On-line Trace Based Automatic Parallelization of Java Programs on Multicore Platforms

Yu Sun and Wei Zhang
Department of ECE, Virginia Commonwealth University
wzhang4@vcu.edu

Abstract

We propose a new approach that automatically parallelizes Java programs at runtime. The approach collects on-line trace information during program execution, and dynamically recompiles methods that can be executed in parallel. We also describe a cost/benefit model that makes intelligent parallelization decisions, as well as a parallel execution environment to execute parallelized code. We implement these techniques upon Jikes RVM and evaluate our approach by parallelizing sequential benchmarks and comparing the performance to manually parallelized version of those benchmarks. According to the experimental results, our approach has low overheads and achieves competitive speedups compared to manually parallelized code.

1 Introduction

Multi-processor has already become mainstream in both personal and server computers. Even on embedded devices, CPUs with 2 or more processors are increasingly used. However, software development does not catch up with hardware at this time. Designing programs for multi-processor computers is still a difficult task and requires a lot of experiences and skills. Besides, a big number of legacy programs that are designed for single-processor computers are still running and need to be parallelized for better performance. All these facts require a good approach for program parallelization.

In this paper, we propose an automatic parallelization approach based on Java virtual machine (JVM). Traces, which are sequences of actually executed instructions, are used in our approach as units of parallel execution. Furthermore, we collect trace information on-the-fly during program execution, so that our approach works simply with any given Java byte code. There is no need for any source code or profiling information. We also utilize some excellent existing features in JVM, such as run-time sampling, on-demand recompilation and multi-thread execution. Enhanced by run-time trace information, our experimental results show that this approach is able to achieve competitive results of parallelization for Java programs, as compared to parallel code by hand.

2 Overview

Figure 1 depicts the system that we implemented upon Jikes RVM [9], and also the main procedure of parallelization. There are four main components in our system, including an on-line trace collector, a cost/benefit model, a parallelizing compiler and a parallel execution environment. First of all, we utilize the on-line sampling-based profiling mechanism of Jikes RVM to identify “hot methods”, which take major parts of the total execution time of Java programs. We only consider hot methods as parallelization candidates in order to reduce unnecessary instrumenting and threading overheads.

In our approach, we filter the hot methods based on several heuristics. First, only methods from user applications are selected. Thus Java core and VM methods are not considered in this work. Second, the length of the methods can not be very short. Third, the methods must contain loops. We call the hot methods satisfying these criteria *good candidates* for parallelization.

After good candidates are identified, we recompile them to insert instrumenting code. As a result, trace information can be collected in the next execution of instrumented methods, and then it is delivered to a cost/benefit model to decide whether the traces are profitable to be parallelized or not. After that, those traces worthy of parallelization are passed to a parallelizing compiler, which recompile the methods that contain those traces. And finally, parallelized traces execute in parallel inside the Parallel Execution Environment (PEE). The details of these components are described in the following sections of this paper.

3 On-line Trace Collection System

3.1 Trace Formation

A trace is defined as a sequence of unique basic blocks which are executed in sequential order during the execution of a program [2]. An example of trace formation is in Figure 2, which is a control flow graph of some program section. Trace T1 is the sequence of $\{B1, B2, B3, B4, B6\}$, and T2 consists of basic blocks $\{B2, B3, B4, B6\}$. In contrast, a sequence of basic blocks $\{B2, B5, B6, B2\}$ may be executed at run-time due to the loop structure, but it is not a trace because B2 occurs twice in the sequence. Traces can be collected by a trace collection system (TCS) that monitors a program’s execution. In our particular case, we extend trace definition to two types: Execution Trace and Memory Access Trace.

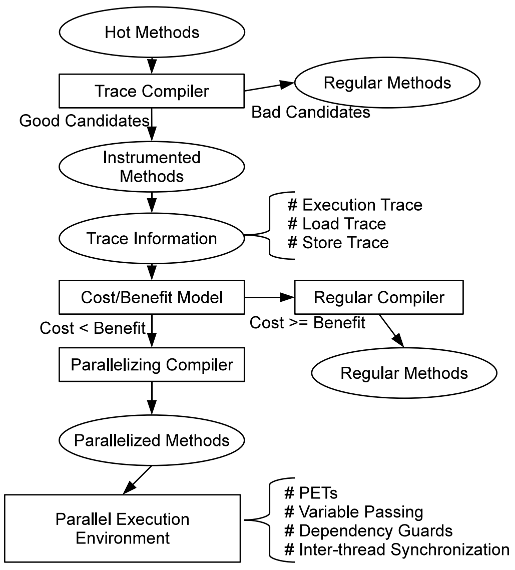


Figure 1. Procedure of on-line trace based automatic parallelization

- *Execution Trace*
  An execution trace records a sequence of executed instructions and basic blocks. Instrumenting code sections are injected at every jump and branch instruction, recording their run-time targets. Similar instrumentation is applied to method calls and returns. Backward jumps or branches are treated particularly to identify loops and collect loop information like induction variables and loop boundaries. Time stamps are also collected at instrumenting points, which is needed by future analysis during parallelization.

- *Memory Access Trace*
  A memory access trace is a sequence of read and write operations to the memory, including variables, fields and array references. It is collected for data dependence analysis during parallelization. As the actual memory addresses can be collected by on-line TCS, it is now possible to perform much more accurate dependence analysis compared to static approaches, especially for class fields and array accesses. To reduce memory space overheads, continuous memory addresses are combined to one entry, and only a limited number of entries are kept in one trace. Our study shows this approach works well with regular loop-carried dependencies.

These two types of traces are bound so that each execution trace has only one memory trace. And the boundaries of traces are determined by specific types of instructions at run-time. A trace starts upon detecting one of two possible trace entry points:

- First instruction of an instrumented method,
- Exit points of another traces.

On the other hand, traces end at several other program points:

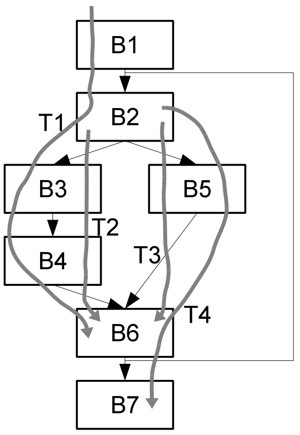


Figure 2. Traces in a program section

- Backward jumps/branches,
- Last instruction (usually return instruction) of an instrumented method,
- points where length of a trace exceeds a given threshold, i.e. 128 bytecode instructions in this study, which is rare to be reached based on our experiments.

Based on the conditions described above, traces are constructed and stored as objects in the JVM. Also, additional information such as execution count of traces is also recorded and saved within the trace objects. As a result, the hot methods that are potential targets for parallelization can be split into multiple traces. For example, a single level loop without any branch can be normally divided into three traces: one is the first iteration with some instructions before the loop; another trace is the loop body that may be executed many times; and the last one is the last iteration that jumps out of the loop and continues executing following instructions.

3.2 Trace Collection

There are several ways to implement a TCS, such as a hardware profiler and a machine code recorder in an interpreting VM. However, our choices are narrowed down by the unique constraints that our on-line TCS needs to satisfy.

- *Low overheads*
  As our TCS is on-line, any overhead introduced here is counted towards the final performance of whole system. Hence, the time spent on TCS must be kept as little as possible. In addition, trace information collected by on-line TCS is stored inside of JVM's runtime heap together with the executing program. As a result, we also need to control space overheads of TCS to avoid unnecessary GC, which may harm performance and complicate dependence analysis.

- *Detail and accuracy*
  On the other hand, on-line TCS is expected to provide detailed and accurate trace information of parallelizing candidates. The parallelizing system relies on this information for the following tasks, including decision making, dependence analysis and final parallelizing

compilation. At least, all branch/call targets must be recorded, as well as memory accesses to variables and arrays.

Due to the two requirements and the fact that Jikes RVM is compilation based, we choose to enhance Jikes RVM's baseline compiler with the ability of inserting instrumenting code into generated machine code. Instrumenting code is executed when certain points in program execution is reached. At that time, useful run-time information like register values and memory addresses is sent to trace collector running in VM. In order to minimize overhead while recording detailed and accurate data, we make the TCS selectively work on different detail levels for various parts of a program. Specifically, there are three levels in trace collection.

- **Level 0** - The lowest trace collection level where only records of method calls and returns are kept. This level is applied on most methods that are not hot.
- **Level 1** - In this level, full execution traces are recorded, while no memory access trace is kept. This level is used for the non-loop sections in hot methods.
- **Level 2** - The highest level that records both execution and memory access traces. The most detailed and accurate information is provided by this level for future analysis and parallelization. Only loops in hot methods, which are best parallelizing candidates, can trigger level 2 trace collection.

Hence, only a small but frequently executed part of a program is fully instrumented. Overheads are significantly reduced while most useful traces still have detailed and accurate information. Besides, after being recompiled and instrumented, the methods execute only once to collect trace information. Upon finishing a method's profiled execution, trace information is passed to the cost/benefit model for making parallelization decisions. Then the method is either parallelized or recompiled into plain machine code without any instrumenting code. No matter what decision is made, there is no more trace collection overhead after recompilation. Because of all the efforts described above, the overhead of our on-line TCS is acceptable. On the other hand, most executions of a method have very similar behaviors even though they have different parameters. Furthermore, trace collection occurs after warm-up stage of the Java programs, which makes the execution behavior rather stable. Thus the trace information is accurate enough for parallelization.

The output of trace collection is a data structure named Trace Graph. In a Trace Graph, the nodes are the traces and the edges describe the control flow dependence among traces. For example, if a trace T2 starts right after another trace T1 exits, a directional edge will be created from T1 to T2. Each edge has a weight that records how many times the edge is passed. A full example is shown in Figure 3(a).

4 Dependence Analysis

Given different memory access patterns, traces may be data dependent on each other. In order to resolve the dependence among traces, we utilize the memory access trace information collected by our on-line TCS. There are two types of dependencies that we deal with: local variables, and arrays. They are processed separately based on memory access traces.

For the local variables in a trace, we generate both read and write lists from the corresponding memory access trace. Since local variables are stored in JVM run-time stack and represented by unique integer values in Java byte codes, read/write lists are sequences of integers. We first simplify the lists of a trace according to in-trace dependence, then calculate inter-trace dependencies with other traces.

Arrays have to be carefully managed because they are in the heap and the only way to access an array element is through its memory address. Our memory access trace records the actual memory addresses accessed by the traces during the instrumenting execution. When an instrumented method executes, its load and store instructions, as well as their memory addresses, are monitored and stored in its memory access traces. Then we perform a similar analysis like what is done for the local variables. The only concern is memory usage of storing those addresses. To deal with this problem, we compact one section of continuous addresses into one data record that stores a range of memory addresses instead of a single address. This compaction is able to save a lot of memory spaces as observed in our experiments.

We also try to simplify dependence analysis by introducing dependent section, which is a section in a trace containing all instructions dependent to another trace. For instance, a single loop has 100 instructions and the 80th and 90th instructions carry dependence between loop iterations. In this case the dependent section of loop body trace is 80 to 90 after dependence analysis. Dependent section is used based on the observation that in most cases only a small section of instructions in a trace carries dependencies. Besides, using single dependent section for each trace greatly reduces synchronization/lock overheads in the busy-waiting mode, which is used in our parallel execution model.

5 Cost/Benefit Model

After collecting trace information for hot methods and analyzing dependencies between traces, our approach then decides whether it is worthy or not to parallelize them. We introduce a cost/benefit model inspired from the one used in Jikes RVM's adaptive optimization system [1]. This model calculates the estimated time of both sequential and parallel execution of a parallelizing candidate with its trace information and some constants. Parallelization is performed if the following inequality is satisfied.

$$T_{EP} < T_E \times f \tag{1}$$

Here, T_{EP} is the estimated time of parallel execution of a trace or a group of traces, including the overheads due to trace collection and parallelization; T_E represents the estimated time of sequential execution; and f denotes a control factor which is a constant 1 and can be tuned. For example, if we want to parallelize only the methods that bring high benefit after parallelization, f can be set to 0.8 or 0.7 in order to filter others out. Because estimation is made on-the-fly during run-time, we have no idea of the exact future execution time for a given section of code. We thus use the same heuristic as the one in Jikes RVM's AOS model, that is, future execution time is equal to the execution time in the past. This assumption works well with AOS, as well as our approach.

$$T_E = T_P = S \times Q \tag{2}$$

In Equation 2, T_P is the execution time in the past, calculated by number of samples S and sampling interval Q. Furthermore, the estimated time of parallel execution can be calculated by the Equation 3, where T_W is waiting/idle time during parallel execution, $T_{overhead}$ represents the time spent on parallelizing and other overheads, and N is the number of available processors (cores).

$$T_{EP} = \frac{T_E}{N} + T_W + T_{overhead} \tag{3}$$

In Equation 4, T_W can be roughly estimated with α, the ratio of the execution time of a dependent section to the total execution time of the whole program. α is calculated using recorded execution time of both the instrumented method and its dependent sections. The timing information is collected during trace collecting instrumentation. When this ratio is small enough, T_W can be even 0. The formal equation to calculate T_W is as follows.

$$T_W = \begin{cases} 0 & \text{if } \alpha \leq \frac{1}{N} \\ \alpha T_E - \frac{T_E}{N} & \text{if } \alpha > \frac{1}{N} \end{cases} \tag{4}$$

And $T_{overhead}$ consists of two components. One is the compilation time of parallelizing compiler, which can be estimated by the byte code length and compiler's compilation speed. The other comes from parallel execution environment and can be represented as a constant. As a result, we have the final equation of future parallel execution time.

$$T_{EP} = \begin{cases} \frac{T_E}{N} + T_{overhead} & \text{if } \alpha \leq \frac{1}{N} \\ \alpha T_E + T_{overhead} & \text{if } \alpha > \frac{1}{N} \end{cases} \tag{5}$$

After T_E and T_{EP} are calculated respectively in (2) and (5) by the cost/benefit model, inequality (1) is applied to make the final decision. Obviously, parallelization is more likely to be performed when dependence is not intensive, i.e. α is a small value.

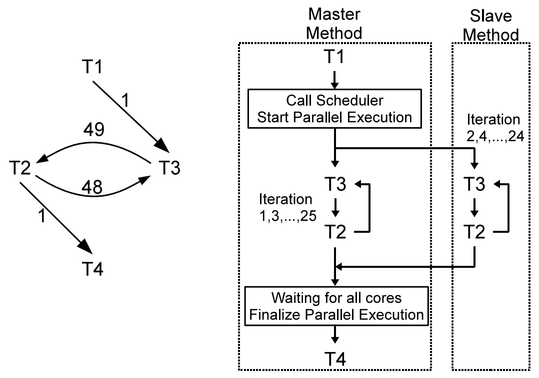
6 Parallelizing Compiler

Parallelizing candidates that pass the check in cost/benefit model described above are sent to the parallelizing compiler. One candidate is then compiled to N new methods, where N is the number of available processors (or cores). For now, our parallelization approach does not consider the underlying hardware features except the number of processors. These N new methods can be divided into two types: a master that runs on the main thread as a part of original sequential execution; and slaves that only executes parallelized tasks and are destroyed at the end of parallel execution. The main workload in the original method, which consists of repetitively executed traces, is partitioned in the unit of trace and assigned evenly to both master and slave methods. A general description of our trace parallelization approach is given in Algorithm 1, where function `FindCircle` looks for circles in a Trace Graph and function `FindOutMostLoop` returns the outermost loop that contains a given trace circle.

Figure 3 illustrates an example of parallelizing traces from Figure 2 on a dual-core machine, in which we assume that only four traces T1, T2, T3 and T4 are collected at run-time, and their relationship is described by a Trace Graph shown in Figure 3(a). These assumptions are made to simplify the example, because traces and their relationship may be changed completely if inputs are different. We also assume the loop in Figure 2 iterates 100 times, and there is no data dependence. Parallelizing compiler looks into Trace Graph for circles, which is T2 and T3 in Figure 3(a), because repetitively executed traces usually have good parallelism. After checking data dependence of the

Algorithm 1: Trace Parallelization

```
input : TG - Trace Graph
input : BC - Byte code of parallelized method
input : N - Number of parallel threads
output: Compiled master and slave methods
1  C ← FindCircle($TG$);
2  while C $\neq NULL$ do
3      L ← FindOutMostLoop(C,$BC$);
4      Create $N-1$ worker classes with slave methods;
5      Copy traces in L to every slave methods;
6      Insert dependence guards for dependent sections;
7      Change induction variable of $k$th thread to $i \times N + k$;
8      Add code to master method that invokes slaves and waits for them;
9      $TG \leftarrow TG - \{$All Traces in L $\}$;
10     C ← FindCircle($TG$);
11 end
```



(a) Trace Graph (b) Parallelized Traces

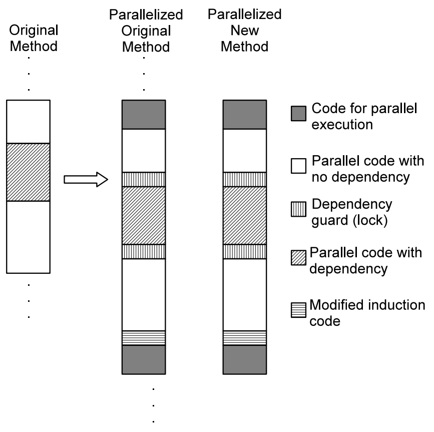
Figure 3. An example of trace parallelization

traces in a circle, parallelizing compiler construct both master and slave methods, as shown in Figure 3(b). Loop induction variables in both methods are configured based on execution frequencies (edge weights in Figure 3(a)) in Trace Graph. Iteration $i \times NumberOfCores + k$ is assigned to the kth core so that workloads are evenly dispatched, where $i = 0, 1, 2, ...$. Also, some maintenance code segments are inserted into the master method in order to manage the multi-threaded execution. Its details are discussed later in this paper. As a result, the theoretical speedup after parallelization in this example is approximately 2, if we ignore all the overheads of parallelization and multi-threaded execution.

Moreover, Figure 4 depicts the details of code generated by a parallelizing compiler. Several special code segments are inserted into both the original method and parallelized new methods. They are:

- Code preparing parallel execution: Pass variables and invoke parallel execution.
- Code for dependency guards: Obtain and release locks.
- Code of modified induction/reduction: Used for parallelized loop iterations.
- Code finalizing parallel execution: Pass variables back and clear the scene.

As described in Algorithm 1, our approach is a hybrid of trace and loop parallelization. The reason of not using pure



Original Method
Parallelized Original Method
Parallelized New Method
Code for parallel execution
Parallel code with no dependency
Dependency guard (lock)
Parallel code with dependency
Modified induction code

Figure 4. Code generated by parallelizing compilation

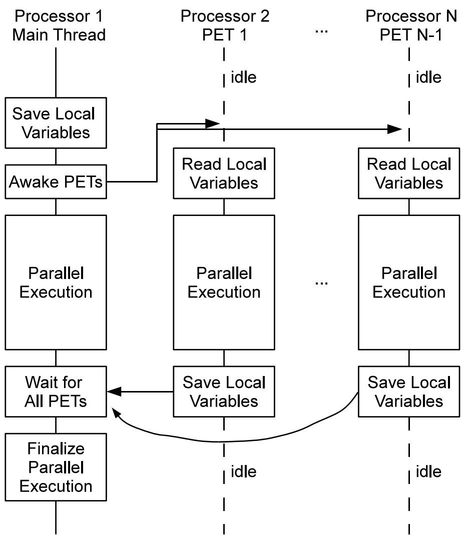


Figure 5. Parallel Execution Environment

trace based parallelization is a limitation of on-line trace collection. Due to existence of branches, a program may have different traces given different inputs, and on-line trace collection does not guarantee to cover all possible traces at run-time. As a result, some execution paths may be missing in parallelized code if pure trace based parallelization is applied. In order to resolve this problem, we combine trace and loop parallelization together. Loop bodies that contain traces are detected and parallelized instead of traces. Hence, all possible paths are covered, and the control flow among traces is taken care of by branches in the loop bodies, which simplifies the code generation for multiple traces. Also, the hybrid approach utilizes more run-time trace information for loop parallelization compared to traditional static loop-based approaches, which provides chances to perform more aggressive and accurate parallelization. A major advantage of our hybrid approach is that data dependence analysis, which is based on run-time trace information as described in Section IV, is much more accurate than what can be done in static loop-based parallelization. For instance, if basic block B3 in Figure 2 accesses some data stored in the heap, it is hard and sometimes impossible for a static loop-based parallelization approach to determine whether this memory access is safe to be parallelized or not. But it becomes much easier in our hybrid approach because the actual memory addresses are recorded during trace collection stage, and thus the memory access pattern of B3 can be constructed and used for parallelization. However, the hybrid approach introduces branches into parallelized code, which may reduce pipeline and cache performance compared to pure trace-based parallelized code that is completely sequential. Besides, dependent sections may increase because more instructions are compiled into the parallelized code. And memory spaces may be wasted for those instructions that will actually never be executed.

7 Parallel Execution Environment

Parallel Execution Environment (PEE) is implemented upon Jikes RVM's multi-thread execution functionality. PEE contains one or more Parallel Execution Threads (PETs), manages copies of local variables passed between the main thread and PETs, and maintains the dependence guards wrapped to the dependent sections in the code. Generally speaking, PEE takes care of executable code generated by the parallelizing compiler, and makes sure they are executed correctly. Figure 5 demonstrates how PEE works with a parallelized code section on N processors.

7.1 Parallel Execution Threads

As shown in Figure 5, PETs are separately assigned to multiple processors. Normally, one PET does not travel between processors to reduce overhead. PETs and daemon threads that start running right after VM is booted. However, PETs stay in the idle state before any parallel execution method is dispatched. PETs may be waken up by the main thread, and it can only be waken by the main thread. After executing the given parallel execution method, a PET goes back to idle and waits for next invocation from the main thread.

There are several benefits by using the wake-up/sleep mechanism instead of creating new threads every time for parallel execution, though the later approach is easier for implementation. First, using a fixed number of threads greatly reduces the pressure on memory usage during runtime. Second, it makes the scheduler more extendable for other advanced scheduling policies.

7.2 Variable Passing

It is necessary to make copies of local variables, which are shared across PETs, and pass them safely to other threads outside of the main thread. We use arrays to do this job. Each parallelized code section keeps its own variable passing array, which is pre-allocated at the compilation stage. The array is filled with local variable values before waking up PETs, so that at the beginning stage of each parallel executed method, those values can be read correctly from the array. Similarly, writing back to this array is the last job of a parallel executed method. And the main thread restores local variables from the arrays before moving on to the next instruction. This procedure is also shown in Figure 5.

Benchmarks		Execution Time(s)		Input Size	Description
		BASE	OPT		
Java Grande Section 2	crypt	2.73	1.38	3,000,000	IDEA Encryption
	lufact	2.72	0.27	500	LU Factorisation
	series	6.99	5.11	10,000	Fourier Coefficient Analysis
	smm	3.96	0.71	50,000	Sparse Matrix Multiplication
Java Grande Section 3	moldyn	33.42	2.80	2,048	Molecular Dynamics simulation
	montecarlo	10.35	4.10	2,000	Monte Carlo simulation
	raytrace	35.97	4.03	150	3-D Ray Tracer

Table 1. Description of benchmarks

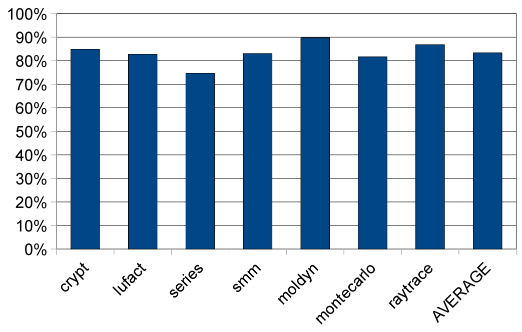


Figure 6. Percentage of hot traces in total execution time

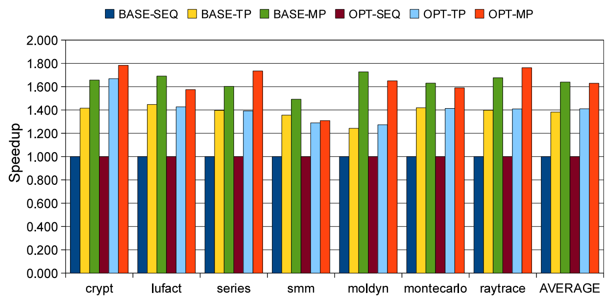


Figure 7. Speedup of trace and manual parallelization (baseline and optimized, respectively)

7.3 Dependence Guards

Dependence guards are implemented with Java thread synchronization mechanism, particularly, locks. During parallelizing compilation, dependent code sections are identified based on trace information, and then wrapped up with two special code segments called dependence guards. First code segment is inserted before a dependent section to acquire a lock, and the other one is right after that dependent section to release the lock. To reduce the complexity and synchronization overhead, each parallelized section holds only one lock object. This means all dependent instructions have to share one lock, and consequently the dependent section is the union of all individual dependent instructions. Although this approach increases α value and thus may decrease speedup after parallelization, we believe it is a simple and fair solution to avoid too many locks and high synchronization overhead.

8 Evaluation Methodology

We implemented our approach described in Section 2 to 7 as an extension on Jikes RVM version 3.1.0. The code is compiled with Sun JDK version 1.5.0_19 and GCC version 4.3.3. We use a dual-core Dell Precision 670 with 2 Intel Xeon 3.6 GHz processors and 2 GB DDR RAM to run the experiments. The operating system on that machine is Ubuntu Linux with kernel version 2.6.28. Each processor has an 8 KB L1 data cache and a 1024 KB L2 cache.

The benchmarks that are used in our experiments are from Java Grande benchmark suite [10, 12]. We use two groups of benchmarks. The first group consists of four benchmarks from *section2*: `crypt`, `lufact`, `series` and `smm`. They are short codes that carry out specific operations frequently used in grande applications. Group 2 contains three benchmarks: `moldyn`, `montecarlo` and `raytrace` from *section3*, which is a group of large-scale application tackling real-world problems. These benchmarks all exhibit good data-level parallelism. Another reason of choosing them is that their manually parallelized versions are also provided in Java Grande benchmark suite, which we can directly compare our approach with. We use the small input size in order to make it easier to observe impacts of our approach sdand all kinds of overheads. Benchmark details such as sequential execution time (baseline and optimized) and input size are described in Table 1.

9 Experimental Result

We first measure the fraction of trace execution time to justify whether it is worthy or not to parallelize traces. Then we evaluate our approach by two metrics: speedup and overhead. Speedup is defined as the ratio of sequential (base) execution time to parallel execution time. We measure the speedup for both our automatic parallelized version and the manual parallelized version provided by Java Grande benchmark suite. Overhead is defined as the time spent on trace collection and recompilation.

9.1 Trace Execution Fraction

We first measure the execution time of traces by instrumenting baseline compiled benchmark programs. We only keep hot traces that execute more than 100 times in the results, because frequently executed traces are potential tar-

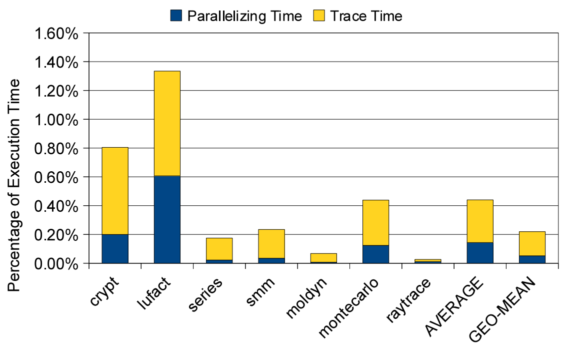


Figure 8. Overhead of compilation and trace collection

gets for parallelization. The results are shown in Figure 6. The execution time spent on frequently executed traces ranges from 74.60% to 89.71%, indicating that the major part of program execution is taken by those hot traces. Hence, decent speedup can be expected if the hot traces are well parallelized.

9.2 Speedup

The speedups of all 7 benchmarks are shown in Figure 7, where `SEQ` represents sequential execution, `TP` stands for trace parallelization, and `MP` represents manual parallelization. We also measure the speedup with and without optimization of Jikes RVM, which are represented as `BASE` and `OPT` respectively. With our approach, all seven benchmarks achieve obvious speedups with and without optimization. The average speedups are 1.38 (baseline) and 1.41 (optimized) on two processors, as shown in Figure 7.

On a dual-core machine without optimization, the first group of four benchmarks shows fair speedups around 1.4. Among other three benchmarks from *section3* of Java Grande benchmarks suite, `moldyn` gains the least speedup with our approach while doing the best with manually parallelized version. The reason is that the α ratio of dependent section to whole parallelized code section in `moldyn`'s only hot method `particle.force()` is high. There is relatively heavy loop-carried dependence in that code section. Our approach can only automatically insert dependence guards that make parallel threads work in the busy-wait mode. Thus `moldyn` shows lower speedup than other two benchmarks, which have smaller α values in their parallelized code sections.

In the beginning of experiments, we do not allow any optimizing recompilation to be done during program execution. The purpose is to study our approach without any possible interference. However, Jikes RVM provides powerful adaptive optimization system (AOS) that boosts performance of Java programs. Hence, we also study the impacts of optimizations on speedup. The results in Figure 7 indicate that our automatic parallelization still works well with Jikes RVM's AOS. The average speed up is 1.41, which is similar to the result in section 7.1 and competitive to manual parallelization with an average speedup of 1.63. Jikes RVM's AOS and our approach integrate naturally because good parallelizing candidates are usually also good targets for optimization, and our framework is capable to exploit the optimization compiler during parallelization.

Benchmarks	Par. Methods	Par. Time	Trace Time
crypt	1	0.20%	0.60%
lufact	3	0.61%	0.73%
series	1	0.02%	0.15%
smm	1	0.03%	0.20%
moldyn	1	0.01%	0.06%
montecarlo	6	0.12%	0.31%
raytrace	2	0.01%	0.01%
AVERAGE	-	0.14%	0.30%
GEO-MEAN	-	0.05%	0.17%

Table 2. Overheads of parallelization

Compared to manually parallelized version of all seven benchmarks, our approach shows less speedup. The reason is that manual parallelization applies more aggressive parallelizing techniques, while our approach uses simple ones. For example, a reduction instruction like $a = a+1$ in a loop can be parallelized manually without waiting for previous iterations, by calculating separate sums on each thread and add them together after parallel execution. However, our approach does not apply this kind of ”clever” parallelization. Instead, we insert locks for variable a and make parallel threads wait until previous value is written to memory. This gap between our approach and manually parallelized code may be filled by equipping our approach with better parallelizing techniques, which will be part of our future work.

9.3 Overhead

The overheads introduced in our approach in shown in Table 2 and Figure 8. They are surprisingly small for a system with on-line instrumenting and dynamic recompilation. As described in Section 2, we put a lot of work on trace collection to reduce overheads. By limiting high-level trace instrument to hot methods and executing instrumented code only once, the time overhead of trace collection is quite satisfying, only 0.30% on average. On the other hand, compilation overhead is also small. This is because of the baseline compiler that we use for both instrumenting and parallelizing. Although generating non-optimized machine code, the baseline compiler is the fastest compiler provided by Jikes RVM. The average compilation overhead is 0.14%. However, higher compilation overhead can be expected if optimizing compiler takes the place of current baseline compiler.

It is also shown in Table 2 that overhead is related to the number of parallelized methods. We divide all 7 benchmarks into two groups based on their total execution time, where group 1 has higher overheads because of their shorter running time. In the group of first four benchmarks, `lufact` has 3 methods parallelized while others have only one, thus its parallelizing and trace overheads are both highest in this group. In the second group, `montecarlo` has 6 methods parallelized, and consequently its overhead (0.43%) is much higher than `moldyn` (1 method, 0.07%) and `raytrace` (2 methods, 0.02%). Additionally, size of accessed memory is another factor that affects trace collection overhead, which is the reason of `moldyn`'s trace collection overhead overrunning `raytrace`'s.

10 Related Work

Some work has been done on trace based automatic parallelization [4, 3]. This work performs an additional execution to collect trace information off-line. Besides, only simple loop induction/reduction dependency is considered in this work. In contrast, we use on-line trace collection to avoid the expensive profiling execution, and introduces more advanced dependency analysis in order to deal with more complicated Java programs.

In the past two decades, a number of parallelizing compilers are developed. Most of them are designed for static high level programming language, like C and Fortran. Some examples are SUIF [8] and Rice dHPF compiler [6]. All these works focus on convert source code into high quality parallel executable code. In another word, source code is required for these approaches. In contrast, our work does not need any source code.

There are also some researches utilizing JVM for parallelization. Chan and Abdelrahman [5] proposed an approach for the automatic parallelization of programs that use pointer-based dynamic data structures, written in Java. Tefft and Lee [13] use Java virtual machine to implement an SIMD architecture. Zhao et. al. [14] developed an online tuning framework over Jikes RVM, so that a loop-based program can be parallelized and tuned at runtime, with acceptable overheads, increasing the performance when compared to traditional parallelization schemes. The core of their work is a loop parallelizing compiler which detects parallelism in loops, divides loop iterations and creates parallel threads. None of these work utilizes trace information in their systems.

Another interesting research area is thread-level speculating. Pickett [11] apply speculative multithreading to sequential Java programs in software to achieve speedup on existing multiprocessors. Also, Java runtime parallelizing machine (Jrpm) [7] is a complete system for parallelizing sequential Java programs automatically. It is based on a chip multiprocessor (CMP) with thread-level speculation (TLS) support. However, speculation always requires additional hardware support. In contrast, our approach is purely software fully implemented inside of Jikes RVM, without any hardware requirement.

11 Conclusion

In this paper, we have introduced a novel approach of automatic parallelization for Java programs at runtime. It is a pure software-based online parallelization built upon Java virtual machine. The parallelization can be done without any source code or profiling execution. Our experimental result indicates that good speedup can be achieved for real-world Java applications that exhibit data parallelism. All benchmarks are accelerated and the average speedup is 1.38. While this is less than an ideal speedup on a dual-core processor (i.e. 2), it is not too far away from the speedup of even manually parallelized version of those Java programs, considering that the parallelization is done automatically at runtime by the compiler. Also, we observe very small overhead, only 0.44% on average, is introduced by our approach during trace collection and recompilation. To conclude, our on-line trace based parallelization can efficiently parallelize Java programs.

Acknowledgement

This work was funded in part by NSF grants CCF 0914543 and CNS 0720502.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalape no jvm. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[3] B. J. Bradel and T. S. Abdelrahman. Automatic trace-based parallelization of java programs. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 26, Washington, DC, USA, 2007. IEEE Computer Society.

[4] B. J. Bradel and T. S. Abdelrahman. The potential of trace-level parallelism in java programs. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 167–174, New York, NY, USA, 2007. ACM.

[5] B. Chan and T. S. Abdelrahman. Run-time support for the automatic parallelization of java programs. *J. Supercomput.*, 28(1):91–117, 2004.

[6] D. Chavarria-Miranda and J. Mellor-Crummey. An evaluation of data-parallel compiler support for line-sweep applications. *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 7–17, 2002.

[7] M. K. Chen and K. Olukotun. The jrpm system for dynamically parallelizing java programs. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 434–446, New York, NY, USA, 2003. ACM.

[8] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, and E. Bu. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, Dec 1996.

[9] IBM. Jikes research virtual machine. `http://jikesrvm.org/`, 2009.

[10] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of java grande benchmarks. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 72–80, New York, NY, USA, 1999. ACM.

[11] C. J. F. Pickett. Software speculative multithreading for java. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 929–930, New York, NY, USA, 2007. ACM.

[12] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 8–8, New York, NY, USA, 2001. ACM.

[13] R. Tefft and R. Lee. Reduction of complexity and automation of parallel execution through loop level parallelism. *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pages 304–308, Oct. 2007.

[14] J. Zhao, M. Horsnell, I. Rogers, A. Dinn, C. Kirkham, and I. Watson. Optimizing chip multiprocessor work distribution using dynamic compilation. In *Euro-Par 2007 Parallel Processing*, pages 258 – 267. Springer, 2007.