A Light-Weight Rollback Mechanism for Testing Kernel Variants in Auto-Tuning

Shoichi HIRASAWA†,‡, Nonmember, Hiroyuki TAKIZAWA‡,§, and Hiroaki KOBAYASHI†, Members

SUMMARY Automatic performance tuning of a practical application could be time-consuming and sometimes infeasible, because it often needs to evaluate the performances of a large number of code variants to find the best one. In this paper, hence, a light-weight rollback mechanism is proposed to evaluate each of code variants at a low cost. In the proposed mechanism, once one code variant of a target code block is executed, the execution state is rolled back to the previous state of not yet executing the block so as to repeatedly execute only the block to find the best code variant. It also has a feature of terminating a code variant whose execution time is longer than the shortest execution time so far. As a result, it can prevent executing the whole application many times and thus reduces the timing overhead of an auto-tuning process required for finding the best code variant.

key words: auto-tuning, rollback, cache, early termination

1. Introduction

Thanks to advances in the computational science, a practical simulation code is becoming more complicated, and usually composed of an initialization routine and multiple computation kernels, each of which consumes a considerable part of the execution time. In such a case, all of the kernels need to be optimized for a specific platform in order to fully exploit the potential of the platform and thereby to reduce the total execution time. As a result, more effort and time are required to optimize such a practical application.

In this work, empirical performance tuning is defined as a process of manually or automatically generating multiple variants of a target computation kernel and then selecting the best kernel variant among them based on performance evaluation. Since a modern HPC computing platform has already become too complicated to accurately predict its performance for a given code, it is difficult for a compiler to judge if a code optimization improves the performance or not. Therefore, empirical performance tuning by evaluating the actual performance of each kernel variant on the target platform is inevitable.

As manual tuning is tedious and time-consuming to generate and evaluate multiple kernel variants, automatic empirical performance tuning, so-called auto-tuning, has been intensively studied for various libraries and applications, including BLAS libraries [1], FFT libraries [2], [3], and stencil applications [4], [5]. Since auto-tuning needs to evaluate the performance of every promising kernel variant to select the best one, the tuning time, which is the execution time to evaluate kernel variants, could be unacceptably long when there exists a large number of kernel variants to be evaluated [6]. Although several approaches such as in [7] have been proposed to reduce the number of kernel variants to be evaluated, the tuning time could be still unacceptably long when it takes a long time to evaluate the performance of each kernel variant by executing the application from the beginning.

There are two problems that make the expected evaluation time of each kernel variant long. One is that performance evaluation of each variant of a target kernel usually needs to execute the initialization routine and preceding kernels so that the target kernel is correctly executed. Thus, if the initialization routines or other kernels are time-consuming, it takes a long time to obtain the performance of each kernel variant. The other problem is that there could exist a kernel variant whose performance is extremely low. For example, the performance would become very low if a kernel variant suffers from thrashings of memory, disks, and other resources. Although such a kernel variant is never selected as the best one, the other kernel variants cannot be evaluated as long as the slow variant is being evaluated. Therefore, the existence of a slow kernel variant could remarkably affect the overall tuning time.

Auto-tuning with a checkpoint/restart system [8] has been developed to reduce the tuning time for a large-scale application. As the checkpoint-restart system simply saves and restores the whole memory image of an application process, however, the timing and storage overheads might not be negligible in practice. Furthermore, it does not address the latter of the above two problems because it still executes a very slow kernel variant as well as other promising kernel variants until the end of its execution.

This paper proposes a light-weight rollback mechanism to reduce the timing and storage overheads for evaluating each variant of a target kernel, thereby reducing the overall tuning time of an application. One important feature of the proposed mechanism is to minimize the memory contents to be saved and restored, based on data dependency analysis. The mechanism saves the minimum memory contents once, and repeats the sequence of executing one kernel variant to evaluate its performance and recovering the saved context to rollback to the original memory state. By evaluating the
performance of each kernel variant without executing initialization routines and other kernels, the proposed mechanism reduces the time for obtaining the performance of each kernel variant, and thereby the overall tuning time.

Another important feature of the proposed mechanism, called Early Termination, is to terminate the execution of a kernel variant when the kernel variant is obviously slower than the so-far best one. This feature is significantly effective to reduce the total time of evaluating kernel variants as discussed later.

For minimizing the data size to be saved and restored, the proposed mechanism keeps using only one process for evaluating the performance of every kernel variant one by one. One concern about this approach is that the cache behaviors could affect kernel variant selection. It is obvious that only the firstly-executed kernel variant causes compulsory cache misses if the whole data accessed by kernel can be stored in the cache; the other variants can reuse the cached data fetched by the first variant. Thus, the performance of the first kernel variant might be underestimated. For discussing such cache effects, the proposed mechanism has an optional feature that initializes the cache state. The basic idea of this feature is to evict all cached data from cache memory by accessing a dummy array before every kernel execution. As a result, it inhibits kernel variant selection from depending on the initial cache state at each kernel execution. The effect of this feature will also be discussed through the evaluation results.

This paper is organized as follows. The problem on timing overheads in auto-tuning is reviewed in Sect. 2. A light-weight rollback mechanism is proposed in Sect. 3 and its effectiveness is evaluated in Sect. 4. Conclusions and future work are described in Sect. 5.

2. Performance Tuning Process and Timing Overheads of Auto-Tuning

A large-scale HPC application usually consists of an initialization routine and multiple kernel routines. As it is difficult to tune the whole application at once to improve the overall execution performance, one of the kernels is first selected as a target and tuned to improve its execution performance. Figure 1 shows a general execution flow of a large-scale HPC application, which consists of one initialization routine and multiple computation kernels. In the figure, one of the computation kernels is selected as a target kernel for tuning.

When the target kernel is tuned, the overall execution of the whole application is profiled again. If the overall performance is acceptable to the application developer, the tuning process is terminated. On the other hand, if the performance is not acceptable, one of the remaining kernels is selected and tuned to further improve the overall execution performance.

As manual tuning is tedious and error-prone, auto-tuning that automatically generates code variants of a kernel and empirically selects the best one, has lately been attracting attention. To select the best variant of a target kernel, auto-tuning needs to execute a lot of kernel variants to collect their profiling results [9].

Suppose that a specific kernel of a large-scale HPC application is selected as a tuning target. Then, if \( N \) variants of the target kernel need to be executed, the total execution time for the tuning process will be, at shortest, \( N \) times longer than that of the application with the best variant, as shown in Fig. 2.

One problem in finding the best kernel variant is that the total tuning time could easily be unacceptable if the execution time of an application itself is long and/or if there exist quite a huge number of kernel variants.

Another problem is that we have to wait for the execution completion of long running kernel variants even if a kernel variant of a shorter execution time has already been found. As illustrated in Fig. 2, we need to execute Variant 2 to the end, even though Variant 1 is known to have a shorter execution time.

To address the former problem, one idea is to execute all \( N \) variants of the target kernel one by one while skipping the initialization routine and other kernels being executed before the target kernel. The ROSE compiler infrastructure has a mechanism for auto-tuning with the support of the BLCR checkpoint/restart system [10]. In the ROSE auto-tuning framework, the target kernel is a time-consuming part of an application, which is usually a loop nest. It is
Fig. 3 The ROSE auto-tuning system with the support of the BLCR checkpoint/restart framework.

Fig. 4 A Light-weight Rollback Mechanism.

separated from the application code to be a discrete function. This is called function outlining. The outlined function is then transformed to multiple variants with source transformation frameworks such as CHiLL [11], POET [12] and built-in transformation tools of ROSE itself. After the code transformation, the generated variants are executed to collect their profiling data to choose one variant from them.

BLCR is used when the auto-tuning framework executes the multiple variants. Before the first variant is executed, the auto-tuning framework takes a snapshot or a checkpoint of the running process, so that the other variants can be tested by restarting from the checkpoint. As a result, it can skip the execution of initialization and other kernels, resulting in a reduction in the tuning time. With this execution process, the mechanism can reduce the timing overheads by checkpointing and restarting the outlined kernel in the original application.

Figure 3 shows the execution framework of the ROSE auto-tuning system with the support of the BLCR checkpoint/restart framework (ROSE+BLCR). As the system saves the whole memory image of the application process into a checkpoint file and restores the memory image to execute another kernel variant, the storage overhead is not negligible. Nonetheless, minimizing the storage overhead has hardly been discussed in the auto-tuning research.

Although modern HPC systems are often equipped with fast storage devices such as SSD and RAM disks, a large checkpoint cannot be saved in such a fast but relatively small storage. Furthermore, the ROSE+BLCR system does not address the problem that a long running variant remarkably extends the overall tuning time.

3. A Light-Weight Rollback Mechanism to Reduce Overheads of Auto-Tuning

3.1 A Light-Weight Rollback Mechanism

In this paper, we propose a light-weight rollback mechanism (LRM). An overview of the light-weight rollback mechanism is shown in Fig. 4.

As discussed in the previous section, there are two problems that could make auto-tuning impractically time-consuming. One is that the checkpoint-restart approach saves and restores the whole memory image of an application process, and results in large timing and storage overheads. The other is that evaluation of a long-running kernel variant extends the overall tuning time. To solve those two problems, the proposed mechanism has two key features: (1) saves and restores the minimum execution state necessary for executing variants of a target kernel one by one, and (2) terminates the execution of a variant when its execution time exceeds that of the minimum execution time obtained so far.

In addition, the proposed mechanism has an optional feature that initializes cache memory to assure fair comparison among kernel variants. Since cache memory is transparent to the mechanism, the mechanism cannot recover the cache state at the rollback. As a result, if a kernel variant stores data to the cache memory as a consequence of a cache miss, its subsequent variant might be able to use the cached data without any cache miss. One concern is that the cache behaviors may lead to unfair performance comparison between those two variants. Therefore, the optional feature evicts all the cached data at every rollback so that the cache memory is filled up with irrelevant data in the beginning of every kernel execution. Although this feature obviously increases compulsory cache misses, the cache miss penalty is evenly given to all the kernel variants, and therefore their performances are fairly compared.

Using the above three features, all of $N$ kernel variants are evaluated one by one without executing their preceding initialization routine and other kernels. LRM transforms the target kernel into a loop that runs all kernel variants. The procedure of the code transformation is summarized as follows.

**Step 1.** Transform a target code region to a discrete function, called a kernel function. This transformation is so-called function outlining [10].

**Step 2.** Analyze each argument of the kernel function to find out the data set, $\text{SavedVars}$, which is to be saved for rollback recovery. Let $\text{LiveInVars}$ and $\text{LiveOutVars}$ be sets of input arguments and output arguments, respectively. Then, the data set is defined by $\text{SavedVars} = \text{LiveInVars} \cap \text{LiveOutVars}$.

**Step 3.** Duplicate the kernel function, and then optimize the duplicated functions in different ways to generate kernel
**Table 1** Data to be saved and restored by LRM and ROSE+BLCR.

<table>
<thead>
<tr>
<th></th>
<th>ROSE+BLCR</th>
<th>LRM</th>
</tr>
</thead>
<tbody>
<tr>
<td>In &amp; 'Out</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>In &amp; Out</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>In &amp; 'Out</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Other</td>
<td>x</td>
<td>-</td>
</tr>
</tbody>
</table>

**Step 4.** Replace the original target region with a loop, called a *profiling loop*, in which one of the kernel variants is called at each iteration.

**Step 5.** Insert a code of saving *SavedVars* before the profiling loop.

**Step 6.** Insert a code of restoring *SavedVars* after the kernel function call.

Each feature is further described below.

### 3.2 A Feature to Save and Restore Only Necessary Data

The proposed mechanism, LRM, saves and restores only necessary data for restoring the previous context changed by executing the target kernel.

In general, a kernel reads a set of input data, and writes a set of output data. If some of the input data are updated during the execution, their original input data must be recovered to ensure that all the input data of all the subsequent variant are the same as those of the current variant. In other words, if input data are not updated during the execution, those input data can be simply reusable when executing the subsequent variant. On the other hand, by definition, output data are always overwritten during kernel execution before using their initial values. The initial values do not affect the kernel behavior, and thus can be different from those of another kernel execution.

As summarized in Table 1, only the input data that are updated during kernel execution need to be saved and restored for rollback recovery; the other data do not need to be saved. Therefore, LRM simply reuses the other data during the execution, unlike ROSE+BLCR that saves and restores the whole memory image of a process.

LRM has a static code analysis feature to figure out the input data being updated during kernel execution. Suppose that a code region being tuned, i.e. a target kernel, is specified by a user and given to LRM. Then, using the function outlining [10], the target kernel is outlined to a kernel function. As a result, all input data and output data of a kernel are passed to the outlined kernel function as arguments. By checking if the value of each argument is used and then updated, LRM finds the data that need to be saved for rollback recovery. Basically, if the value of an argument is used and then appears as the left-hand operand of an assign operator, the argument is considered the input data being updated during the kernel execution.

For example, in Fig. 5, array *a* is updated using array *a* itself and arrays *b* and *c*. In this case, LRM stores only the data of array *a*, while ROSE+BLCR stores all the arrays *a*, *b*, and *c*. After the target code is outlined as a function (*kernel_function*), which is shown in Fig. 6, the necessary data, which is *a* in the example, is obtained as the product set of the read set (*a*, *b*, *c*, *n1*, *n2*, and *n3*) and the write set (*a*).

### Function Definition

```c
function kernel_function (a, b, c, n1, n2, n3) // call by reference for function arguments
// k, i and j are local variables
do k = 1,n1
do i= 1,n3
    do j = 1,n2
        a(i,k) = c(i,k) + a(i,j) * b(j,k)
    enddo
enddo
enddo
```

Fig. 5 Sample code of selecting necessary data for rollback recovery in LRM.

### Function Definition

```c
function currentVariant;
storeSavedData();
// profiling loop
while (currentVariant = getNextVariant())
{
    startVariant(currentVariant, arguments..);
    restoreSavedData();
}
startVariant(function variant, Vars arguments..)
{
    variant(arguments..);
}
```

Fig. 6 Sample result code for function outlining.

Data, which is *a* in the example, is obtained as the product set of the read set (*a*, *b*, *c*, *n1*, *n2*, and *n3*) and the write set (*a*).

After the code analysis, the kernel function is duplicated and optimized in different ways to generate kernel variants. A profiling loop to run all variants of *kernel_function* is generated. The pseudo code of a profiling loop generated by LRM is shown in Fig. 7. In the figure, *storeSavedVars*() and *restoreSavedVars*() are function calls to save data and restore data, respectively. A profiling loop runs one of the remaining kernel variants at each iteration by performing rollback recovery after evaluating a kernel variant. As a result, all variants are evaluated so as to select the variant with the highest performance among them.
Since the feature only saves some of the input data of the target kernel, the storage overhead is always less than ROSE+BLCR, which stores not only all input and output data but also data that are unrelated to the target kernel. As a result, the timing and storage overheads of LRM are expected to be smaller than those of ROSE+BLCR.

3.3 A Feature to Terminate Long Running Kernel Variants

LRM has a feature to terminate a long running variant of a target kernel. When the execution time of a kernel variant is clearly longer than the minimum execution time obtained so far, the execution is terminated.

For example, although Variant 2 in Fig. 4 does not reach the end of the execution, the execution is terminated earlier because its execution time exceeds that of Variant 1 that is the best variant so far. To check if the execution time is sufficiently long, the termination is decided using a predefined threshold given by the user.

The pseudo code of LRM with Early Termination for running a kernel variant is shown in Fig. 8. In LRM with Early Termination, a main thread executes a while loop that starts a thread to execute a variant of the target kernel. Then, the main thread waits for completion of the execution of the variant until the elapsed time exceeds the minimum execution time obtained so far plus a threshold specified by users. When the time has elapsed and the variant is still running, the main thread terminates the execution of the variant.

3.4 A Feature to Initialize Cache Memory for Fair Comparison among the Target Kernel Variants

ROSE+BLCR creates one process for profiling one kernel variant, and terminates it after executing the kernel. Then, it runs another process by restarting from a checkpoint file to profile another kernel variant. In this way, ROSE+BLCR uses multiple processes for profiling kernel variants. On the other hand, LRM uses only one process to profile all kernel variants in order to minimize the data to be accessed for rollback recovery and also to avoid the overhead of creating a new process for every kernel variant.

One concern about LRM’s approach is that the cache behaviors potentially affect the performance comparison results. In the case of ROSE+BLCR, all the cached data would usually be evicted before executing the next kernel variant because certain amounts of data are accessed to launch a new process and restart from a checkpoint file. On the other hand, in LRM, one process repeats running kernel variants that would access the same data, and hence the data cached by executing one kernel variant might be reused by its subsequent kernel variant. As a result, there is a risk that the performance of the subsequent kernel variant is overestimated.

LRM reduces the risk by achieving the same performance comparison results as ROSE+BLCR. Specifically, LRM internally has a memory chunk whose data are never accessed by the target application being tuned; the data are unrelated to any kernel variants. The memory chunk size is identical to the last-level cache size of the target platform. To ensure that all kernel variants can be executed with the same cache state, the cache data are initialized by accessing the unrelated data at a rollback recovery process. Although this does not recover the original cache state, it can avoid unfair comparison among kernel variants because every kernel variant starts with the same cache state, in which the cache does not store any data being accessed by the kernel. The effects and overheads of this cache initialization feature are discussed in Sect. 4.

4. Implementation and Evaluation of the Light-Weight Rollback Mechanism

This section shows a reduction in timing overheads required for evaluating kernel variants. The tuning time of each kernel variant and the total tuning time to find the highest-performance variant are measured for discussions.

4.1 Implementation of the Light-Weight Rollback Mechanism

In this paper, all the features of LRM are implemented on top of the ROSE compiler framework version 0.9.5a-20584. In the following evaluation, three versions of LRM are used
to discuss the effects of each feature and the overall tuning time.

**LRM:**
A buffer array is internally used to store all necessary data, which are passed to an outlined kernel function and updated by the function.

**LRM-Cache:**
After restoring the saved data to start executing another kernel variant, a dummy array, which is defined out of any input and output data of the target kernel, is sequentially accessed to evict the cache data. LRM-Cache denotes LRM with the cache initialization feature.

**LRM-ET:**
A kernel function is executed by creating a new thread. The main thread basically waits for the new thread to complete its execution. However, if the thread execution time is obviously longer than the shortest execution time obtained so far plus a certain threshold, the main thread immediately terminates the executing thread. The threshold is set to 20 ms in the following evaluation. LRM-ET means LRM-Cache with the early termination feature.

### 4.2 Evaluation Setup and Results

In this paper, two applications are used in the following evaluation.

The first application is a 2-dimensional fluid-flow simulation based on the fractional step method [13]. Most execution time of the simulation is spent for one Jacobi kernel solving a Poisson equation for pressure calculation. To generate kernel variants, the innermost loop of the kernel, which consists of 24 code lines, is unrolled $N$ times where $N$ is changed from 1 to 23. Moreover, the innermost loop and the second innermost loop are also interchanged, and the new innermost loop is unrolled $N$ times.

The other application is a real-world application, called Numerical Turbine, that simulates two- and three-dimensional multi-stage stator-rotor cascade flows in gas and steam turbines [14]. As with other practical simulation codes, Numerical Turbine has a lot of computationally-expensive loop nests and hence all of the loop nests have to be tuned for a target platform to reduce the total simulation time. In the evaluation, the most time-consuming loop nest, which consists of 239 code lines, is selected as a target kernel, and optimized in different ways. In the simulation code, the loop nests are offloaded to a GPU using OpenACC directives [15]. The simulation code has three OpenACC execution parameters of seq, gang and vector that need to be tuned appropriately for individual platforms. In the evaluation, the simulation code is executed on a GPU platform with different OpenACC execution parameters.

The system used in the evaluation has Intel Xeon CPU E5-2670, which has a 20MB last level cache (LLC), running at 2.60 GHz, 64 GB memory, NVIDIA Tesla K20 GPU, and SATA HDD. The system is operated by Linux 2.6.32. ROSE version 0.9.5a-20584 and g++ 4.4.7 with the -O3 option are used for code transformation and compilation, respectively. PGI Accelerator Compiler 14.03 is also used to compile an application code for the GPU.

### 4.3 Evaluation with the 2-Dimensional Jacobi Kernel

Figure 9 shows the tuning time of each kernel variant of the 2-dimensional fluid-flow simulation. In the figure, Org represents the tuning time for each variant when the performance of each variant is evaluated by running the application from the beginning. BLCR, LRM, and LRM-Cache represent the tuning times of ROSE+BLCR, LRM, and LRM-Cache respectively.

Although the tuning time of BLCR is shorter than that of Org, the tuning time of LRM is even shorter than that of BLCR. This difference comes from that LRM saves and restores only necessary data for continuously evaluating variants one by one, while BLCR always saves and restores the whole memory image of the process. For rollback recovery, BLCR always reads a checkpoint file of approximately 72MB, while the data size read by LRM is only 16MB. Because LRM automatically handles the data savings and restorings, users do not need to concern them. Accordingly, Fig. 9 clearly indicates that the timing overhead of rollback recovery can be reduced by selectively storing and restoring only necessary data.

Figure 9 also shows that LRM-Cache achieves a comparable performance to LRM. The differences between LRM and LRM-Cache are measurement errors rather than the overhead of LRM-Cache. This means that the overheads of cache initialization and additional compulsory cache misses induced by LRM-Cache hardly affect the total execution time and are negligible in the case of this kernel. This is because the average execution time of kernel variants are much longer than the overheads.

Figure 10 shows the accumulated tuning time in the case of incrementally evaluating kernel variants shown in Fig. 9 from left to right. From the figure, we can see that the accumulated tuning time of LRM-ET is always shorter than those of the others. This is because LRM-ET can successfully terminate the execution of a kernel variant whose execution time already exceeds the minimum one obtained so far. Note that the growth rate of the accumulated tuning time changes depending on the order of kernel variants being evaluated. The accumulated tuning time can be expressed by

$$T_{acc} = \sum_{i=1}^{N} \min[t_j, j = 1, \ldots, i],$$

where $N$ is the number of kernel variants and $t_j$ is the execution time for evaluating the $j$-th kernel variant, i.e. the tuning time for the $j$-th kernel variant. It is obvious that
$T_{\text{acc}}$ becomes minimum, $N_{t_1}$, if $t_1$ is the minimum, $t_1 = \min[t_j] j = 1, \ldots, N$. In the worst case where kernel variants are evaluated in descending order of execution time, the accumulated tuning time will theoretically be the same as LRM-Cache, even though a trivial runtime overhead might be seen in practice. Accordingly, even if a kernel variant is randomly selected for evaluation, it is expected that LRM-ET always works faster than the others. If an intelligent selection method is available to evaluate a more promising variant earlier, the superiority of LRM-ET in accumulated tuning time becomes further remarkable.

### 4.4 Evaluation with Numerical Turbine

Using a real-world application code, Numerical Turbine, the effects of LRM and its features on the tuning time are evaluated. In this evaluation, the most time-consuming loop nest is selected as the target kernel for tuning. The loop nest consists of four loops, K, J, I, and L. The innermost two loops I and L are selected to find the best OpenACC parameters of seq, gang, and vector. As each loop has three parameter values, nine variants for the two loops in total are evaluated.

Figures 11 shows the tuning time of each variant of the kernel in Numerical Turbine. As with Figs. 9 and 10, Org represents the tuning time when the performance of each variant is evaluated by running the application from the beginning. BLCR, LRM, and LRM-Cache represent the tuning times of ROSE+BLCR, LRM, and LRM-Cache, respectively.

Since Numerical Turbine consists of multiple kernels, Org executes not only the target kernel but also the initialization routines and other preceding kernels whenever the performance of a kernel variant is evaluated; in Org, the application is executed from the beginning many times. As a result the tuning time of Org is 11.1x longer than that of BLCR on average. Therefore, especially in the case of autotuning a practical application, it is important to execute only the target kernel by using a rollback recovery mechanism.

For rollback recovery, BLCR always reads a checkpoint file of approximately 552MB, as shown in Fig. 12. Numerical Turbine is executed with a relatively small data set in this evaluation, and the checkpoint file size could be even larger if it is executed with a larger data set. On the other hand, the data size read by LRM is only 193MB, because LRM selectively saves only the data to be accessed and then updated during the kernel execution. As a result, LRM can achieve rollback recovery at a much lower cost than BLCR.

Figure 11 also shows that LRM-Cache achieves a comparable performance to LRM as in the case of the 2-dimensional fluid-flow simulation. The difference in execution time between LRM and LRM-Cache comes from overheads of initializing the LLC by LRM-Cache and compul-
sory cache misses caused by the target kernel accessing necessary data. Note that the overhead of initializing the LLC is unchanged during the execution because it only depends on the size of LLC. The overhead of compulsory cache misses will not surpass the memory accessing time to the size of the LLC. This results in a constant overhead for all kernel variants, and the same variant is always selected as the best one by ignoring performance changes depending on the initial cache state of each kernel execution. From the figure, we can see that both LRM and LRM-Cache can successfully select Variant 5, which is the variant with minimum execution time.

Although the overheads might be visible in the case where the average execution time of kernel variants is comparable to the overheads, it is not worthful to optimize such a low-cost kernel. Generally, a kernel is the most time-consuming part of an application code, and its execution time is much longer than the overheads caused by cache initialization and compulsory cache misses. Therefore, the overheads would be acceptable in practical uses. Although LRM provides the cache initialization as an optional feature for discussions on the cache effects, it would not be required for performance tuning of a practical kernel in most cases.

As well as Fig. 10, Fig. 13 shows the superiority of LRM-ET in the accumulated tuning time. As shown in Fig. 11, LRM-Cache is slower than LRM in the case of Numerical Turbine because the execution time of this kernel is likely to be shorter than that of the Jacobi kernel. In this particular case, Early Termination does not work when Variant 2 is evaluated, because Variant 2 needs a shorter tuning time than Variant 1. We can also see that the overheads of cache initialization and compulsory cache misses increase the accumulated tuning time of LRM-Cache and LRM-ET, i.e. LRM-Cache with Early Termination, compared to that of LRM. However, as Variant 4 is obviously slower than Variants 2, which is the fastest one at that time, its execution is terminated before the completion on LRM-ET. As a result, the accumulated tuning time of LRM-ET is shorter than that of LRM since Variant 4 is evaluated. Although the accumulated tuning time of LRM-ET strongly depends on the order of kernel execution as discussed using Eq. (1), the executed time is 1.5x shorter than that of LRM-Cache in this particular case. The expected accumulated tuning time of LRM-ET among all orders can be calculated by considering all sequences of executing all kernel variants. The expected accumulated tuning time of LRM-ET among all orders is 1.2x shorter than that of this case. Accordingly, it is clearly demonstrated that LRM-ET is significantly effective to reduce the tuning time in practical uses.

5. Conclusions

In this paper, a light-weight rollback mechanism has been proposed. The proposed mechanism can execute only the variants of a kernel one by one, so as to reduce the overall tuning time of a large-scale application, which is composed of initialization routines and multiple kernels.

The proposed mechanism saves and restores only necessary data to reduce the timing and storage overheads, compared to the existing approach. It can also terminate the execution of long running kernel variants so as to reduce the total execution time required for evaluating all the variants. In addition, it provides an optional feature to evict the cached data for discussions on the risk of selecting a wrong kernel variant due to the cached data.

The evaluation results demonstrate that the proposed mechanism can successfully reduce the tuning time thanks to these features. The results also show that the overheads of initializing the cache data and additional compulsory cache misses for the execution of each kernel variant are negligible. In addition, the results show that the order of evaluating the variants is an important factor to decide the overall tuning time if the early termination feature is available.

The performance penalty of compulsory misses is negligible because the kernel execution time itself is usually much longer than the penalty. Therefore, it is also clarified that the cache initialization is usually unnecessary for tuning a practical kernel.

In this work, users need to take on the responsibility for the correctness of the execution of each kernel variant. Automatic correctness checking for the execution results of all kernel variants during the whole tuning process will be discussed in the future work.
Acknowledgments
This work is partially supported by JST CREST “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems” and Grant-in-Aid for Scientific Research(B) #25280041.

The authors would like to thank Prof. Yamamoto and Prof. Sano of Tohoku University for allowing us to use their applications. The authors also appreciate Prof. Katagiri and Prof. Suda of The University of Tokyo for their meaning-full suggestion about the cache behavior on auto-tuning.

References


Shoichi Hirasawa is currently a researcher of Graduate School of Information Sciences, Tohoku University. His research interests include high-performance computing systems, programming languages processing, and their supporting tools. He received the B.S. Degree in School of Science of The University of Tokyo, and the M.S. Degree in Graduate School of Information Science and Technology of The University of Tokyo in 2000 and 2002 respectively. He is a member of IEEE CS, ACM and IPSJ.

Hiroyuki Takizawa is currently an associate professor of Graduate School of Information Sciences, Tohoku University. His research interests include high-performance computing systems and their applications. He received the B.E. Degree in Mechanical Engineering, and the M.S. and Ph.D. Degrees in Information Sciences from Tohoku University in 1995, 1997 and 1999, respectively. He is a member of the IEEE CS, and the IPSJ.

Hiroaki Kobayashi is currently Director and Professor of Cyberscience Center and Professor of the Graduate School of Information Sciences, Tohoku University. His research interests include high-performance computer architectures, grid and P2P computing, and multimedia applications. He received the B.E. Degree in Communication Engineering, and the M.E. and D.E. Degrees in Information Engineering from Tohoku University. He is a senior member of IEEE CS, and a member of ACM and IPSJ.