February 20, 2007

RT0709

Computer Science 21 pages

Research Report

A Data Partitioning Method Using Dynamic Data Dependence Graphs

Ryo Neyama and Mikio Takeuchi

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

A Data Partitioning Method Using Dynamic Data Dependence Graphs

Ryo Neyama and Mikio Takeuchi

IBM Research, Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan
{neyama,mtake}@jp.ibm.com

Abstract. While data partitioning significantly improves the scalability of multithreaded and clustered transaction processing (TP) systems, selecting a correct and effective partitioning criterion (i.e. how to partition data) requires deep insight into the target TP systems. In this paper, we propose a novel analysis method to find even non-intuitive partitioning criteria for TP systems using our tracing tool that generates *dynamic data dependence graphs (dynamic DDGs)*. Analyzing the exact behavior of a TP system from its dynamic DDG, our method can also generate a routing function of transaction requests for each partitioning criterion. We have demonstrated that our method could find the candidates of partitioning criteria and their routing functions, with a non-trivial TP system scenario.

1 Introduction

Demands on high performance of transaction processing middleware and its applications (transaction processing (TP) systems) [1] have been accelerated in recent years as more businesses in the World rely on computer systems. As Gordon Moore recently suggested [2], the speed of current CMOS processors is not guaranteed to keep exponential growth, therefore the future TP systems may meet CPU bottleneck depending on their workloads.

Recent trend in the enterprise server arena to overcome such CPU bottleneck is moving toward the scaling-out and the scaling-up approaches. The scaling-out approaches are represented by cluster servers and more recently blade servers [3], both of which offer high performance and fault tolerance at reasonable hardware cost. The scaling-up approaches are represented by symmetric multiprocessing (SMP) systems, which have been commoditized in these days, and multi-core processors. Major microprocessor vendors, such as Intel, IBM, Sun and AMD, are planning to release multi-core processors with many cores [4–7].

TP systems cannot always benefit from the recent scaling-out and scaling-up efforts by enterprise servers for free. Most TP systems are already multithreaded and already have a clustering capability for scalability. Even though, there are some cases in which increased number of servers or processors does not contribute to better performance. To make things worse, a TP system may even slow down with more servers or processors depending on their design.

Data contention is one of the major reasons why TP systems do not scale. This is because most TP systems must handle data with atomicity, consistency, isolation and durability (ACID) properties [8]. On the other hand, a recent high performance TP system usually caches data both at the server and the processor levels. Such a TP system also needs to guarantee cache consistency among servers as well as among processors.

Sharing updated data among servers causes a data synchronization overhead across local area network (LAN). Frequent data synchronization heavily consumes processor power and network bandwidth due to significant locking, cache invalidation and replication costs. Also, some TP systems may not be able to sufficiently use their processor power due to lock waiting time. Similarly, there is a data synchronization overhead among processors across processor interconnect along with locking and invalidation.

Data contention can be removed or reduced by appropriately applying *data partitioning* technique at the server level and *CPU affinity* technique at the processor level. With these techniques, frequently updated data can be localized to a server or a processor, while read-only or read-most data can be replicated among servers and/or can be accessed by multiple processors without or with less synchronization overhead.

For appropriate data partitioning, TP system developers need to carefully design TP systems, considering their characteristics of data access patterns. Namely, it is important to exploit the data locality based on their data access patterns. Performing such characterization, however, is not always straightforward especially in a complicated TP system that handles many types of data simultaneously. Furthermore, there can be a trade-off among partitioning criteria. For example, localizing customer data may result in sharing product data. In such case, how to partition data is not very clear.

Our work aims to make clear the data access patterns of TP systems, allowing TP system developer to easily define *partitioning criteria*. In addition, our future goal is to eventually automate such task. In this paper, we propose a novel analysis method to find even non-intuitive partitioning criteria for Java-based TP systems using our tracing tool, called *Samsara*, that generates *dynamic data dependence graphs (dynamic DDGs)*. Our method can also generate a *routing function* of transaction requests for each partitioning criterion. Partitioning criteria, dynamic DDGs and routing functions are described later in Section 2.

The rest of this paper is organized as follows. Section 2 discusses why we can benefit from tracing dynamic DDGs of TP systems. Section 3 gives the details of Samsara. Section 4 demonstrates data partitioning with a TP system scenario. Section 5 presents related work, and Section 6 offers conclusions.

2 Data Partitioning Using Dynamic Data Dependence Graphs

In this section, we first define a TP system model that uses data partitioning in Section 2.1. We also define a taxonomy of Java objects (objects, for short) based on dynamic DDGs, which is useful for data partitioning. We then describe the details of dynamic DDGs, by which we can generate routing functions as well as non-intuitive partitioning criteria, in Section 2.2. We finally consider how to find the candidates of partitioning criteria and their routing functions using the dynamic DDGs and the taxonomy in Section 2.3.

2.1 A Model of a Transaction Processing System That Uses Data Partitioning

Figure 1 depicts a model of a clustered TP system that uses data partitioning. Two types of data, shared data and partitioned (non-shared) data, are treated in this model. A TP system developer usually defines which non-shared data should be partitioned into which partition. We call this definition *partitioning criteria*.

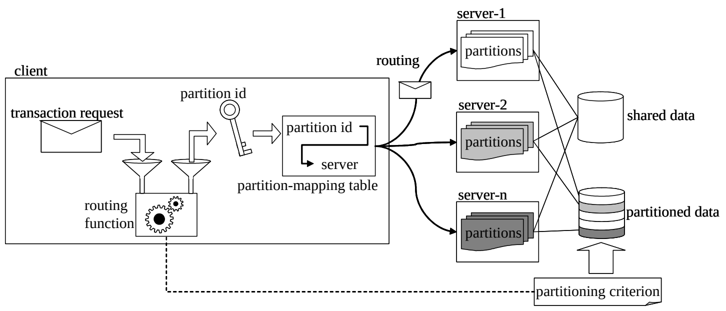


Fig. 1. A Model of a Transaction Processing System That Uses Data Partitioning

Determining an appropriate partitioning criterion in a complicated TP system is not a trivial issue. Our method helps TP system developers to determine partitioning criteria by offering the candidates of partitioning criteria as well as generating their routing functions (described later).

In the model, a transaction accesses a set of partitioned data that belongs to a single partition as well as any shared data in its transaction scope. In other words, each transaction request must be associated with at most one specific partition in a TP system.

A partition is identified with a *partition identifier (id)*. Each server manages multiple partitions and the sets of partitions managed by each server are disjoint. When a transaction request is associated with a partition, a client sends the transaction request to the server that manages the partition. To identify a partition from a transaction request, a client uses a *routing function*, which is specific to the partitioning criterion applied to a TP system. A routing function calculates a partition id from the values contained in a transaction request. To identify a server from a partition id, a client uses a *partition-mapping table* .

While how to map partitions to servers (how to define a partition-mapping table) is essential for load-balancing, we do not address this issue in this paper. Rather, we regard the partition-mapping table as a black box.

We classify the objects used in a TP system by their access patterns as shown in Table 1. This taxonomy aims to make a necessary and sufficient distinction of objects to discuss how to find partitioning criteria. An object is classified in this taxonomy based on how it is accessed by threads. A *read-only object* is an object that has never been written by any thread, while a *read-write object* is an object that has ever been written. A *non-shared object* is an object that is accessed by only one thread, while a *shared object* is ever accessed by at least two distinct threads.

	Read-only objects	Read-write objects
Non-shared objects	Potentially partitioned objects	
Shared objects	Potentially shared read-only objects	Potentially shared read-write objects

Table 1. Object Taxonomy by Access Patterns

2.2 Dynamic Data Dependence Graphs

In this section, we first define dynamic data dependence, and then define a dynamic DDG, which is used for offering partitioning criteria and generating routing functions later in Section 2.3.

Dynamic Data Dependence A *dependence* between two statements in a program is a relationship that constrains their execution order [9]. A *data dependence* is a constraint that arises from the flow of data between statements. There exist three types of data dependence: *true (flow) dependence*, *anti dependence* and *output dependence* (Figure 2). Each dependence represents an order constraint from a definition to a reference, from a reference to a definition, and from a definition to another definition, respectively.

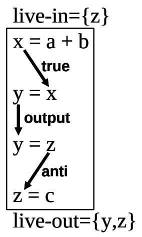


Fig. 2. Three Types of Data Dependence

A *dynamic data dependence* is a data dependence that actually arises from the execution of a program. Because it ignores potential data dependence from unexecuted paths, it is suitable to observe the behavior of a program from representative runs.

Dynamic Data Dependence Graphs with True-dependence Only Figure 3 shows how a dynamic DDG is constructed from an execution of a multithreaded Java program. A dynamic DDG is represented as a directed acyclic graph (DAG), where each node represents the definition of a value and each edge represents a reference to the value. Each node is color-coded by the thread id which defines the value.

In Figure 3, one thread $T1$ executes a Java statement `obj.count++`, then another thread $T2$ executes the same statement. The statement is compiled into Java bytecode (Figure 4).

At the time when the thread $T1$ executes the statement, the nodes 11 and 12 (a node is identified with the number specified between '<' and '>') are created in addition to the initial value node 8. The node 11 is created for a constant value '`1`' contained in the bytecode (line 4). The node 12 is created for the result of iadd (line 5). Thus, this dynamic DDG means "*the node 12 is created out of the nodes 8 and 11 by* iadd *at* `Simple1.exec()` *line 25*."

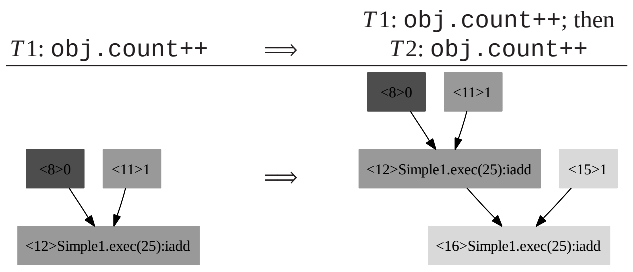


Fig. 3. A Dynamic DDG from an Execution of a Multithreaded Java Program

```
                      1:  aload obj
                      2:  dup
obj.count++   ⟹      3:  getfield count
                      4:  iconst 1
                      5:  iadd
                      6:  putfield count
```

Fig. 4. Compiled Java Bytecode

At the time when the thread $T2$ executes the statement, the nodes 15 and 16 are created. The node 15 is created for the same constant value as the node 11. Note that we can see the node 16 is created out of the node 12, which is created by another thread, and the node 15 from the dynamic DDG.

Full-functional Dynamic Data Dependence Graphs The dynamic DDG we have defined so far represents true dependence only. We also use anti- and output-dependence as well for more precise data partitioning. Therefore, we revise the definition of dynamic DDG so that we can know whether an object is read or written in which order and by which thread.

We will explain how to revise the definition of dynamic DDG in the rest of this section. For later use, we define the notation for data dependence: a read, a write, true dependence, anti dependence and output dependence (Figure 5).

$R_f(t)$:	a read of a field f by a thread t.
$W_f(t)$:	a write of a field f by a thread t.
$W_f(t1) \xrightarrow{t} R_f(t2)$:	$R_f(t2)$ depends on $W_f(t1)$ by true dependence.
$R_f(t1) \xrightarrow{a} W_f(t2)$:	$W_f(t2)$ depends on $R_f(t1)$ by anti dependence.
$W_f(t1) \xrightarrow{o} W_f(t2)$:	$W_f(t2)$ depends on $W_f(t1)$ by output dependence.

Fig. 5. Notation of Data Dependence

Now, let us consider the situation where a field of an object (denoted by f) is accessed by multiple threads as shown in Figure 6. We shall not treat array elements and static fields (although Samsara can treat them) in this section to simplify the discussion. Besides two true dependences ($W_f(T1) \xrightarrow{t} R_f(T2)$ and $W_f(T1) \xrightarrow{t} R_f(T3)$), there exist two anti dependences ($R_f(T2) \xrightarrow{a} W_f(T4)$ and $R_f(T3) \xrightarrow{a} W_f(T4)$) and one output dependence ($W_f(T1) \xrightarrow{o} W_f(T4)$).

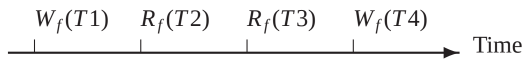


Fig. 6. A Set of Reads and Writes on f

For every reads and writes of a field, Samsara creates a new node to trace anti- and output-dependence. Samsara memorizes the latest write node followed by zero-or-more read nodes after the write for each field. When a field is written again by a thread, all the nodes in the set except for those created by that thread become the source of a new write node. The set is then cleared and the new write node is added to the set.

We define the notation of a node and a set as shown in Figure 7.

$N_{field}(f)$:	The current definition of a field f.
N_{object} :	The definition of an object reference on the stack in a read or a write of a field.
N_{value} :	The definition of a value on the stack in a write of a field.
$N_{access}(a)$:	A node that is created by an access a, where a is a read or a write of a field.
$\langle n1, n2, \ldots \rangle$:	A composite node that consists of $n1$, $n2$, $\ldots$.
$T(n)$:	The thread that creates the node n.
$S_f(k)$:	The set for the field f at time k.

Fig. 7. Notation of Node and Set

Now, the nodes for a read, a write and a set can be defined as follows:

$$\begin{aligned} N_{access}(R_f(t)) &:= \langle N_{object}, N_{field}(f) \rangle \\ N_{access}(W_f(t)) &:= \langle N_{object}, N_{value}, s_1, s_2, \ldots \rangle \qquad (s_i \in S_f(k) \land T(s_i) \neq t) \\ S_f(0) &:= \emptyset \\ S_f(k+1) &:= \begin{cases} \left\{ N_{access}(W_f(t)) \right\} & \text{for } W_f(t) \\ S_f(k) \bigcup \left\{ N_{access}(R_f(t)) \right\} & \text{for } R_f(t) \end{cases} \end{aligned}$$

In the example in Figure 6, during the time after $R_f(T3)$ and before $W_f(T4)$, the set $S_f(k)$ is expressed as follow:

$$S_f(k) = \left\{ N_{access}(W_f(T1)), N_{access}(R_f(T2)), N_{access}(R_f(T3)) \right\}$$

When $W_f(T4)$ writes a value defined by a node v, $N_{access}(W_f(T4))$ and $S_f(k+1)$ are expressed as follows:

$$\begin{aligned} N_{access}(W_f(T4)) &= \langle N_{object}, v, N_{access}(W_f(T1)), N_{access}(R_f(T2)), N_{access}(R_f(T3)) \rangle \\ S_f(k+1) &= N_{access}(W_f(T4)) \end{aligned}$$

By looking into $N_{access}(W_f(T4))$ on the resulting dynamic DDG, we can find anti- and output-dependence.

2.3 Data Partitioning

Figure 8 illustrates the usage scenario of our tools, Samsara and *Partitioning Criteria Finder (PCF)*. The scenario consists of the following three steps: 1) A TP system developer first launches the target TP system under the control of Samsara and then examines some representative transactions that sufficiently cover the actual behavior of the TP system. Tracing the representative transactions, Samsara produces a dynamic DDG; 2) Analyzing the dynamic DDG, PCF offers the candidates of partitioning criteria with the corresponding routing functions to him/her; 3) He/she determines a partitioning criterion that is even valid with his/her knowledge of the TP system and then applies the partitioning criterion and its routing function to the TP system.

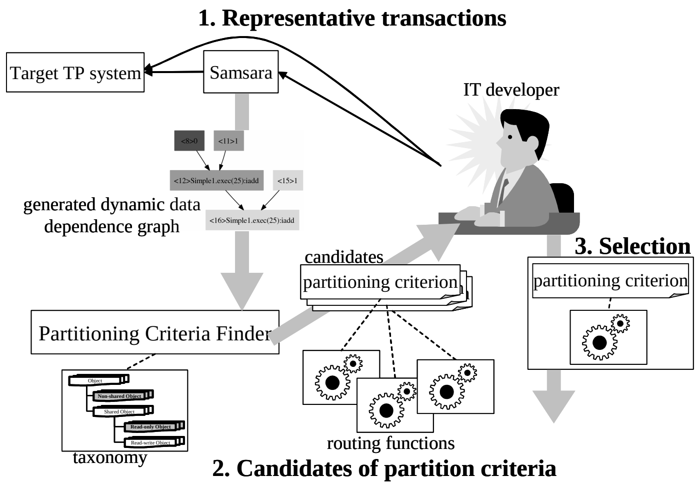

1. Representative transactions
Target TP system
Samsara
IT developer
generated dynamic data dependence graph
candidates
partitioning criterion
3. Selection
partitioning criterion
Partitioning Criteria Finder
taxonomy
routing functions
2. Candidates of partition criteria

Fig. 8. Usage Scenario

To simplify the discussion, we would like to assume that a transaction begins at the entry of a specific method, and commits at the exit of the same method. Samsara traces dynamic data dependence during the method call. A transaction request is a list of arguments for the method.

As is described in Section 1, there can be a trade-off among multiple partitioning criteria. It is not always possible to completely partition all the data in a TP system. Under such trade-off relationship, some data need to be shared among servers whatever the partitioning criterion is. We, therefore, assume the use of a distributed lock manager [10] with the target TP system so that it can guarantee the consistency even if all the data cannot be completely partitioned.

By applying an appropriate partitioning criterion, overall data locality of a TP system is increased. As a server can hold an ownership for a lock across transactions, we can effectively reduce the cost of distributed locking for partitioned data. Furthermore, if a TP system developer can guarantee that some data are partitioned with a partitioning criterion, he/she can remove the cost of distributed locking for the data by explicitly replacing the distributed lock with a local lock.

In general, database keys (keys, for short) that a transaction uses are determined by the result of some calculation using the arguments in a transaction request, the states that the server internally holds, and the data obtained from the backend database by queries.

Our target of data partitioning is the keys that can be calculated from the arguments in a transaction request and the read-only server states. The routing function we generate is used to calculate a partition id from the arguments in a transaction request and the read-only server states. We can capture the keys from representative transactions by using the knowledge of the configuration and the application programming interface (API) of a TP system middleware, or by analyzing SQL statements issued.

A client needs to hold a replica of the read-only server states on which the routing function depends. There may exist a case where the client cannot afford to hold read-only server states because of their size. In this case, the client can get a partition id from the transaction request by delegating the execution of the routing function to one of the servers. There is no locking overhead to the server for executing the routing function since the server states it requires are read-only.

We shall exclude the keys that are calculated depending on the read-write server states and/or the data obtained from the backend database from the candidate partitioning criteria because we cannot generate routing functions for such keys. It might still be possible to partition the non-partitioned data associated with such key if the key depends on other keys that are partitioned by a partitioning criterion if the non-partitioned data depends on such partitioned keys.

An argument in a transaction request or a read-only server state may or may not be used for calculating the keys. We need to know which part of the arguments and the server states will actually be used to calculate the keys for generating routing functions. It is however not obvious.

In a more complicated scenario, the combinations of values, which are completely different each other, of more than one transaction requests may result in the equivalent keys. We determine the code location where the values that are well-calculated for choosing a key are defined, defining such a set of values as a partition id and the set of data identified by the partition id as a partitioning criterion.

How can we possibly determine such well-calculated values? If the calculation of partition id is not sufficient, two different transaction requests that eventually read or write the equivalent keys may be recognized as different partitions. If the calculation of partition id is too much, clients need to pay unnecessary cost for the calculation. Moreover, it may make the calculation unnecessarily depend on the read-only server states.

Our method allows a client to define a partition id even from a transaction request that is not executed as a representative transaction by abstracting the calculation to define a partition id. We focus on multiple transactions that defines the equivalent key. We find the common and uncommon part of the calculations that affect the selection of the key among such transactions. We use the uncommon part as a routing function and define the set of the output values of the uncommon part as a partition id.

Specifically, we traverse backward the definition of the key from the key itself to the arguments while each separate value that affects the definition of the key is equivalent among the multiple transactions. We consider the most upward values that satisfy such equivalence as an *n*-dimensional vector. We finally define this *n*-dimensional vector as a partition id. Two separate *n*-dimensional vectors are equivalent if every corresponding elements are equivalent.

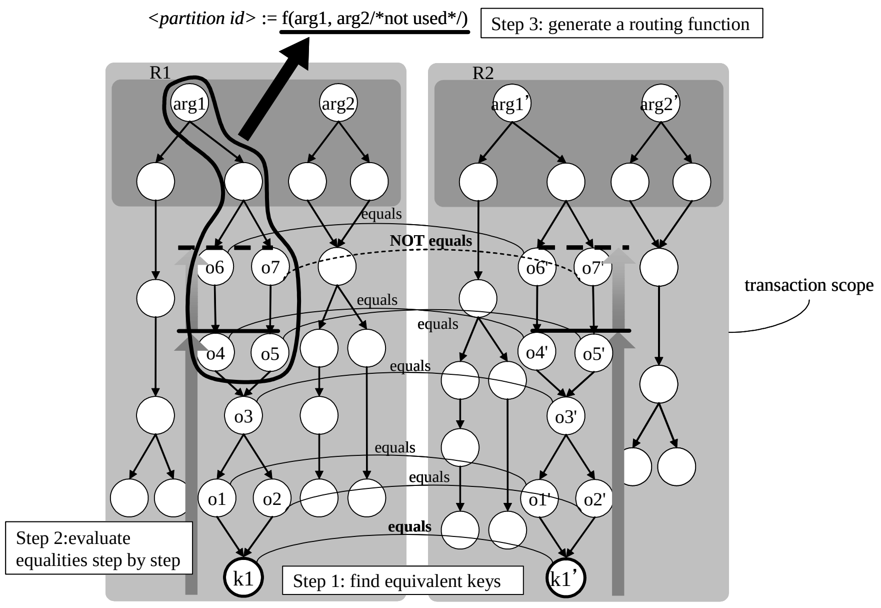
In the step 1 of the scenario shown in Figure 8, the TP system developer executes a variety of representative transactions. The representative transactions should contain multiple transactions that read or write the equivalent key. When tracing dynamic data dependence, for every dominators in the method, Samsara records all the live values at their entries so that PCF can later check the equality of the values.

PCF uses the '`==`' operator for primitive values and the `equals()` method for objects respectively to check the equality. When recording values, PCF writes a primitive value as it is while for objects it writes only the instance fields that are necessary for invoking the `equals()` method, and the unnecessary instance fields are filled with `0` or `null` values. By not writing unnecessary fields, PCF can avoid writing values that are not suitable for logging (e.g. a file descriptor).

The algorithm of PCF consists of the following three steps:

1. Find the multiple transactions that access the equivalent key from a dynamic DDG.
2. Find an appropriate partition id by traversing the paths through which the key is defined.
3. Generate a routing function.

Let us look into the details of the algorithm of PCF. We first consider the simplest case, a single key is accessed by the set of two transactions (Figure 9). Note that we can easily extend this algorithm to the set of three-or-more transactions.



<partition id> := f(arg1, arg2/*not used*/)
Step 3: generate a routing function
R1
R2
arg1
arg2
arg1'
arg2'
equals
NOT equals
o6
o7
o6'
o7'
transaction scope
equals
equals
o4
o5
o4'
o5'
equals
o3
o3'
equals
equals
o1
o2
o1'
o2'
equals
Step 2:evaluate equalities step by step
k1
Step 1: find equivalent keys
k1'

Fig. 9. Algorithm of PCF (1)

PCF finds two transactions ($R1$ and $R2$) that access the equivalent keys ($k1$ and $k1'$) by looking into the dynamic DDG in the step 1. In general, such set can consist of more than two transactions. In addition, it is possible that more than one sets exist for different keys. When each transaction in a set accesses more than one keys and they have multiple equivalent keys, PCF adopt the key with which associated data is the most frequently written according to the profiling of the TP system. We reduce the cost of distributed locking as much as possible by partitioning such frequently written data.

At the step 2, PCF traverses backward the definition of the key from the keys ($k1$ and $k1'$) to the arguments ($arg1$, $arg2$, $arg1'$ and $arg2'$). The gray arrows in the figure indicate these backward traversals.

The equivalence of corresponding live values at the entries of every common dominators is evaluated from the lowest common dominator. The values of the nodes $o1$–$o6$ in $R1$ and their corresponding nodes $o1'$–$o6'$ in $R2$ are all equivalent respectively.

PCF continues this backward traversal while all the corresponding live values are equivalent, and finally stops at the highest common dominator that satisfies the equivalence of the live values. At the level of $o6$ and $o7$, $o7$ is not equal to $o7'$, although $o6$ is equal to $o6'$. Thus PCF stops the backward traversal at the level of $o4$ and $o5$.

PCF then defines the n-dimensional vector that is comprised of the live values as the partition id for these representative transactions. The two-dimensional vectors of $o4$ and $o5$, and their corresponding $o4'$ and $o5'$, are the partition ids in this case.

At the step 3, PCF finally generate a routing function as a *static backward program slice* with respect to the partition id. The routing function takes a list of arguments as its input and generates a partition id as an n-dimensional vector.

Every instruction to calculate the partition id is added to the routing function in the order of their appearance in the dynamic DDG. The generated routing function usually starts from the accesses to the arguments and ends with the definition of a partition id. We use the result of control-flow analysis for the target TP system to add branch instructions to the routing function (Figure 10). PCF generates the guard code to terminate the routing function to avoid executing unexecuted paths. If the routing function evaluates a conditional branch then branches to an unexecuted path, it immediately terminates the calculation of partition id with a special error code. We can avoid an unexpected behavior of routing function such as an exception with this guard code.

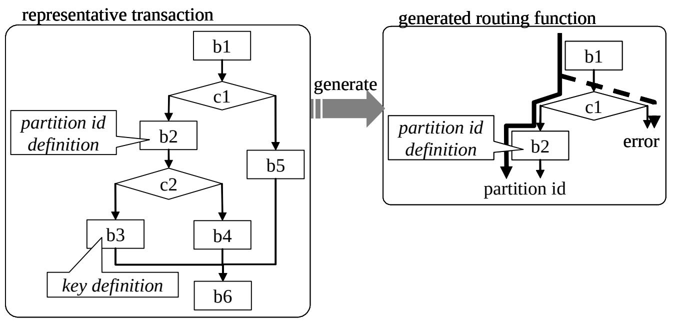


Fig. 10. Considerations on Conditional Branches

We secondly consider the case where the two distinct sets of equivalent keys appear at the same code location (Figure 11). *R*1 and *R*2 define a set of equivalent keys at a code location while *R*3 and *R*4 define the other set of equivalent keys at the same code location. It is not guaranteed that the backward traversal at the step 3 reaches the same common dominator in this case, namely either one may stop the backward traversal earlier. We adopt the lowest common dominator to make the most abstracted routing function with broader coverage of input arguments. We adopt the routing function `f()` rather than `g()`. We use the lowest common dominator among multiple sets in general. Even though a client needs to pay extra cost to calculate a partition id, we can reduce wrong partitioning as much as possible by expanding the coverage of the acceptable arguments.

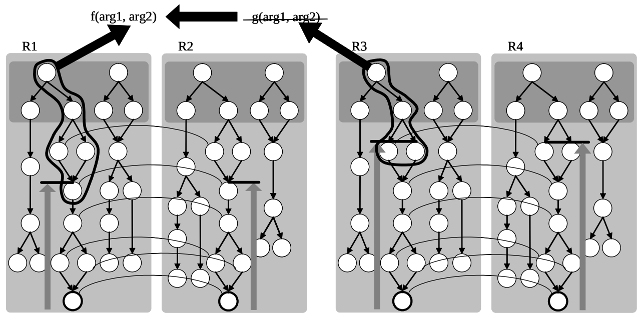


Fig. 11. Algorithm of PCF (2)

We also discuss the case where the two distinct sets define equivalent keys respectively at the different code locations by conditional branches (Figure 12). The set of *R*1 and *R*2, and the set of *R*3 and *R*4, define the keys in the different code locations. If both sets can be merged into a common dominator on the way of the backward traversal, then we adopt the lowest common dominator for defining a partition id so that we can expand the coverage as well. If both sets cannot be merged into a common dominator, that is, if either one or both of which stop the backward traversal within their specific branches, we adopt both as the partition id definitions and generates two routing functions. In this case, the client needs to call each of them one by one until a partition id is generated without any error.

The routing function may modify its arguments on the way of calculating a partition id, although it would actually be a rare case. If the routing function does not modify the arguments, then the client passes the arguments of a transaction request to the routing function. Otherwise, it copies the arguments of the transaction request and passes them to the routing function. Usually the arguments are not very big objects, and the overhead of copying the arguments is not a big issue in that case.

When determining a partitioning criterion, the TP system developer needs to be aware of the fact that the representative transactions may not reflect all the behavior of the TP system. There is a case where some actually shared objects in the dynamic DDG can be seen as if they were non-shared depending on the representative transactions. In that case, the resulting candidates of partitioning criteria may include wrong

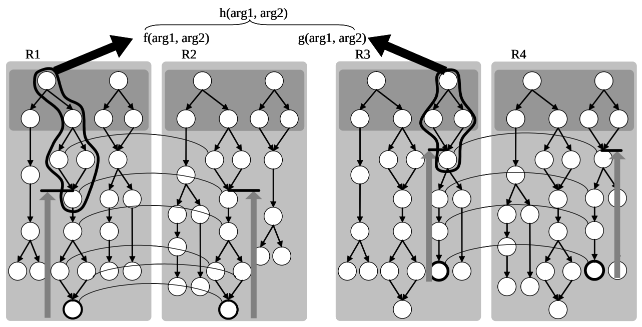


Fig. 12. Algorithm of PCF (3)

partitioning criteria. As we assume the use of distributed lock manager in the target TP system, the consistency of data is not broken even if it uses a wrong partitioning criterion. Rather, it just slows down.

Static backward program slice [11] can provide the information that is useful for the his/her decision. More specifically, even if there is a conditional branch that has never been executed in the representative executions but there is possible use of the object in it, Samsara can remind him/her of that.

When the data is partitioned, there needs to be a way of identifying a partition from a transaction request at the client side for routing the transaction request to the target process of TP system that manages the partition. As our dynamic DDG holds all the calculation process to reach each value in a transaction, we can extract the routing function by which a partition id can be calculated from the values contained in the transaction request. There is no impact on a TP system when the algorithm depends only on the transaction request. Even if it depends on the internal states of the TP system, we can replicate the states to the client side if such states are read-only objects. If it were not for dynamic DDGs, it would be quite difficult to automatically determine such routing algorithm.

There is a restriction in Samsara. As Samsara only traces what happens within a single Java VM process, a dynamic DDG normally loses some information when the TP system reads data from or writes data to the outside world, which typically happens due to the TP system's disk or network I/O. This seems to be a big restriction when we do not have any assumptions on TP systems. We, however, can mitigate this restriction by making some reasonable assumptions.

For representative executions, we should use a single-server version of the TP system so that we can avoid losing the information by communication among servers. All the configurations that are loaded from local storage or via network should be statically loaded by initialization threads at the start-up time. Otherwise, Samsara treats the configurations dynamically loaded from the outside world, which can be considered as read-write objects in conservative understanding. Thus, it prevents Samsara from partitioning data on such paths.

Another approach to mitigate the restrictions is to give Samsara the knowledge of the TP system. In general, a TP system consists of a transaction processing middleware and its applications. The applications are typically restricted their behavior especially regarding their access to the system resources. Instead, they usually use an API that the middleware provides. With such assumption, Samsara can assert that the outgoing data does not affects the behavior of the application using the knowledge on the API.

One of the major concerns on losing the information to the outside world is the existence of a backend database for the TP system. We, however, can trace the behavior, when the TP system itself has in-memory database as the front-end of the backend database and the in-memory database allows Samsara to trace its behavior. Current trend in the TP system arena, where transactional cache and in-memory database are used, can support this approach.

3 How Samsara Works

This section explains the details of Samsara. It includes how we achieved the portability of Samsara using JDI.

3.1 Mirroring Java VM

Unlike other Java-based dynamic data dependence tools, Samsara does not require a modification of Java VM, and thus it is portable to any Java VMs. Samsara dynamically captures data dependence of target Java programs by emulating the execution of target programs at bytecode instruction level using JDI events. Samsara does not use byte-code instrumentation except for accessing the array indexes, which is needed due to the current functional limitation of the Java Debug Interface (JDI) [12] on which Samsara depends. JDI is a high-level Java programming interface to build debugger applications for the Java 2 Platform, Standard Edition (J2SE) [13].

Samsara includes so-called a *mirroring Java VM* that mirrors the state of the target Java VM running the target Java program. The state mirrored by the mirroring Java VM is the definition of data, rather than the data itself which an ordinary Java VM traces.

Dependence Within a Frame Figure 13 shows how mirroring Java VM mirrors the state of the target Java VM for tracing dependence within a frame. While the target Java VM traces values on a stack, Samsara's mirroring Java VM traces definitions on a stack.

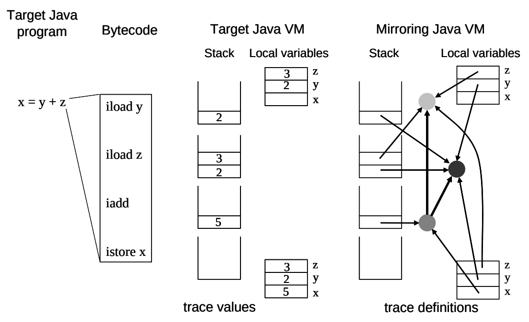


Fig. 13. Dependence Within a Frame

In Figure 13, the target Java program calculates y + z and assigns the result value into x. The Java program is compiled into a sequence of bytecode instructions: iload y, iload z, iadd and istore x. These instructions load the values of local integer variables y and z onto the stack, pops the top-two integer values from the stack, add them, pushes the result value to the stack, and then stores the stack top integer value into the local variable x.

The mirroring Java VM, on the other hand, understands the semantics of each bytecode instruction from the viewpoint on how values are defined. The mirroring Java VM first pushes the definition of the local variable y and z onto the stack, then pops the top-two definitions from the stack, creates a new definition that refers to the two definitions and pushes it onto the stack, and finally stores the new definition to the local variable x. Note that the definitions and the references to them form a DAG, that is a dynamic DDG generated in this execution.

Dependence Across Frames Data dependence is also generated by calling methods and by throwing exceptions. On calling methods, a caller may pass arguments to its callee, and the callee may return a return value to the caller. Thus, when the callee defines some value using its arguments and when the caller defines some value using the return value, new data dependences are generated. Note that the this object is also passed as the "zeroth" argument.

Figure 14 explains how Samsara traces data dependence passed through arguments and a return value of a method call. In this scenario, the caller calls the method add(a,b) with the arguments y and z (plus this), then assigns the return value to x. The callee returns a + b, which is equivalent to y + z in the caller's frame, to the caller.

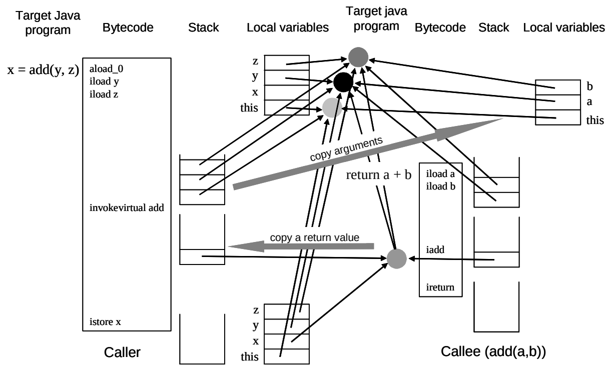


Fig. 14. Dependence by Method Calls

Receiving the event of method invocation instruction, the mirroring Java VM copies the definitions of the arguments, y and z, on the caller's stack onto the local variables of the callee and removes the arguments from the caller's stack. Similarly, receiving the event of return-from-method instruction, the mirroring Java VM copies the definition of the return value (a + b) on the callee's stack onto the caller's stack.

For native methods which do not have bytecode, mirroring Java VM cannot trace dependence at bytecode instruction level. If the callee is a native method, mirroring Java VM treats its return value as if it only depends on its arguments assuming that the callee does not access data other than its arguments.

Some bytecode instructions implicitly or explicitly throw exceptions. We call such instruction *potentially-excepting instruction (PEI)*. For example, a division instruction like idiv throws an exception when divisor value is zero. Such exception may be caught and referred by exception handlers, therefore we need to trace data dependence of exception objects. When a exception handler defines a new value out of the caught exception, a new data dependence is created.

Data dependence of an exception object depends on its excepting instruction and its exception class. In case of division instruction, it is caused by the fact that the divisor value is zero. Thus, it is considered that the dependence comes from the divisor value. For each thread, Samsara memorizes the causes of all potential exceptions before executing every PEIs, and when an exception is actually thrown, Samsara sets references to the memorized causes to the thrown exception.

Dependence Across Threads When a reference to an object is stored in the fields of other objects or classes (i.e. the object is *escaped* from the origin thread [14]), the other threads may refer to the object and they can define another value using that object. Samsara traces this kind of data dependence across threads as well.

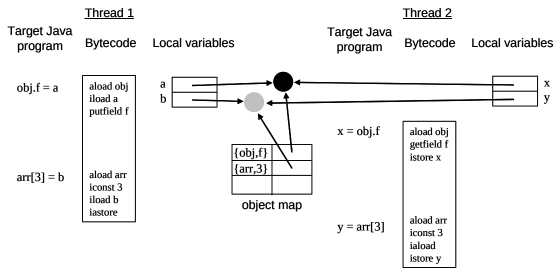


Fig. 15. Dependence Across Threads

Figure 15 depicts how Samsara traces the data dependence generated across threads. Unlike other data dependence tools, Samsara distinguishes the different instances of the same class. Samsara manages an object map that associates the key of an object field or an array element with its definition. An object field is specified as a pair of an object identifier and a field identifier. An array element is identified as a pair of an array object identifier and an index value. When a thread stores a value to an object field or an array element, Samsara puts the definition to the object map with an appropriate key. When another thread loads a value from the object field or the array element, Samsara gets the definition from the object map with the same key. Thus, a definition created by some thread can be referred by another thread. We made bytecode instrumentation to runtime libraries and target Java programs, however this is due to the lack of JDI's capability to capture array access with an array object and an array index. If the future specification of JDI is changed to send a special event for array access or provide an access to the operand stack of a stack frame, then Samsara does not require bytecode instrumentation.

Samsara Architecture Figure 16 depicts the architectural details of Samsara. The target Java VM is executed in the debug mode as if they were debugged with a JDI-compliant debugger, thus there is no need for additional changes for them.

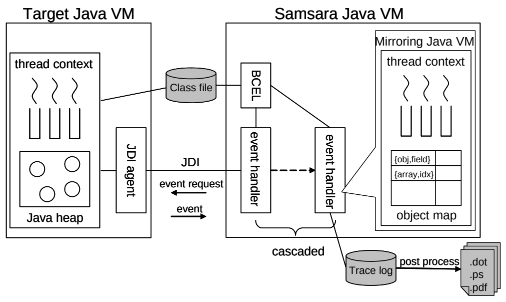


Fig. 16. Architecture of Samsara

Samsara has an event handler cascading architecture, where multiple event handlers can be chained. In practice, this chain always ends with the mirroring Java VM for data dependence tracing. Other event handlers in front of the mirroring Java VM are used for *separation of concern* on event sequences. Actual usage of these event handlers are 1) starting and stopping event requests using the knowledge on the target TP system; 2) emulating missing events due to vendor specific bugs in JDI implementation; 3) event logging for debugging Samsara itself, etc.

Samsara needs to read the same class files as the target Java VM uses. This is because current JDI does not provide an access to the constant pool of class definitions. This limitation will be mitigated in Java 6, which allows a debugger to access to constant pools [15]. Samsara looks into the class file using the Byte Code Engineering Library (BCEL) [16]. By post-processing the trace log of Samsara, dynamic DDGs can be visualized with Graphviz [17].

4 An Example of Data Partitioning Using Our Method

We proposed a novel data partitioning method that is capable of generating routing functions as well as offering partitioning criteria by using dynamic DDGs in Section 2. In this section, we will demonstrate the effectiveness of our data partitioning method with a simple but non-trivial TP system scenario. We could verify that our method could successfully define a partition id, generating the corresponding routing function with our analysis method through this example scenario.

The example scenario is a simple stock exchange scenario, where clients send order transaction requests to buy stocks. A client specifies the symbol that represents the stock he/she wants to buy with its price to the stock exchange server (sx-server).

The source code of the sx-server is listed in Figure 17. There are two methods: `initialize()` and `exec()`.

The method `initialize()` initializes the state of the sx-server. It is called only once at the start-up time exclusively by a single thread, namely no other thread begins any transaction until the sx-server completes the initialization. Samsara memorizes the

```
 1  void initialize() {
 2      requests[0] = "foo,120";
 3      requests[1] = "foo,150";
 4
 5      symbol2codeTable = new HashMap();
 6      symbol2codeTable.put("foo", new Integer(123));
 7      symbol2codeTable.put("bar", new Integer(456));
 8
 9      inMemoryDB = new HashMap();
10      List resultSet = new ArrayList();
11      resultSet.add("foo-key1");
12      resultSet.add("foo-key2");
13      inMemoryDB.put(new Integer(123), resultSet);
14      resultSet = new ArrayList();
15      resultSet.add("bar-key1");
16      resultSet.add("bar-key2");
17      inMemoryDB.put(new Integer(456), resultSet);
18  }
```

```
 1  void exec(int threadId) {
 2      String request = requests[threadId];
 3
 4      String symbol = request.substring(0, request.indexOf(',''));
 5      Integer dbKey;
 6      synchronized (symbol2codeTable) {
 7          dbKey = (Integer) symbol2codeTable.get(symbol);
 8      }
 9      synchronized (inMemoryDB) {
10          List resultSet = (List) inMemoryDB.get(dbKey);
11          resultSet.add(request);
12      }
13  }
```

Fig. 17. Java Source Fragments

initial state of the sx-server by tracing this particular thread. This knowledge is later used for recognizing shared read-only and read-write objects.

The method first creates two requests for later use for the transaction threads (lines 2 and 3). It also initializes the in-memory database (lines 9–17) as well as the symbol-to-code table that maps a stock symbol to its code (lines 5–7). Two stock symbols are defined ("`foo`" and "`bar`"), associating with their code ("`123`" and "`456`") (lines 6 and 7). The code are later used to query on the in-memory database. It is intended that the symbol-to-code table is to be a shared read-only table and the in-memory database is to be a shared read-write object.

The method `exec()` is called by the threads that execute transactions. The transaction begins at the entry of the method and commits at the exit of the method. The method first gets the request for the thread (line 2). It then parses the request to get the symbol, calling the methods `indexOf()` and `substring()` (line 4). The symbol is then used as the key for the symbol-to-code table to retrieve the code (line 7). The code is used as a database key (key) to query a result set from the in-memory database (line 10).

Our first goal is to find an appropriate partition id from the dynamic DDG that is generated by executing the sx-server. Our second goal is to generate a routing function, defining an appropriate partition id.

Figure 18 shows the result of our analysis method. The upper graph shows a whole dynamic DDG generated by executing two transaction threads that request "`foo,120`" and "`foo,150`" respectively. Our analysis method has extracted the paths from the requests to the keys (the definitions of the keys) from the whole dynamic DDG as shown in the lower graph, which is the subgraph of the whole dynamic DDG.

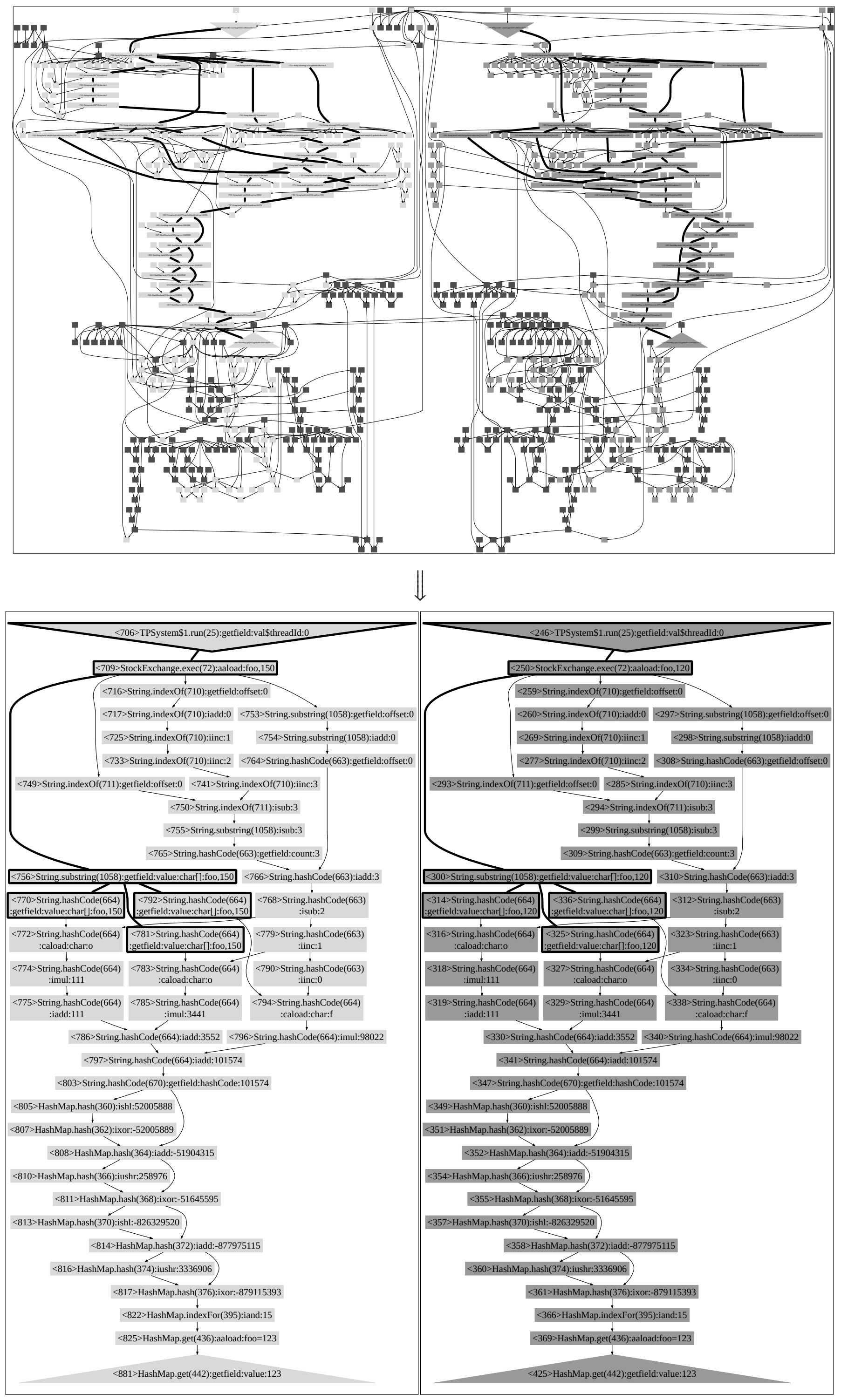

⇓
<706>TPSystem$1.run(25):getfield:val$threadId:0
<709>StockExchange.exec(72):aaload:foo,150
<716>String.indexOf(710):getfield:offset:0
<717>String.indexOf(710):iadd:0
<753>String.substring(1058):getfield:offset:0
<725>String.indexOf(710):iinc:1
<754>String.substring(1058):iadd:0
<733>String.indexOf(710):iinc:2
<764>String.hashCode(663):getfield:offset:0
<749>String.indexOf(711):getfield:offset:0
<741>String.indexOf(710):iinc:3
<750>String.indexOf(711):isub:3
<755>String.substring(1058):isub:3
<765>String.hashCode(663):getfield:count:3
<756>String.substring(1058):getfield:value:char[]:foo,150
<766>String.hashCode(663):iadd:3
<770>String.hashCode(664) :getfield:value:char[]:foo,150
<792>String.hashCode(664) :getfield:value:char[]:foo,150
<768>String.hashCode(663) :isub:2
<772>String.hashCode(664) :caload:char:o
<781>String.hashCode(664) :getfield:value:char[]:foo,150
<779>String.hashCode(663) :iinc:1
<774>String.hashCode(664) :imul:111
<783>String.hashCode(664) :caload:char:o
<790>String.hashCode(663) :iinc:0
<775>String.hashCode(664) :iadd:111
<785>String.hashCode(664) :imul:3441
<794>String.hashCode(664) :caload:char:f
<786>String.hashCode(664):iadd:3552
<796>String.hashCode(664):imul:98022
<797>String.hashCode(664):iadd:101574
<803>String.hashCode(670):getfield:hashCode:101574
<805>HashMap.hash(360):ishl:52005888
<807>HashMap.hash(362):ixor:-52005889
<808>HashMap.hash(364):iadd:-51904315
<810>HashMap.hash(366):iushr:258976
<811>HashMap.hash(368):ixor:-51645595
<813>HashMap.hash(370):ishl:-826329520
<814>HashMap.hash(372):iadd:-877975115
<816>HashMap.hash(374):iushr:3336906
<817>HashMap.hash(376):ixor:-879115393
<822>HashMap.indexFor(395):iand:15
<825>HashMap.get(436):aaload:foo=123
<881>HashMap.get(442):getfield:value:123
<246>TPSystem$1.run(25):getfield:val$threadId:0
<250>StockExchange.exec(72):aaload:foo,120
<259>String.indexOf(710):getfield:offset:0
<260>String.indexOf(710):iadd:0
<297>String.substring(1058):getfield:offset:0
<269>String.indexOf(710):iinc:1
<298>String.substring(1058):iadd:0
<277>String.indexOf(710):iinc:2
<308>String.hashCode(663):getfield:offset:0
<293>String.indexOf(711):getfield:offset:0
<285>String.indexOf(710):iinc:3
<294>String.indexOf(711):isub:3
<299>String.substring(1058):isub:3
<309>String.hashCode(663):getfield:count:3
<300>String.substring(1058):getfield:value:char[]:foo,120
<310>String.hashCode(663):iadd:3
<314>String.hashCode(664) :getfield:value:char[]:foo,120
<336>String.hashCode(664) :getfield:value:char[]:foo,120
<312>String.hashCode(663) :isub:2
<316>String.hashCode(664) :caload:char:o
<325>String.hashCode(664) :getfield:value:char[]:foo,120
<323>String.hashCode(663) :iinc:1
<318>String.hashCode(664) :imul:111
<327>String.hashCode(664) :caload:char:o
<334>String.hashCode(663) :iinc:0
<319>String.hashCode(664) :iadd:111
<329>String.hashCode(664) :imul:3441
<338>String.hashCode(664) :caload:char:f
<330>String.hashCode(664):iadd:3552
<340>String.hashCode(664):imul:98022
<341>String.hashCode(664):iadd:101574
<347>String.hashCode(670):getfield:hashCode:101574
<349>HashMap.hash(360):ishl:52005888
<351>HashMap.hash(362):ixor:-52005889
<352>HashMap.hash(364):iadd:-51904315
<354>HashMap.hash(366):iushr:258976
<355>HashMap.hash(368):ixor:-51645595
<357>HashMap.hash(370):ishl:-826329520
<358>HashMap.hash(372):iadd:-877975115
<360>HashMap.hash(374):iushr:3336906
<361>HashMap.hash(376):ixor:-879115393
<366>HashMap.indexFor(395):iand:15
<369>HashMap.get(436):aaload:foo=123
<425>HashMap.get(442):getfield:value:123

Fig. 18. Dynamic DDG with the StockExchange Scenario

The left-hand side nodes of the whole dynamic DDG were most created by the thread of "`foo,150`," while the right-hand side nodes were most created by the thread of "`foo,120`." The bottom nodes in the dynamic DDG were mostly created by the initialization thread.

The two downward triangles were the nodes that represent the requests, that is, the thread ids passed as the argument for the method `exec()`. The two upward triangles were the keys. The paths from the requests to the keys are specified by the bold edges. The nodes on the paths are specified by the bigger rectangle in order to distinguish them from other nodes that are irrelevant to the definition of the keys. It might be observed that the paths resemble each other despite the fact that each threads were executed independently. The dynamic DDG revealed that the calculation of the keys only depend on the requests and the shared read-only objects.

The lower subgraph that were extracted using our analysis method shows the details of how the keys have been defined. As is discussed in Section 2, we traversed backward the dynamic DDG from the keys to the requests, evaluating the equivalence between the corresponding nodes. Note that each node displays its own value on the label. For example, the node 706 asserts that an integer value zero is obtained by executing getfield. Also, the node 709 asserts that a string value "`foo,150`" is obtained by aaload.

As the result of the backward traversal, our analysis method could find the definition of a partition id and its corresponding routing function. Our method discovered the path specified with the bold edges and the nodes with bold sides were the nodes that were not equivalent to their corresponding nodes. Other nodes had exactly the same values with their corresponding nodes. Thus, the vectors that consist of the nodes 250, 300, 314, 325 and 336, and the nodes 709, 759, 770, 781 and 792, can be defined as partition ids. The routing function can be defined by extracting every instruction to calculate those nodes. The routing function calculates partition ids from transaction requests.

5 Related Work

Redux [18] is a tool that generates dynamic dataflow graphs of target programs at machine instruction level. It is implemented as a skin of Valgrind [19] and works with program binaries, and thus is not restricted to programs written in any particular language. However, for Java programs which are represented in bytecode, it generates dataflow graphs of the Java VM that interprets the target Java programs instead of the graphs of the target Java programs themselves. It is not straight-forward or almost impossible to understand dependences in Java programs from the instruction level trace of Java VM.

Umemori et al. [20] propose a dynamic program slicing technique called *dependence cache (DC) slicing* which combines a dynamic data dependence analysis with a static control dependence analysis. Their implementation uses aspect-oriented programming (AOP) with AspectJ [21] to instrument code for collecting dynamic data dependence, thus it requires an access to the source code of the target Java programs. Samsara works with Java bytecode (class files) and does not require an access to their source code.

Nishiyama [22] proposes an improved dynamic escape analysis method with read-barrier technique. Although its original goal is to reduce the number of memory locations that must be checked at runtime for detecting data races, it effectively eliminates the same conservativeness of conventional escape analysis as we addressed with Samsara in the sense that it captures only objects that are actually accessed by multiple threads. Our method can not only classify the objects used in a TP system by their access patterns, but also find the candidates of partitioning criteria and their routing functions using its dynamic DDG. Unlike his dynamic analysis which is implemented as a modification of HotSpot Java VM [23], our tool *Samsara* is portable to any Java VM.

Coign [24] is an automatic distributed partitioning system for binary applications built from distributable COM components. It detects the application's inter-component communication through scenario-based profiling. Then it applies a graph-cutting algorithm to partition the application across a network to minimize execution delay due to network communication. Unlike Coign, our method focuses on partitioning data across servers and/or processors to optimize the cost of data access.

6 Conclusions

We have proposed a novel analysis method that allows a TP system developer to find even a non-intuitive partitioning criterion by generating a dynamic DDG for a target Java-based TP system. We used dynamic DDGs to see the calculation of the database keys as well as to recognize data access patterns such as non-shared and shared read-only objects. Using the data access patterns, our method has enabled to discover the database keys that could be used for data partitioning. While escape analysis method also could exhibit data access patterns, our method have enabled to generate routing functions that calculate partition ids as well, taking advantage of a dynamic DDG.

We are planning to measure the performance of a TP system in more realistic scenario. We assumed that there exists a single entry point of transactions in the target TP system. We will extend our method so that it can allow multiple entry points as well. Samsara traces the dynamic data dependence within a single Java VM. We will extend Samsara to trace dynamic data dependence across the Java VMs, and more generally any processes.

7 Acknowledgements

We would like to thank the members of the Business Computing Platform group of Tokyo Research Laboratory for their support and valuable comments.

References

1. Jim Gray and Andreas Reuters. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.
2. Gordon E. Moore. No exponential is forever: But "forever" can be delayed! In *Proceedings of the 2003 IEEE International Solid-State Circuits Conference (ISSCC '03)*, pages 20–23, 2003.

3. IBM Corporation. IBM BladeCenter. http://www.ibm.com/systems/bladecenter/.
4. Intel Corporation. Multi-core from Intel. http://www.intel.com/multi-core/.
5. IBM Corporation. Cell Broadband Engine resource center. http://www.ibm.com/developerworks/power/cell/.
6. Sun Microsystems, Inc. UltraSPARC T1. http://www.sun.com/processors/UltraSPARC-T1/.
7. Advanced Micro Devices, Inc. AMD multi-core. http://multicore.amd.com/.
8. Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2002.
9. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.
10. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.
11. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
12. Sun Microsystems, Inc. Java debug interface (JDI). http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/.
13. Sun Microsystems, Inc. Java 2 platform, standard edition (J2SE). http://java.sun.com/j2se/.
14. Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 1–19, 1999.
15. Bug Database. Bug ID 5024104: Allow access to constant pool in classtype. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5024104, 2004.
16. Apache Software Foundation. The byte code engineering library. http://jakarta.apache.org/bcel/.
17. Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233, 2000.
18. Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV '03)*, 2003.
19. Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV '03)*, 2003.
20. Fumiaki Umemori, Kenji Konda, Reishi Yokomori, and Katsuro Inoue. Design and implementation of bytecode-based Java slicing system. In *Proceedings of the 3rd International Workshop on Source Code Analysis and Manipulation (SCAM '03)*, pages 108–117, 2003.
21. The Eclipse Foundation. AspectJ project. http://www.eclipse.org/aspectj/.
22. Hiroyasu Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM '04)*, pages 127–138, 2004.
23. Sun Microsystems, Inc. Java HotSpot technology. http://java.sun.com/products/hotspot/.
24. Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation (OSDI '99)*, pages 187–200, 1999.