is the set of software characteristics, introduced in Sect. 4.2. We have observed in Sect. 6 that the accuracy of our predictions highly depends on the software characteristics. We have already emitted a hypothesis in order to explain these numbers from the inherent properties of the programs. In this section, we propose to rather use visualization in order to try to qualitatively explain the performance of the software characteristics. In particular, we want to answer the following two questions: 1) what are the intrinsic characteristics of the sets of software characteristics to yield better predictions? 2) would it be possible to yield better figures with other prediction techniques?

In this section, we propose to qualitatively answer to those questions by means of visualization. To this end, we project the TC kernels in the space of the software characteristics, colored by target class. In such visualization, the points close to each other are the TC kernels with similar software characteristics. If the colors appear mixed, that would mean that some TC4 kernels with similar software characteristics yield different classes. In other words, that would mean that the software characteristics do not separate the classes, independently of any machine learning technique. If, on the other hand, the colors appear well separated, that would mean that is theoretically possible to predict the class: one would only need a machine learning technique that is able to find the frontiers between the clusters of different colors.

We show in Fig. 9 the projection of the data of Sect. 4 for each set of software characteristics. For clarity, we only show a representative close-up of the whole data. In order to create a human-readable instance of such visualization, we project all the data points from the many-dimension space of software characteristics to a two-dimension plane. We use a method proposed by van der Maaten et al. [11], called t-Distributed Stochastic Neighbor Embedding, or t-SNE in short. It is a dimension-reduction technique meant for visualization that strives to reproduce in 2D the similarities between data located inside a many-dimension space. To this end, it first considers similarity measures in both spaces; then it calculate the 2D coordinates by solving an optimization problem that tries to minimize the differences with the similarities in the many dimension space††.

We can now answer question 1: the best sets of software characteristics are also the ones that best discriminate the two classes in term of similarity. Indeed, both classes overlap for milestone and rand, but are clearly separated for other sets of software characteristics, especially proposal2 and assembly. It is interesting to notice that even though milestone exhibits some clusters, those are unrelated to the classes. This means that it stores some information about the programs, but this information is unrelated to the problem of deciding if one should force vectorization. The t-SNE visualization further answers to question 2: we are likely to obtain similar prediction trends regardless of the machine learning algorithm that we use. Indeed the set of software characteristics should at least be able to discriminate between classes. If not, no machine learning algorithm would be able to re-create this information. This is an intrinsic property of the set of software characteristics, unrelated to the machine learning method.

†A synonym for dimension reduction is embedding.

††Please refer to the article for the definitions of similarity. In non-mathematical terms, it uses the Euclidian distance scaled by a Gaussian and a t-Student distribution in respectively the many-dimension and two-dimension space.
8. Conclusion

Intel Compiler tends to be too conservative and often fails to vectorize programs even when it would have been profitable to do so. A user can accelerate programs by forcing vectorization when it is profitable. He should however avoid the option in the opposite situation (6.17% of our data set). This situation happens for 64.18% of our dataset, made of 136500 tensor contraction kernels. In this article, we achieve 93.2% of accuracy in determining both situations leading to up to 3-times speedup (69% median speedup for a relevant sub-set of our data). With our technique, the training phase is rather time-consuming, around 20 minutes. The prediction time is under a second: this makes this technique non disruptive, that is, realistic to implement within the compilation flow. Indeed, training would have to be performed only once, before delivering the compiler. All of our data is openly available for reproduction at http://www.bodic.org.

The key of our successful predictions mainly lies in the quality of our set of software characteristics (SC). In particular, it successfully expresses the memory access pattern where other works have failed to do so [2], [4], [6]. It is however specific to tensor contraction kernels. Still, these often appear in high-performance-computing workloads. Therefore we believe our technique may have a significant impact in real-word situations.

There remains however the question of how to apply our technique to other categories of programs. Despite being researched for several decades, we have yet to see in the literature any performant general-purpose SC. We therefore believe that it is more pragmatic to research a taxonomy of kernels so that we can find performant SC for each category. Then, assuming that we can programmatical recognize in which category falls the sub-parts of the programs to compile, it would be possible to use a dispatch system in order to apply the relevant SC case by case. This will constitute the focus of our future research. We believe that the visualization technique that we introduce at the end of this article will be a valuable tool toward this goal.

References


Antoine Trouvé received his master degree in 2006 from the Computer Department of ENSEIRB-MATMECA, Bordeaux (France). He later received his Dr. Eng. degree in 2011, from the Department of Advanced Information Technology of the Faculty of Information Science and Electrical Engineering of Kyushu University, Fukuoka (Japan). He has worked as a researcher at the Institute of Systems, Information Technologies and Nanotechnologies, Fukuoka (Japan) from 2007 to 2014. He is now assistant professor at the Department of Advanced Information Technology of the Faculty of Information Science and Electrical Engineering of Kyushu University.
Arnaldo J. Cruz is a PhD candidate at the Department of Advanced Information Technology of the Faculty of Information Science and Electrical Engineering of Kyushu University, Fukuoka (Japan). His research interests include embedded systems, high-performance computing, optimizing compilers and workload characterization.

Kazuaki J. Murakami received the B.E., M.E., and Ph.D. degrees in computer science and engineering from Kyoto University in 1982, 1984, and 1994, respectively. From 1984 to 1987, he worked for the Fujitsu Limited, where he was a Computer Architect of the mainframe computers. In 1987, he joined the Department of Information Systems of Kyushu University, Japan. He is currently a Professor of the Department of Informatics, and also the Director of the Computing and Communications Center. He is a member of the ACM, the IEEE, the IEEE Computer Society, the IPSJ, and the JSIAM.

Masaki Arai received the B.S. degree in information engineering from Tokyo University of Technology in 1990 and the M.S. degree in information science from Tokyo University of Science in 1992. In 1992, He joined Fujitsu Laboratories Ltd. His research interests are in the area of compiler optimizations and computer architectures.

Tadashi Nakahira is research manager at the systems laboratories of Fujitsu Laboratory Limited. His research interests include compiler optimizations and high-performance computing.

Eiji Yamanaka is director at the next-generation-technical-computing unit of the software-development division at Fujitsu Limited. His research interests include compiler optimizations and high-performance computing.