IMPLEMENTING PARALLELISM AND SCHEDULING DATA FLOW GRAPHS ON JAVA VIRTUAL MACHINE

Jin Xu

University of Notre Dame
Dept. of Comp. Science and Eng.
Notre Dame, IN 46556
jxu1@cse.nd.edu

Edwin H.-M. Sha

University of Texas at Dallas
Dept. of Comp. Science
Richardson, TX 75083-0688
edsha@utdallas.edu

ABSTRACT

In this paper, we present a scheme which explores the parallelism on JVM. An algorithm, called *dynamic-duplication scheduling* is developed for solving the static scheduling and code generation for data flow graphs (DFG) on the parallel JVM. Experimental results show that the schedule produced by the algorithm on parallel JVM is significantly improved compared with the traditonal JVM.

1. INTRODUCTION

Java is an Internet programming language that can run on a variety of platforms. Its popularity results from the fact that it is a machine-independent executable code with the stack-based Java Virtual Machine. Basicly, the current Java Virtual Machine runs sequentially without any parallel processing. The operation is executed by ”pushing” or ”popping” an element on the stack. Having only one operand stack in each method makes execution of programs slow. If we can use several operand stacks at the same time in each method, the standard interpretive implementation of the Java Virtual Machine will be improved.

Previous work has focused on efficient compilation to generate optimized stack code. An algorithm to generate optimal programs for expression evaluation is presented in [3]. Cierniak develops a research compiler for Java class files [4]. This compiler recovers a high-level structure from the class file and then applies optimizations obtained by applying data transformations to improve the locality of memory accesses before writing out the optimized code. Koopman [2] introduced stack scheduling which substitutes loads of local variables by stack copying and manipulation instructions. Maierhofer improves stack scheduling by adding an instruction scheduler in [5].

However, realizing parallelism of Java programs on the stack-based Java Virtual Machine has received relatively little attention. Most of the previous research did not take this into account. The main contribution of this paper is to explore the parallelism on the stack-based Java Virtual Machine, analyze the modification of system implementation which makes parallelism realistic, and build a system scheme for a parallel JVM. By applying list scheduling, we develops techniques and an algorithm to schedule control cycles based on the manipulation of cyclic data-flow graphs with delays on the parallel JVM. Our work provides a general solution to implement parallelism with multiple stacks and to optimize the schedule length of the DFG by dynamically adjusting the schedule at compile-time and perform global referencing for local variables.

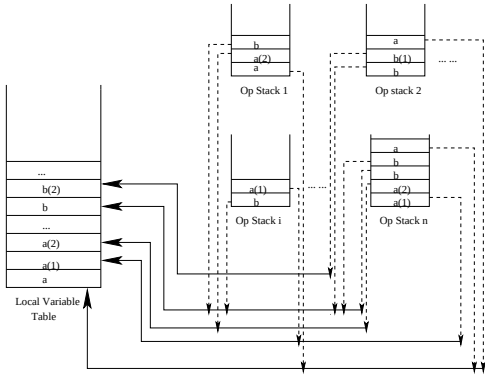

Fig. 1. Architectural Model of Parallel JVM

The rest of this paper is organized as follows. Section 2 presents the system specification of the parallel JVM. In Section 3, we discuss an algorithm to implement parallelism for the Java Virtual Machine. Experiment results are given in Section 4. Section 5 gives conclusions.

2. SYSTEM MODEL

2.1. Architecture Model

In the JVM, each method invocation has its own private operand stack, which is used to pass arguments to methods and to collect returned results. When there is a method invocation, local variables are used to store partial results and state information. When a method returns, any values in its local variables are discarded. Methods declare how many local variables they use. The operand stack of this method does operations by loading or storing values from local variables, which allows reentrancy and recursion.

In order to implement parallelism for each method in the JVM to improve speed for every method call, we design an architecture model with multiple operand stacks and a shared local variable table, shown in Figure 1. In our model, we make the following design:

Multiple operand stacks. Each method has multiple operand stacks instead of one. In this way, a task can be executed in differ-

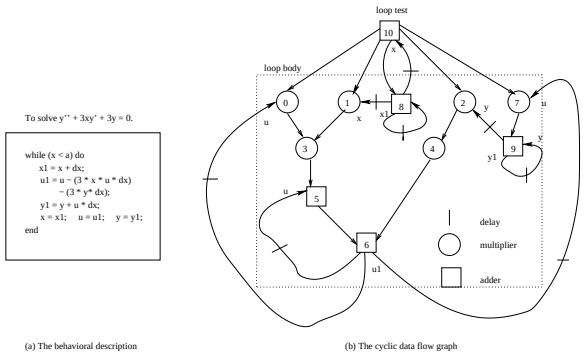

Fig. 2. The differential equation solver

ent operand stacks at the same time which can improve the speed of computation.

"Shared" local variables. We call these "shared" because all of operand stacks share local variables which means that the same local variable can be loaded into or stored from different stacks. In Figure 1, local variable *a* is shared by stack 1, stack 2 and stack n. If a local variable is changed in one stack, this stack will write this new value to the local variable table. Then all load instruction to this local variable will fetch the new value.

Multiple variables for different iterations. In a DFG, because there are different lengths of data dependency, we may need different versions of a node in one iteration. For the sake of simplicity, a node has different local variables to store its value in different iteration periods. At the end of each iteration, these values are updated to prepare for the next iteration.

2.2. System Scheme

Based on the parallel JVM model, we will show how to schedule a data flow graph (DFG) in this new model. In this paper, we use DFG to represent a task. Because an unlooped DFG is a special form of a looped DFG, the DFGs we discuss in this paper are looped DFGs. So our problem is: *Given a DFG, produce its static schedule in the parallel JVM.*

Definition 2.1 A data-flow graph (DFG) $G = (V, E, d)$ *is a node-weighted and edge-weighted directed graph, where* V *is the set of nodes,* $E \subseteq V \times V$ *is the set of edges and* d *is a function from* E *to the nonnegative integers.*

We use the differential equation solver in [7] as an example of using data-flow graphs to model while loops. The behavioral description is in Figure 2-(a). We assume that an adder performs additions, subtractions, and comparisons. For the DFG in Figure 2-(b), we use boxes to represent adder nodes and circles to represent multiplier nodes.

Our scheduling scheme has three phases:

Prework: In this phase, the main task is to find the number of local variables needed for each iteration. As we discussed earlier, a given node may need more than one local variable. This assignment is finished before the loop is executed, so all the iterations have the same variable allocation. In figure 2-(b), node 6, 8, 9, 10 each has 1 delay, so each of them needs 2 local variables to store its current value and the previous iteration value; For other nodes, each needs 1 local variable to store its current value.

Scheduling Period: This is the core of our system scheme. This phase schedules the DFG and arranges the order of each node. For a static schedule, we just need to find the schedule of one iteration, then repeatedly follow this schedule in each iteration. List scheduling is used in this phase to find the order of node execution. We propose an algorithm to arrange nodes to operand stacks.

Postwork: At the end of each iteration, local variables are updated. Each variable is forwarded to occupy the position of the previous iteration. If there is no variable allocated for a previous level, it is discarded and the position is left for the coming iteration. So in Figure 2, at the end of 2nd iteration, the local variable for current node 10 will be put in the local variable for node 10 of $1st$ iteration.

Of these three phases, the *scheduling period* and *postwork* are executed in a loop, while *prework* is performed outside of the loop.

3. DYNAMIC-DUPLICATION SCHEDULING

In this section, an algorithm called *dynamic-duplication scheduling* is provided to produce the static schedule on the parallel JVM model in the previous section.

Usually, memory accesses will take more time than other operations. If we store and load each variable every time, this process will cost a long schedule time. By stack copying and manipulation of instructions, the schedule can be improved. Koopman designed many kinds of stack manipulations in [1]. Most stack processors use a stack buffer to cache the topmost stack elements for improved performance [1]. In analogy to register machines, access to these stack elements is faster than access to memory [5]. This provides a basis for our improvement: the machine will keep copies of the elements on the stack and reuse these copies later, instead of loading them from main memory every time. But in earlier research, the use of duplication and the duplicated elements' positions in the stack are limited because it is hard to guarantee the proper working of tghe stack. In this paper, we discuss these issues and present a condition which can guarantee the correctness of duplication.

Property 3.1 *Given a DFG* $G = (V, E, d)$*, for* $v \in V$*, where* S_1 *is the sequence of stack elements after duplicating* v *and* S_2 *is the sequence of stack elements after loading* v*, the duplication of* v *is legal if* $S_1 = S_2$.

In order to guarantee that a duplication is legal, we must check the sequence of nodes on the stack when we want to copy a node. Each time we load a node, we will check to see if it is already loaded or computed in the selected stack. If we find it in the stack and the duplication can be legal, we make a copy and do not need to load it from memory. A duplication needs to be inserted into the previously produced schedule. The related store operation might be affected, so the schedule needs to be further changed. Because of this , we call this scheduling *dynamic duplication.*

We are also concern with the position of the duplicated element on the stack. For simplicity, we use the top element as the reference, and define the following duplication operation.

Definition 3.1 *In a stack* S*,* dup_k *is defined as an operation to duplicate an element to the* $(top + k)th$ *position on the stack and dup_k of node* v *is always put next to the loading or computing of* v.

Table 1. An example of dynamic duplication scheduling

$c = a + b$	(–)	iload_a
	(a–)	iload_b
	(b a–)	dup_3
	(b a () b–)	iadd
	(c () b–)	istore_c
$d = e + f$	(() b–)	iload_e
	(e () b–)	dup_1
	(e e b–)	iload_f
	(f e e b–)	iadd
	(d e b–)	istore_d
$h = e + g$	(e b–)	iload_g
	(g e b–)	iadd
	(h b–)	istore_h
$z = b + c$	(b–)	iload_c
	(c b–)	iadd
	(z–)	istore_z

Lemma 3.1 *Given a DFG $G = (V, E, d)$, dup_k of v is legal if $k = m + n$, m is the number of duplications inserted after loading or computing v in the previously produced schedule, n is the number of nodes obtained by other operations. These nodes satisfy two conditions:*

1. *They appear before v;*
2. *They are still present.*

Proof: The computation in the stack always pops the top elements. So v must appear as the top element when it is needed in a computation. According to dynamic-duplication, before duplicating v, suppose m duplications are inserted in the previously produced schedule after v on the stack. So when we insert the duplication of v, these duplications are still on the stack and they will be popped before duplication of v is popped. So the insertion of duplication be put behind all of them. Combined with n other nodes, the duplicaiton must be put at $k = m + n$, and this guarantee the sequence will not be destroyed, according to definition 3.1, this schedule is legal. □ Table 1 gives an example of how to determine k. In $c = a + b$, we duplicate it at $top + 3$ because we push a before we push b and according to the previous step of dynamic duplication, we know we need to leave a space for the copy of e which is produced in $h = e + g$. In order to simplify the implementation, in this paper, we do not consider m. Instead, we always copy a node at the bottom of the current stack. In this way, the duplication may be illegal. In such a case, we cannot make a copy so we reload this node.

Figure 3 (a) shows a DFG. For simplicity, there is only one operand stack. Figure 3 (b) is its schedule if there is no duplication operation, while Figure 3(c) is the schedule replacing *iload* instructions with *dup_b* instructions which means duplicate at bottom. If memory accesses take 2 cycles and other operations take 1 cycle, the latter will take 3 fewer cycles than the former. Figure 3(d) shows the contents in the operand stack in which 'C' means a constant. Further, the algorithm will do the following things:

- Because a constant has no predecessor, it should be put in the stack as early as possible. Every time when there is a constant in ancestors, this constant will be scheduled first.
- The algorithm will load ancestors according to their finishing positions which means it will load the earliest possible ancestor first.
- We do not store every node in each iteration because they may not be used in the later iterations. We just store two kinds of nodes:
  1. Nodes which will be used in other stacks in the current iteration;
  2. Nodes which will be used in the later iterations.

We add a checking process to check the store of nodes during the iteration. If a node is needed in a later iteration, it will be stored immediately after it is computed. If ancestors are computed on the different stack but not stored, we will insert store instructions into that stack. The choice of node is based on list scheduling. We design a stack scheduler (Algorithm1) to choose the operation stack. This scheduler always chooses the stack with the minimum number of cycle length from all stacks.

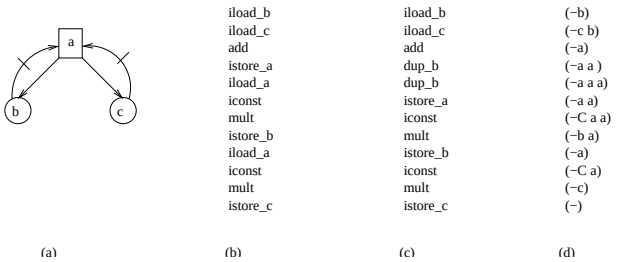

Fig. 3. A DFG graph & its schedule & its stack contents

Algorithm 1 Stack scheduler in dynamic-duplication

```
stack_scheduler(v,n)
 1: for i ← 1 to n do
 2:    Order ancestors by their finishing order, always put constants at beginning;
 3:    /*Decide load or duplication for ancestors*/
 4:    for all ancestors in finishing order do
 5:       if the ancestor v.an is in the stack; then
 6:          if duplication is legal then
 7:             insert dup;
 8:          else
 9:             load the ancestor;
10:          end if
11:       else
12:          Find finishing position p and finishing stack s of v.an
13:          if v.an is not stored then
14:             if v.an is used in stack s then
15:                add dup;
16:             end if
17:             store v.an at stack[s][p+1];
18:          end if
19:          add load;
20:       end if
21:    end for
22:    add operation;
23:    stack ← stack with the minimal row;
24:    /* Check if this stack will be used in the later iteration*/
25:    for all v -e-> u do
26:       if delay[u] ≠ 0 then
27:          store v;
28:       end if
29:    end for
30: end for
31: return stack;
```

4. EXPERIMENTAL RESULTS

In this section, we present some experimental results on the selected DSP benchmarks. In these experiments, we assume the function of each node is addition or multiplication. We compare

Table 2. Machine cycles for different algorithms

Benchmark	cycles						
	non-parallel	dynamic-duplication					
		2 stacks		3 stacks		4 stacks	
Diff.Equation	106	45	57.5%	35	67.0%	35	67.0%
2-stage IIR	142	53	62.7%	43	69.7%	32	77.5%
4-stage IIR	230	84	63.5%	52	77.4%	43	81.3%
Elliptic filter	314	139	55.7%	91	71.0%	88	72.0%
All-pole Lattice	138	50	63.8%	50	63.8%	50	63.8%
5th Elliptic	376	159	57.7%	119	68.4%	111	70.5%
Voltera	236	115	51.3%	70	70.3%	60	74.6%

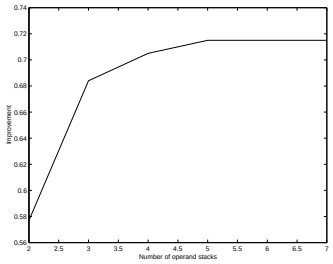

Fig. 4. Improvement vs. Number of operand stacks for $5th$ elliptic filter

the non-parallel schedule with the parallel dynamic-duplication algorithm.

The DSP filter benchmarks used in these experiments include a differential equation solver, a 2-stage IIR filter, an elliptic filter, an all-pole lattice filter, a 5th order elliptic filter and a volterra filter.

Table 2 summarizes the number of schedule cycles of non-parallel JVM and the parallel algorithms. These schedules are computed under the condition that loads and stores cost 3 machine cycles each and other operations cost 1 machine cycle. The percentage data represents the improvement compared with the non-parallel JVM schedule. For the dynamic-duplication algorithm, we measure the schedule when there are 2, 3 and 4 operand stacks. From this table, we get the following observations:

- The dynamic-duplication algorithm greatly improves the schedule on a traditional sequential JVM. The dynamic-duplication algorithm improves the schedule more than 60% in each benchmark, because the dynamic-duplication decreases the number of loads and stores which cost much execution time.
- The degree of parallelism depends on the properties of the DFG. Figure 4 shows the relationship between improvements obtained by dynamic-duplication and the number of operand stacks for $5th$ elliptic filter. This figure shows that the improvement will be increased when the number of operand stacks is less than 5, and it is the largest at 5 operand stacks, after that, there will be no gain when we increase the number of operand stacks. This is due to the number of nodes at each level in the DFG. According to the properties of the DFG, the number of operand stacks can be decided.

Table 3 shows the instruction. The instructions have been classified in three categories:

- **Loads/Stores** Contains all instructions accessing local variables.

Table 3. Instruction distribution for different algorithms

Benchmark	non-parallel			dynamic-dup		
	LS	dup	Other	LS	dup	Other
Diff.Equation	31	0	13	19	6	13
2-stage IIR	39	0	25	27	3	25
4-stage IIR	63	0	41	37	2	41
Elliptic filter	89	0	47	48	12	47
All-pole Lattice	39	0	21	11	8	21
5th Elliptic	104	0	64	55	10	64
Voltera	64	0	44	33	1	44
Total	429	0	255	230	42	255
Percentage	65.1%	0.0%	34.9%	43.6%	8.0%	48.4%

- **Duplication** Represents all duplication manipulations.
- **Others** Includes other instructions such as addition, multiplication and iconst.

In the non-parallel algorithm, 65.1% instructions are loads and stores, while in the dynamic-duplication algorithm, they account for only 43.6%. There is no duplication manipulation in non-parallel algorithm, but the amount is 8.0% in the dynamic-duplication. The total amount of instructions decreases greatly. The load and store instructions drop from 429 in the non-parallel JVM to 230 in the parallel JVM, with a slightly increase of duplication and no change in other instructions. This fact can account for the huge reduction in schedule length.

5. CONCLUSION

In this paper, we have presented a scheme to obtain parallelism on the JVM.The scheme includes replacing one operand stack with multiple operand stacks. All operand stacks share local variables. The values of the same node in different iterations are stored into different local variables. A scheduling algorithm is provided to generate the static schedule of DFG on the parallel JVM. On the basis of list-scheduling, the dynamic-duplication algorithm uses a series of methods to reduce the schedule length. The stack duplication which copies the stack elements forms a good way to reduce loads from local variables.

6. REFERENCES

[1] Philip Koopman, *Stack Computers-the new wave*, Mountain View press, 1989.

[2] Philip Koopman, *A Preliminary Exploration of Optimized Stack Code Generation.*, preceedings of the 1992 Rochester Forth Conference, Rochester, June 1992.

[3] Bruno,J. & Lassagne,T., *The Generation of Optimal Code for Stack Machines*, Journal of the ACM. Vol.22, No.3, July 1975.

[4] Michal Cierniak & Wei Li, *Optimizing Java Bytecodes*,February,1997, {http://www.cs.umd.edu/class/fall98/cmsc731/papers.html}.

[5] Martin Maierhofer & M.Anton Ertl, *Local Stack Allocation*, Compiler Construction, Springer LNCS, pages 189-203, 1998.

[6] Paulin, P. & Knight, J.P. *Force-directed Scheduling for the Behavioral Synthesis of ASIC's*, IEEE Transactions on Computer-Aided Design 8,6, June 1989, 661-679.