Speculative Parallel Execution on JVM

Kazuki YOSHIZOE, Takashi MATSUMOTO and Kei HIRAKI

July 31, 1998

Abstract

To improve the performance of JVM, we propose to exploit coarse grain parallelism. Java bytecode's machine independence requires to perform any parallelization dynamically. In runtime parallelization, the analysis of the program must be done in a short time. In this paper we propose to apply the idea of "Run–time restructuring" to JVM. This proposal is base on speculative parallel execution of loops based on dynamic analysis of existing program. We implemented an experimental interpreter JVM which performs speculative execution. With simple information about the target loop, our JVM execute iterations of the loop in parallel. In the preliminary experiment, speedups for loop kernels were observed.

1 Introduction

There have been researches on techniques to improve performance of JVM. It is widely known that the JIT compiling is playing a main role in improving the performance of sequential machine based JVM. But recent demands upon high performance computing on Java encourages to search for a further improvement. The best considered candidate method to improve performance of JVM is parallel execution. There have been studies about exploiting instruction level parallelism for Java such as [1] and [2]. In research [1], a runtime re–compilation for VLIW is described. [2] is an application of [1] for Java. JIT compiling is also exploiting instruction level parallelism in combination with superscaler CPU. Here we discuss dynamic parallelization on parallel machine based JVM. Because of the machine independence of Java bytecode, the parallelization must be performed dynamically to make it suitable for the platform

which the JVM is based on.

To search for a new possibility of JVM optimization, we propose here exploiting a multiple block level parallelism for Java. Here multiple block means a part of a program which is separated by branches. Multiple block is a group of basic blocks which may contain method calls and branches. The size of the multiple block varies from thread level size to an basic block level size.

There have been many studies on exploiting parallelism in higher level than instruction level parallelism such as multiscalar processor, superthreading architecture, and hydra project ([3, 4, 5]). These researches are for ordinary binary program and the mechanisms are proposed to work on on–chip multi processors. The aim of this paper is to investigate the possibility of pure software implementation of a JVM for multiple block level parallelism.

In hardware based approach, the parallelization is not necessary to be performed at runtime. In examples [3, 4, 5] the parallelization is performed by a combination of compiler and hardware. In static analysis there is enough time to perform precise analysis of the program, such as detecting data dependencies. But because of the machine independence of Java bytecode, JVM must perform parallelization at runtime. It is possible to include an additional attribute in the class file to assist parallelization, but at least the bytecode must be the same on distribution. Therefore we can and must select an appropriate parallelization for the machine environment.

Another approach is dynamic parallelization by "Run–time restructuring" of the program. Run–time restructuring is proposed in [6]. The proposal is to perform dynamic parallelization by analyzing loops at execution time. The runtime parallelization causes an overhead but on the other hand parallelization is possible while maintaining binary compatibility.

To execute consecutive blocks in parallel, control flow prediction and data dependency management must be considered. Control flow prediction could

be handled in the same manner as conventional branch prediction. (Of course more precise prediction is possible.) Data dependency management is another problem. This problem is specific in multiple block parallel execution. When the value of a register or memory is modified in a block and the same value is used in the succeeding block, there is a data dependency. It must be checked to avoid data dependency violation (read after write hazard). In hardware based approaches, data dependency violation through registers and through memory accesses are handled in different manner because the hardware required for these two types of data dependencies are totally different. The value of a register would be passed to the succeeding blocks to avoid data dependency violation. Data dependency violation through memory accesses is another problem. It would be necessary to prepare an exclusive hardware to examine memory accesses at runtime. See figure 1.

Strict management for multiple block level parallelization is difficult for software implementation, because unlike in hardware approaches, we can not rely on hardware support to manage data dependencies. Thus we introduce here an approach of speculative execution of each multiple block. Because it is inefficient to examine every instruction at runtime to detect data dependency violation, we execute each multiple block speculatively and the results are checked after the execution of the block is ended. On failure the block would be re–executed.

In the paper [6], the hardware examine the binary program dynamically and finds the loop invariant operation instructions. It then allocates each iteration of the loop to machine elements and execute them in parallel.

There are two approaches to implement block level parallelization for Java. One is to use static approach by a Java compiler or bytecode analyzer. But the static analysis approach requires the time to perform the analysis. We can include the analysis into a class file as a custom attribute (see [7]) but it is not recommended. Therefore we decided to apply the "Run–time restructuring"

onto JVM. In this approach, it is possible to keep the class file unmodified. We have done an experiment based on an interpreter JVM. The more detailed discussion is written in section 2.

We implemented an experimental JVM which performs speculative execution limited to loops. The implementation is described in section 3. We measured speedups for loop kernels. The results are written in section 4. Related work and summary is in section 5 and section 6.

2 Speculative Execution on JVM

In this section, we describe the characteristic of static anaylsis approach and dynamic analysis approach. It is possible to perform static analysis without breaking the machine independence of the bytecode. As mentioned in [7], it is permitted to include an custom attribute into class files. We can add the result of static analysis to the class file as a new attribute. Thus optimization by the combination of a custom Java compiler and a custom JVM is possible. In dynamic analysis approach it is difficult to performe sophisticated parallelization. Because the information which is necessary for parallelization must be simple enough to be possible to be obtained at runtime.

From now on we call the executing unit to allocate each block Virtual Machine Element (or VME). Each VME should have its own Java stack.

To implement block level parallelism, it is necessary to handle data dependency violation properly. The common mechanism for block level parallelization is shown in figure 1. Each VME executes one block of the program.

When the succeeding block uses the value of a stack which value was modified by the preceding block, a data dependency violation would be caused. If we analyze the program statically, we have time to examine all instructions which could possibly cause data dependency violation. We can indicate the

corresponding instructions at which the VME should forward or wait for the data. The idea is shown in figure 1. This approach could be implemented using synchronization mechanism of threads. But if we use dynamic analysis, we have no time to examine instructions. In this case, we can't start examining data dependency through stacks at runtime, because checking all instructions would cause serious overheads. Therefore we introduce the prediction of the value in the stacks. We can't forward data at runtime to dissolve data dependency but we can check if the prediction was correct or not.

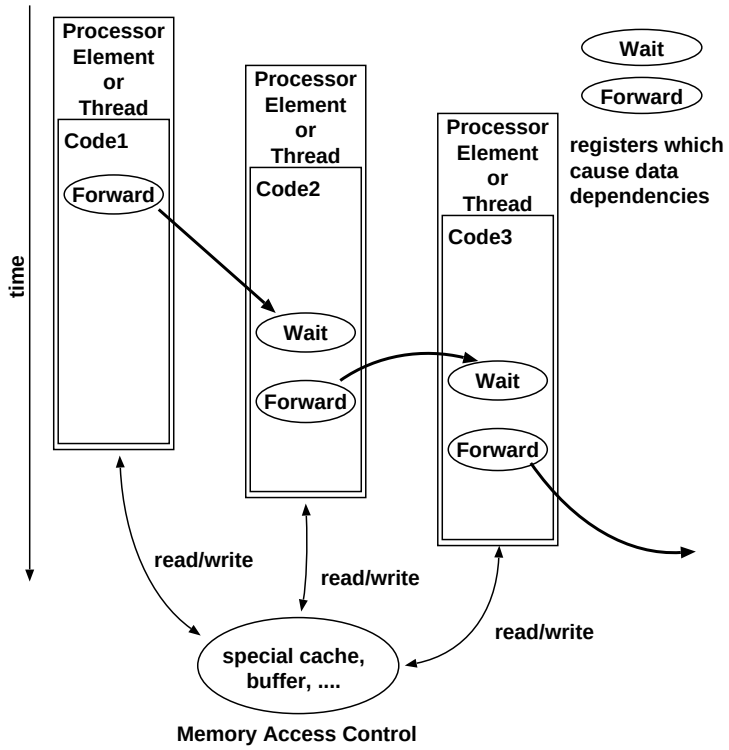

Figure 1: Outline of Static Approach.

While speculative execution, it is necessary to watch all heap access instructions, because heap accessing instructions can also cause data dependency violation. If there was a *write* to an address in the preceding block, the follow-

ing blocks should check *reads* from the same address. In the approach of static analysis of the program, it is possible to check the heap access instructions and could know which instruction would cause data dependency violation. Because Java uses different instructions to access heap area and Java stack, we don't need to see the source code to pick up only the heap access instructions. Those heap access instructions are, array access instructions and class member variable access instructions. Because the number of heap access instructions are small compared to stack access instructions, it is not difficult to indicate all potentially dangerous instructions. We may wait at a *read* for the corresponding *write* to be performed. If more precise examination is possible, we can distinguish potentially dangerous instructions and safe instructions. To detect data dependency violation, only checks for the potentially dangerous instructions are necessary. If the number of the dangerous instructions are small enough, it is possible to perform runtime data dependency resolution, by forwarding the value of the heap accesses to succeeding blocks. Meanwhile in dynamic analysis approach, it is inefficient to perform precise analysis of heap access instructions. Thus we can't distinguish only the dangerous heap accesses from safe ones. To resolve data dependency at runtime, it is necessary to check all heap accesses. Therefore it is impossible to perform parallel execution of blocks without speculation of memory accesses. This is the reason that in dynamic analysis approach speculative execution is inevitably necessary. While speculative execution, *writes* should not be performed in reality. The address and the value of the *writes* should be recorded and performed after it is confirmed that the speculation was a success. The address of the *read* instructions should also be recorded. At the end of each block, it is necessary to check the addresses of the *reads* and *writes* to detect data dependency violation. When there was a violation, the corresponding block should be re–executed. See figure 2. In the dynamic approach, heap access instructions would not be examined while execution. All blocks

are executed speculatively and heap access instructions are only checked at the end of a speculative executed block. No data would be forwarded to succeeding blocks and only checks at the end of each block are performed.

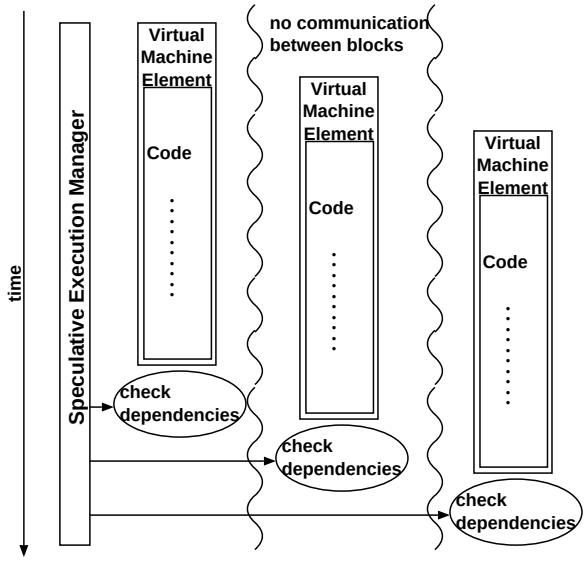

Figure 2: Speculative Execution in Dynamic Analysis Approach.

As we have discussed in this section, static analysis has an advantage at execution time, because the analysis would be more precise. The disadvantage of static analysis is that the analysis takes time. Static analysis approach is possible in a condition that we have the source code and it is compiled by a certain Java compiler, or we have the time to run a class file analyzer before the execution. Conversely dynamic analysis has disadvantage at runtime because the parallelization would be more inefficient. But in this approach, Neither specific Java compiler nor previous analysis of the bytecode is necessary.

Considering the machine independence and distribution transparency of Java and the recent research of HotSpot [8], we have decided to do an experiment on dynamic approach. Because although it is permitted to define a new attribute in

a class file, it is not recommended. In our approach, the information necessary for parallelization is simple enough and could be obtained at runtime.

3 Implementation

3.1 Outline

We implemented block level parallelization on a pure software implementation of Java Virtual Machine. In this implementation we decided to use simple analysis of the loops in existing bytecode for parallelization.

3.2 Target Loop Specification

The scheme to select the target loop is described in this section. An profiling information is prepared for the method which contains the target loop.

We expect the target loop to have the following characteristics.

- The change of the loop invariant is completely (or in high probability) predictable.
- The number of memory access in an iteration is small.
- There is no (or very infrequent) data dependency through Java stack.
- There is no (or very infrequent) data dependency through heap area.

An example for a loop is shown in figure 3. The compiled bytecode for this program is shown in figure 4.

Following informations are necessary for our JVM.

1. Loop invariant change in one iteration.
2. Program counter at the start point and end point of the loop.
3. Program counter at the start point and and end point of speculative execution.

```
public class loop {
  public static void main(String args[]) {
    int k;
    for(int i=0;i<96;i++) {
      k = 0;
      for(int j=0;j<100000;j++) {
        k++;
      }
    }
    System.out.println("good bye");                                    10
  }
}
```

Figure 3: An ideal loop example.

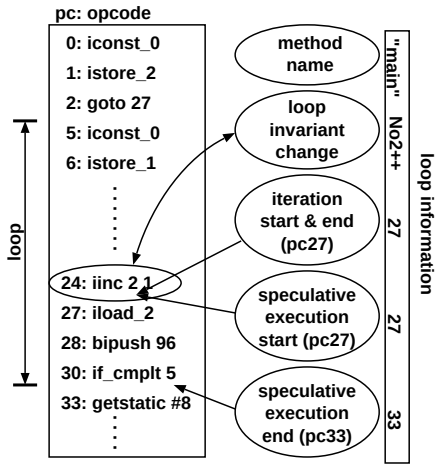

Figure 4: An example for bytecode and loop information.

These informations would be written into a separate file. (Shown in figure 4.) In this example, loop invariant is local variable No. 2. It is incremented by the instruction "`iinc 2 1`" at pc 24.

In current implementation loop detection is done manually. However, for small loops, it is not difficult to detect the change of loop invariants at compile time. (The method for detecting loop invariants is described in the textbook [9].) In future implementation, we should perform loop detection by JVM at runtime.

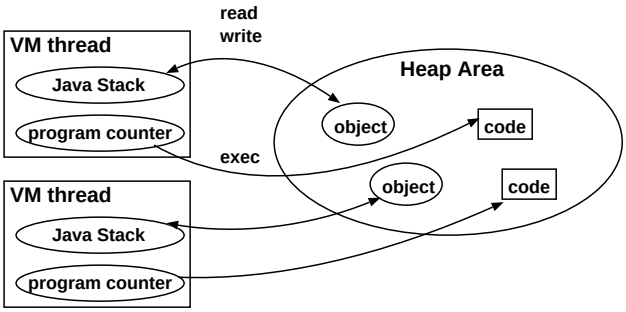

Figure 5: Ordinary JVM with Threads.

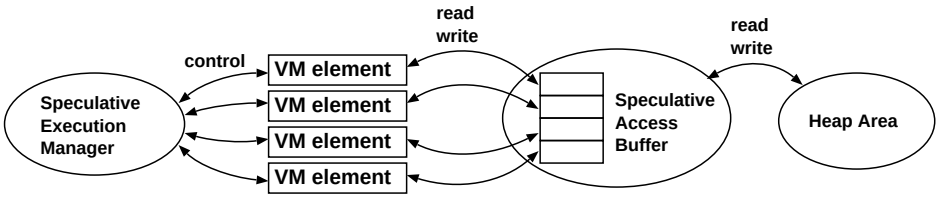

Figure 6: JVM supporting speculative execution of loops.

3.3 Speculative Execution Management

In figure 5 shown an ordinary implementation of JVM. The outline of our Loop Speculation JVM is illustrated in figure 6. VMEs in the figure 6 are the parallel execution units. One VME looks very similar to a Thread of an ordinary JVM. In our implementation, each VME executes one iteration of the target loop speculatively in parallel. Each VME works as an thread on a shared memory machine.

The structure named as Speculative Execution Manager (or SEM) controls speculative execution. Functions of SEM are listed below.

- Start and end speculative execution mode.
- Create the loop invariants for each iteration following the loop information, and distribute them to each VME.

- Check the success or failure of the loop invariant prediction at the end of each iteration.
- Keep the records of memory accesses into Speculative Access Buffer (or SAB) while speculative execution. SEM further examine the record at the end of each iteration to detect data dependency violations.
- Communicate with each VME.

The details of these functions are described further in this section.

3.3.1 Start and End Speculative Execution Mode

When our JVM loads class files at execution time, it searches for the loop information file. The contents of loop information are

1. The name of the method which include the target loop.
2. Loop invariant change in one iteration.
3. Program counter at the start point and end point of the loop.
4. Program counter at the start point and and end point of speculative execution.

If a method is indicated to have a target loop inside, our JVM watches the value of the program counter as it executes the method. When the program counter reaches the point to begin the speculative execution mode, SEM saves the current status of JVM and begin speculative execution mode.

Once speculative execution mode is started, it ends in the following conditions.

1. Program counter reaching the end point for speculative execution mode.
2. Return from the method which includes the target loop.

3. Uncaught exception in the method which includes the target loop.

SEM will terminate speculative execution mode and switch to normal mode, if one of these conditions are satisfied. SEM confirms that the speculative memory accesses are performed, and our JVM continues working as a normal JVM.

Our JVM watches the program counter to detect the start and end point for both speculative execution mode and each iteration. Apparently this causes execution time overhead. We could choose an alternative way such as inserting custom instructions at the corresponding points but we chose the more simple method.

3.3.2 Creation of Loop Invariants

The prediction of loop invariant change in an iteration is written in the loop information. Following to this information, SEM modify the local variables and operand stacks of each VME. Then each iteration is started.

In current implementation, only increment for integer loop invariants is supported. See figure 7.

3.3.3 Checking Prediction of Loop Invariants

When an iteration ends successfully, SEM checks the contents of the local variables and operand stacks. Because the succeeding iterations are started with the predicted values, SEM must check whether the prediction was a success of failure. If the succeeding iteration had been started with wrong values, all the following iterations are wrong. SEM squashes all the following speculation, and re–execute corresponding iterations.

3.3.4 Checking Memory Accesses

In the speculative executing mode, if the preceding iteration *writes* to an object and succeeding iteration *reads* from the same object, a data dependency

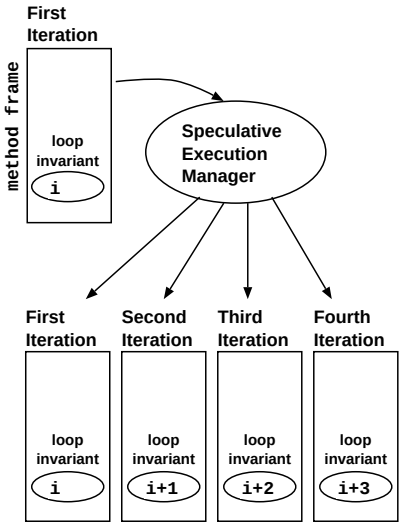

Figure 7: Loop invariant creation.

violation might occur.

Data dependency violation is avoidable if the *write* was preceding the *read* in time. But in our implementation, data dependency violation would occur every time a *write* and a *read* is performed in this order. It is because our JVM performs no communication between each VME at runtime.

See figure 8. The time is flowing from the top to the bottom. The left figure shows an example of a static analysis approach. In this case, if the *write* is performed ahead of time than the *read*, no data dependency violation would occur. It is possible because there is a support from the compiler and some kind of a mechanism such as memory instruction buffer or a special cache.

Because our implementation is based on a pure software, we can not rely on such support. As shown in example on the right of the figure 8, despite that the *write* is performed before the *read*, a logically unnecessary data dependency violation is caused. This is because the preceding *write* would not be performed in reality and the *read* is performed speculatively.

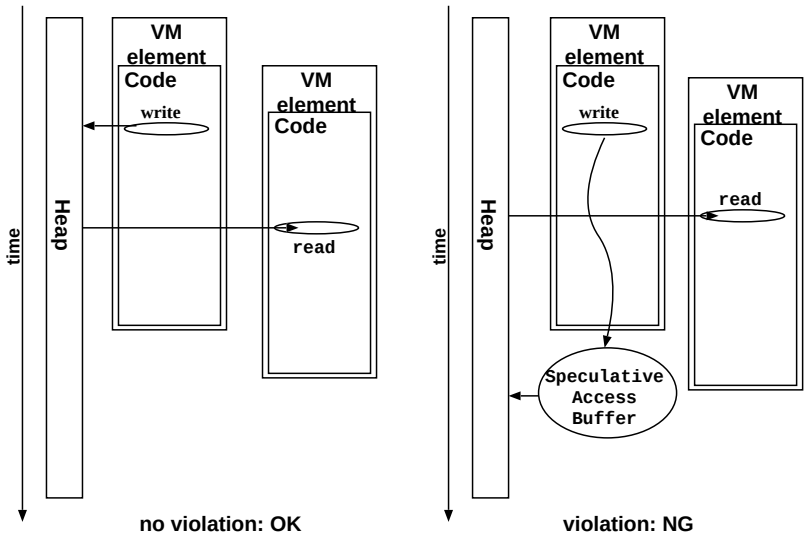

Figure 8: Data dependency violation.

While speculative execution, all *writes* to objects in heap area are recorded in Speculative Access Buffer (or SAB) and performed after the iteration is confirmed be be a success. *Reads* are performed at the time but the address of the *reads* are also recorded into SAB. The entry of the record is, the opcode of the memory access instruction, the parameters to the instruction, and the value which was read from or written to the heap area.

The recording scheme is as follows. Assume that there is an object O in the heap area, and the JVM is now executing an iteration speculatively.

1. The first *read* from O in this iteration is kept in SAB.
2. The last *write* to O in this iteration is kept in SAB.
3. After a *write* to O had occurred, *reads* from O get the value from the recorded entry of the *write*.

After the iterations is finished, SEM does operations as follows.

1. Examine the record in SAB to check the data dependency violation. If no violation was detected, perform the recorded *writes*.

2. The *writes* recorded in SAB are kept until it is certain that the *reads* in succeeding iterations can receive the results of the *writes*.

If one or more data dependency violation was detected, SEM squashed all the iterations including the one just finished.

3.3.5 Communication with Each VM Element.

When the loop variants are created and it is ready to start an iteration, SEM sends `speculative mode start` command to the VME. At the end of an iteration or end of speculative execution mode, each VME sends corresponding status to SEM. If the SEM receives the `speculative mode end` status, it sends `stop` command to all remaining VMEs.

4 Results and Conclusion

We implemented our JVM on Sun Microsystems Enterprise 10000. SEM and each VME are implemented as a thread. The communications between thread are performed with an ordinary store followed by a memory barrier instruction. See SPARC instruction manual [10].

We measured the speedups on loop kernels of two simple programs. In one program there is no heap access instruction in the loop. In the other, there is 1 *read* in an iteration, and 1 *write* in 10 iteration. The *read* and *write* instructions are programmed to cause 1 data dependency in 10 iterations.

The results are shown in figure 9. We measured the speedups for loop in four different sizes. The approximate numbers of the instructions in one iteration are 1K, 10K, 100K, and 1M in each. Speedups are possible if a loop contains more than 10000 instructions. Lines in the graphs starts at somewhere lower

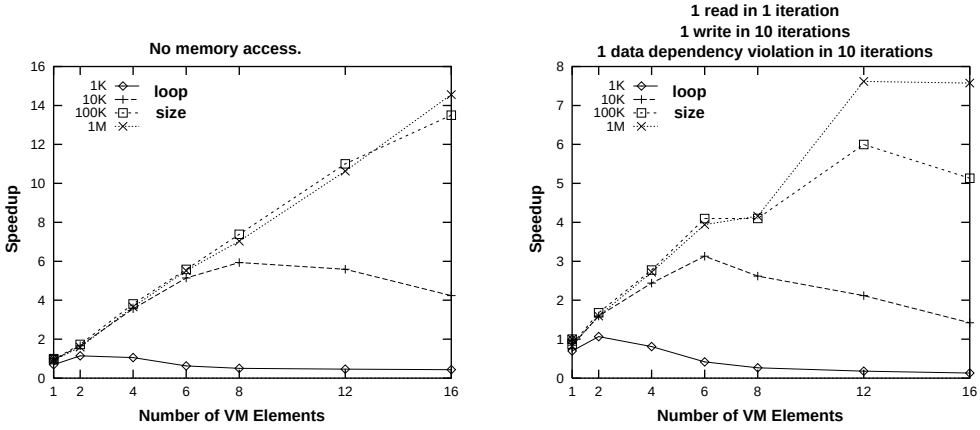

Figure 9: performance comparison

than 1.0. This is because the first point shows the case in which the JVM tries to execute each iteration separately but has only one VME. This case shows a similar example to the case in which every iteration has data dependency thus the loop is executed sequentially.

5 Related Work

There are several approaches for multiple block level parallelism. In most of the approaches, parallelization is performed by using both static analysis and dynamic reconfiguration, but there are many variations in the weight of the two approaches.

In researches such as multiscalar [3], superthreading architecture [4], and hydra project [5], parallelization is performed by a combination between hardware and compiler. Program is partitioned into blocks by the compiler. In [3] a binary to binary translation is used to exploit parallelism. [4] and [5] perform parallelization at compile time.

In [3] execution units are connected by a circular unidirectional ring. The compiler makes the control flow graph and divide a program into blocks. The

data dependency through registers are analyzed at the compile time. The compiler does not perform any analysis of data dependency through memory accesses and the execution is done speculatively. To check data dependency through memory access, memory accesses are recorded into a special buffer called *Address Resolution Buffer*. Because the *Address Resolution Buffer* proposed in [3] was a very expensive hardware, in [4], the compiler checks the memory access instructions statically and mark potentially dangerous instructions. Only address and value of store instructions are recorded. Because store instructions are fewer than load instructions, this would save the size of the buffer. Furthermore these memory accesses instructions are replaced to a code which detect data dependency at runtime and wait for the data to be computed. This is a combination of runtime data dependency checking and runtime synchronization. The speculation is done at branches and if the prediction was incorrect the following blocks are killed. Unlike these two approaches, [5] does not need any special hardware. It only needs a fast enough communication between each execution unit. Data dependency through memory accesses are handled by a special cache mechanism between primary and secondary cache. The data dependency through registers are cleared by compiler. Each execution unit are given different code for each blocks. For example, to execute loops in parallel, the loop invariants are analyzed and incremented at the compile time.

The described three approaches use a compiler or binary to binary translator to exploit parallelism. But we estimate that the most parallelism is found in loops. That is the basis of the idea, "Run–time restructuring". It was proposed in the paper [6]. On–chip multi processor based approach of dynamic parallelization is described in the paper. The proposed machine dynamically analyze the program and find the instruction which change the loop invariant. Binary compatibility is maintained in this approach. If the loop invariant operation was found, the corresponding instruction is applied repeatedly for each iteration of

the loop. Instead of forwarding the value through a ring concatenated register, the value is computed at each execution unit.

There are studies about exploiting instruction level parallelism for Java. A runtime re–compilation for VLIW is proposed in an application of DAISY [1, 2]. Sun Microsystems had announced in the [8], a dynamic analysis of the program to decide the weight of optimization on JIT compile.

6 Summary

We propose a "Run–time restructuring" to exploit block level parallelism in Java program. This is a dynamic parallelization limited to loops. Static precise analysis of the existing bytecode or source code would be also possible, but we decided it is more important to maintain the class file unmodified. Because the original "Run–time restructuring" uses a simple dynamic analysis of loops in existing binary program, it is possible to perform the analysis while runtime. Therefore it is practical to apply this technique to JVM. We implemented an experimental JVM which performs speculative parallel execution of loops. It was a preliminary experiment. In conclusion, a pure software based speculative execution can improve performance in a good condition.

As we adopted the idea software based parallelization, we estimated that frequent data dependency violation would make the overheads high. Nevertheless, to hide the overheads in speculative execution management, we need to divide the program into coarse grain blocks. In conclusion, the target block should not cause frequent data dependency violation and at the same time, the block must be large enough to hide the latency. Consequently, in our implementation, we limited the target for speculative execution to large loops.

Ordinary JVM support the mechanism of garbage collection. As a result of garbage collection, it is possible that two objects of the same type is allocated at

the same address. In this case, to distinguish two objects we need to add some functions to garbage collection. Our implementation does not support garbage collection. The research about garbage collection for speculative execution is necessary.

The experiment was a preliminary one and there is a Java specific problem left. That is the use of native method calls in JVM. For example, to perform IO a JVM inevitable calls native methods. This means it is impossible to support parallelization just by watching execution of Java bytecode. We should combine our JVM with techniques used in binary program based speculative execution.

As a future work, we need to combine speculative execution with JIT compiler to compare with recent JVMs in reality. Researches about the comparison between static analysis and dynamic reconfiguration is also necessary. Building a Java compiler which assist the JVM to perform more precise parallelization would also be an interesting issue. Overall, the approach of parallel speculative execution can stand further improvement in combination with existing researches.

References

[1] K. Ebcioglu and E. Altman, "Daisy: Dynamic compilation for 100% architectural compatibility," tech. rep., IBM Research Division, T.J.Watson Research Center, Yorktown Heights, New York, 1996.

[2] K. Ebcioglu, E. Altman, and E. Hokenek, "A java ilp machine based on fast dynamic compilation," in *IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java, Eilat, Israel, January 9-10*, 1997.

[3] G.Sohi, S.Breach, and T.Vijaykumar, "Multiscalar processors," in *proceedings of the 22nd Annual International Symposium on Computer Architec-*

ture, 1995.

[4] J.Y.Tsai and P.C.Yew, "The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation," in *proceedings of Internatinal Conference on Parallel Architectures and Compilation Techniques*, 1996.

[5] K.Olukotun, B.Nayfeh, L.Hammond, K.Wilson, and K.Y.Chang, "The case for a single-chip multiprocessor," in *proceedings of the 7th International Symposium on Architectural Support for Parallel Languages and Operating Systems*, 1996.

[6] J. Tamatsukuri, T. Matsumoto, and K. Hiraki, "Run–time loop restructuring for on–chip parallel processor," in *In proceedings of PDPTA'98* (H.R.Arabnia, ed.), (Las Vegas, Nevada, USA), pp. 1489–1496, 1998.

[7] T. Londholm and F. Yellin, *The Java Virtual Machine Specification*. ADDISON–WESLEY ISBN 0-201-63452-X, 1996. Also available at http://java.sun.com/docs/books/vmspec/.

[8] Java^{TM} HotSpot^{TM} VM Team, "Jdk software performance technologies," 1998. slides are available at www.javasoft.com.

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

[10] D. L. Weaver and T. Germond, eds., *The SPARC Architecture Manual*. Englewood Cliffs, New Jersye 07632: Prentice Hall, 1986. ISBN 0–13–099227–5.