IMPROVING SCALABILITY OF JAVA ARCHIVE SEARCH ENGINE THROUGH RECURSION CONVERSION AND MULTITHREADING

Oscar Karnalim
Program of Information Technology, Faculty of Information Technology,
Maranatha Christian University, Bandung 40164, Indonesia
Email: oscar.karnalim@it.maranatha.edu

***Abstract*—Based on the fact that bytecode always exists on Java archive, a bytecode based Java archive search engine had been developed [1, 2]. Although the system is quite effective, it still lacks of scalability because many modules applying recursive calls and the system only utilizes one core (single thread). In this research, Java archive search engine architecture is redesigned in order to improve its scalability. All recursions are converted into iterative forms although most of these modules are logically recursive and quite difficult to convert (e.g., Tarjan's strongly connected component algorithm). Recursion conversion can be conducted by following its respective recursive pattern. Each recursion is broken down to four parts (before and after actions of current and its children) and converted to iteration with the help of caller reference. This conversion mechanism improves scalability by avoiding stack overflow error caused by method calls. System scalability is also improved by applying multithreading mechanism which successfully cut off its processing time. Shorter processing time may enable system to handle larger data. Multithreading is applied on major parts which are indexer, vector space model (VSM) retriever, low-rank vector space model (LRVSM) retriever, and semantic relatedness calculator (semantic relatedness calculator also involves multiprocess). The correctness of both recursion conversion and multithread design are proved by the fact that all implementation yield similar result.**

Keywords: **Scalability; Recursion Conversion; Multithreading; Java Archive Search Engine; Multiprocess.**

I. INTRODUCTION

Scalability is a prominent factor in a search engine development since the size of index data the search engine may grow rapidly [3]. Many techniques are applied to handle data growth such as algorithm optimization, preprocessing, multithreading, and parallelism.

Received: March 22, 2016; received in revised form: March 28, 2016; accepted: March 30, 2016; available online: April 4, 2016.

Based on the fact that bytecode always exists on Java archive, a bytecode based Java archive search engine had been developed [1]. The system utilizes bytecode on Java archive as its primary information source and extracts many textual information on class files as document terms through reverse engineering mechanism (e.g., class name, field name, method name, control flow weighted string literals in method content, and method calls). Its retrieval model is also embedded with relatedness in order to improve its recall [2]. Although the system works well, it still lacks of scalability since many modules are applying recursive calls and the system is single threaded.

Since recursion generates many function calls and each function call is pushed on stack to keep the track of the program flow [4], recursive calls may yield stack overflow error in Java environment. Stack overflow error occurs when memory stored in JVM stack is larger than its size [5]. This error can be avoided by converting all recursive algorithms to its iterative form. However, some algorithms are logically recursive and inconvenient enough to design it on iterative manner (e.g., Tarjan algorithm for detecting strongly connected components on graph [6]). Although logically recursive algorithms are exceptional cases where recursive approach is more beneficial than iterative approach in software development [7], these algorithms are still needed to be converted in order to avoid stack overflow error.

As hardware technologies advances, many regular computers are backed up with multi core processors which enable operating system to complete many task at a time using multithreading [8]. Liu & Wang [9] applied multithreading on their ensemble learning in

order to filter spam in Short Message Services (SMS). Their multithread design runs faster than single thread design which strengthens the proof that multithread may reduce processing time. This mechanism may also be utilized to reduce time latency in many search engine tasks such as indexing and retrieving documents [10–12].

In this research, a bytecode based java archive search engine is designed in more scalable way which involve recursion conversion and multithreading. Three main recursive algorithms which are converted to its iterative form are loop encapsulation, recursive method elimination, and method expansion. These algorithms are needed at indexing phase in order to extract method contents. The search engine architecture is also redesigned as multithread search engine to diminish its processing time.

II. METHODS

A. Recursion Conversion

It is known that all algorithms can be implemented recursively or iteratively [13]. Some algorithms are more convenient to be implemented with iterative approach whereas the others are more convenient to be designed as recursive one. Many nave and straightforward algorithms such as sequential searching, sorting, and string processing are easier to be implemented iteratively. However, some permutation and combinatorial problems are easier to solve recursively (e.g., map coloring, insertion on binary tree, and traversing all nodes in graph).

Recursive algorithms converted in this research are logically recursive. Loop encapsulation is designed similar to Depth First Search (DFS) pattern and utilizes Tarjans Strongly Connected Component (SCC) algorithm for loop detection. Recursive method elimination also utilizes similar Tarjan's algorithm to detect recursive methods although they do not share the same data type. Therefore, object oriented technique called Generics is applied in order to enable Tarjan's SCC algorithm takes various data types on its process without explicit typecasting. Method expansion is designed in recursive dynamic programming manner which never re-expand already-expanded method to cut off its processing time. Since stack overflow error is the biggest issue in recursive implementation, especially on large scale recursion [4, 5]. Therefore, all recursive implementation in this research will be converted as iterative one to improve the program scalability.

For clarity at conversion phase, some recursive parts of algorithm in both implementation are marked with different color which details can be seen in Table I. Abbrevations are also given for each description to simplify the illustration.

B. Loop Encapsulation

Loop encapsulation encapsulates loops based on Miecznikowski's algorithm [1, 14]. Loop encapsulation example can be seen in Fig. 1. This module takes control flow graph as its argument and utilizes Miecznikowski's to detect loop candidates. All nodes which is a member of certain loop are wrapped and merged as one loop node. Loop node is a wrapper which consists of loop member nodes and inherits its successors / predecessors. This loop replacement mechanism is conducted repeatedly from inner-most loops until no more loop exists.

Recursive implementation of loop encapsulation module can be seen in Alg. 1. Control Flow Graph (CFG) is represented as array of Control Flow (CF) nodes which will be broke down as a bunch of sub-

TABLE I
RECURSIVE PARTS COLOR DETAILS.

Color	Description
Blue	Current process before processing its children (B)
Red	Current process after processing its children (A)
Green	Current children's process before recursion (CB)
Brown	Current children's process after recusion (CA)

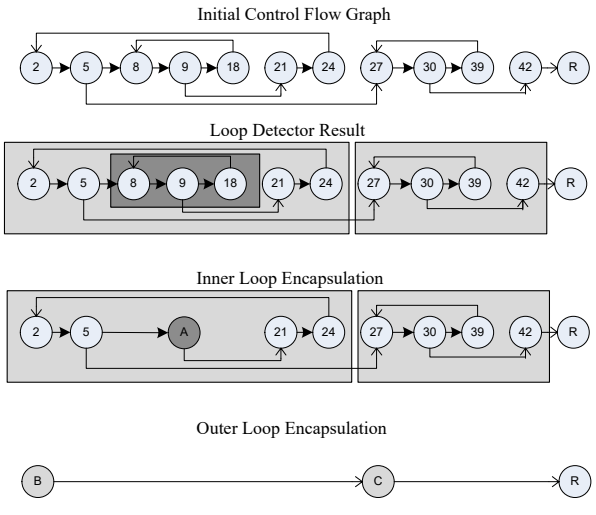

Fig. 1. Loop encapsulation example.

graph based on Tarjan's algorithm (`getSCC` function). Each subgraph which consists of more than one node is considered as loop candidates and its respective conditional node is removed. Removing conditional nodes is required to detect inner loop on remaining nodes by executing similar mechanism recursively.

Converting loop encapsulation to iterative form is quite simple since this algorithm follows simple DFS pattern (without A and CA parts). Simple DFS pattern can be remodeled to iterative form with the help of a stack and a while loop (which are used to keep track of recursive calls). Iterative loop encapsulation can be seen in Alg. 2 where each recursive call is replaced by pushing its recursive argument on a stack. B and CB are translated as loop action whereas each element on stack is popped and processed until the stack is empty.

C. Recursive Method Elimination

Recursive method elimination removes all recursive methods from method expansion candidates since all recursion may yield endless method expansion (each expansion consists of additional recursive call). This module takes method list as its argument, augments it as method graph, detects recursion candidates, and removes them from method expansion candidates. Method graph is augmented from method calls where each method is converted to a node and a method call is converted as an edge from caller to called method. Since this module only apply recursion on its Tarjans SCC algorithm, This module will not be discussed further.

Algorithm 1 Recursive Loop Encapsulation.

```
procedure encapsLoopR(nodeList :CF[])
| CF[][] sccList= getSCC(nodelist)
| for each scc in sccList
| | if(scc.length > 1)
| | | encapsulate scc as loop node
| | | detectLoopType(scc)
| | | removeConditionalNodes(scc)
| | | encapsulateLoopR(scc)
| | end if
| end for
end procedure
```

Algorithm 2 Iterative Loop Encapsulation.

```
procedure encapsLoopI(nodeList : CF[])
| Stack s
| s.push(nodeList)
| while(s is not empty)
| | CF[] tmp = s.pop()
| | CF[][] sccList = getSCC(tmp)
| | for each scc in sccList
| | | if(scc.length > 1)
| | | | encapsulate scc as loop node
| | | | detectLoopType(scc)
| | | | removeConditionalNodes(scc)
| | | | s.push(scc)
| | | end if
| | end for
| end while
end procedure
```

D. Tarjan's SCC Algorithm

Since loop encapsulation and recursive method elimination involve SCC detection on their graph respectively, Tarjan's SCC algorithm is also converted to iterative form. Tarjan's SCC algorithm applies Generic which enables this module to take various data types on its process without explicit typecasting. Generics is a Java feature which enables developer to use same class for many kind of data types without using explicit typecasting [15].

Tarjans SCC algorithm is logically recursive which is quite difficult to convert it as iterative algorithm. This algorithm also has all recursive algorithm parts which declared at Table I (B, A, CB, and CA). Tarjans recursive implementation can be seen in Alg. 3. This algorithm can be conducted by calling `getSCC`. Index and lowlink of each node are used to detect SCC wherein -1 stands for unprocessed nodes. `Index` variable in this algorithm is considered as global variable.

Despite of its iterative conversion difficulty, this algorithm still can be converted by imitating its recursive patterns with the help of caller reference. Tarjan's iterative implementation can be seen in Alg. 4. This implementation follows some rules which are:

- Each data tuple is encapsulated as IterTuple which has an additional field called caller reference. Caller reference is used to keep track of its recursive pattern and to perform its "after recursion" part (A and CA).
- Procedure call of `constructSCC` is replaced by a loop which perform similar pattern as `constructSCC`.
- Current process is stored in `cur` and each recursive call is replaced by assigning cur with its recursive process (which is similar to let this process visit its child first).
- A and CA parts are conducted after all of its successor are "recursively" processed. A part is conducted first before CA part since CA part conducted in this phase is its parent process's CA (parent process is accessed using caller reference).

E. Method Expansion

Method expansion unwrap all method encapsulation by replacing all method calls with its respective method contents until no more method call exists (which is quite similar to method inlining [16]). This

Algorithm 3 Recursive Tarjan's SCC Algorithm.

```
function getSCC(nodeList:T[]) : T[][]
| T[][] sccList; Stack s; int index = 0
| for each node in nodeList
| | if(node.index = -1)
| | | constructSCC(node, sccList, s, index, nodeList)
| | end if
| end for
| return sccList
end function
procedure constructSCC(node : T, sccList : T[][], s : Stack, index : int, nodeList : T[])
| node.index = index; node.lowlink = index
| stack.push(node); index = index + 1
| for each successor suc of nodein nodeList
| | if(suc.index == -1)
| | | constructSCC(suc, sccList, stack, index, nodeList)
| | | node.lowlink = min(suc.lowlink, node.lowlink)
| | else if(suc.onStack)
| | | node.lowlink = min(suc.index, node.lowlink)
| | end if
| end for
| if(node.index = node.lowLink)
| | T[] scc; T n
| | do{
| | | n = stack.pop();
| | | scc.add(n)
| | }while(node != n)
| | sccList.add(scc)
| end if
end procedure
```

mechanism is applied to reweight terms since terms used in frequently called methods should have greater weight based on its occurences. Each inserted method content is also weighted by its prior method call weight in order to keep its relevancy to its caller method. Since recursive calls may yield unlimited loop during expansion phase, recursive methods are marked and only expanded at N times (N is defined as a parameter at indexing phase).

Method expansion applies dynamic programming to speed up its process. Each method is only expanded once and all unexpanded method which is called in method content is expanded first. This module is initially implemented with recursive approach because of its natural recursive logic. Recursive implementation of method expansion can be seen in Alg. 5 where its iterative form can be seen in Alg. 6. Conversion of this module imitates Tarjans SCC algorithm conversion where each tuple is encapsulated as IterTuple with an additional field caller reference. Although copying all term in method with `nID` to `mtt` is considered both A and CA parts, this instruction still can be converted by duplicating its instruction (The first one is for A part whereas the second one is for CA part).

F. Multithread Design

It is obvious to state that single thread program in multi core processor is not efficient enough since it only utilizes one core. To utilize all cores, program must apply multithreading mechanism which let program to do tasks at many different cores at the same time. Therefore, program developed in this research is redesigned to multithread manner in order to cut off its processing time. Two major parts are redesigned which are indexer and retriever. Indexer part conversion is quite simple since its jobs can be split based on documents (Java archives). Retriever part involves two retriever mechanism which are VSM and LRVSM. Unlike indexer part, this module requires some global calculation after multithreading to yield similar result as sequential one. In addition, semantic relatedness calculator between term pairs is also redesigned in multithread manner since semantic relatedness is required at EVSM retriever.

G. Multithread Indexer

Since document-partitioned indexes are more frequently used than term-partitioned indexes in most search engines [17], multithread indexing developed in this research are based on document-partitioned indexes. Documents (Java archives) are split and indexed separately using many threads at the same time. Because document-partitioned indexes assures that all indexes are not tightly-coupled to each other, erroneous indexes only affect the indexed documents and the re-index phase will not affect remaining indexes. Therefore, it may yields faster index error correction.

Cite this article as: O. Karnalim, "Improving Scalability of Java Archive Search Engine Through Recursion Conversion and Multithreading", CommIT (Communication & Information Technology) Journal 10(1), 15–26, 2016.

Algorithm 4 Iterative Tarjan's SCC Algorithm.

```
function getSCC(nodeList : T[]): T[][]
| T[][] scc
| int index = 0
| Stack stack
| for each node in nodeList
| | if(node.index == -1)
| | | IterTuple cur
| | | cur.node = node; cur.caller = null
| | | cur.node.index = index; cur.node.lowlink = index
| | | stack.push(cur); index = index + 1
| | | while(true)
| | | | if(cur.node has unprocessed successors which is nodeList member)
| | | | | IterTuple next
| | | | | next.node = cur.node.getNextUnprocessedNode(); next.caller = cur
| | | | | if(nodeList.contains(next.node))
| | | | | | if(next.node.index == -1)
| | | | | | | next.node.index = index; next.node.lowlink =index
| | | | | | | stack.push(next); index = index + 1
| | | | | | | cur = next
| | | | | | else if(next.node.onStack)
| | | | | | | cur.node.lowlink = min(cur.node.lowlink, next.node.index)
| | | | | | |end if
| | | | | else
| | | | | | if(cur.node.index = cur.node.lowlink)
| | | | | | | T[] scc; IterTuple top
| | | | | | | do{
| | | | | | | | top = stack.pop()
| | | | | | | | scc.add(top.node)
| | | | | | | }while(top.node.index != cur.node.index)
| | | | | | | sccList.add(scc)
| | | | | | end if
| | | | | | IterTuple caller = cur.caller
| | | | | | if(caller != null)
| | | | | | | caller.node.lowlink = min(caller.node.lowlink, cur.node.lowlink)
| | | | | | | cur = caller
| | | | | | else break
| | | | | | end if
| | | | |end if
| | | |end while
| | end if
| end for
| return sccList
end function
```

Documents are partitioned based on greedy load-balance mechanism which can be seen in Alg. 7. The number of initialized stacks is similar with the number of expected jobs wherein each document is placed on the lowest document size stack at that time. This mechanism intends to distribute documents evenly among all jobs with greedy approach. Although greedy approach does not always yield the best result, it may yield fairly good distribution among all jobs at linear time. Greedy approach is extremely faster than brute force approach which complexity is O(N!).

Multithread indexer design can be seen in Fig. 2. All documents are listed and distributed based on greedy load-balance algorithm. Since each job yields an index based on its given documents, this mechanism will result many indexes rather than one. These indexes need not to be merged as one since split indexes may yields faster index error correction with the help of distribution list. Distribution list is a comma separated values file that contain document list for each job. This file is also generated at indexing phase. Any weighting schemes conducted in previous sequential design are delayed to retriever part based on these reasons:

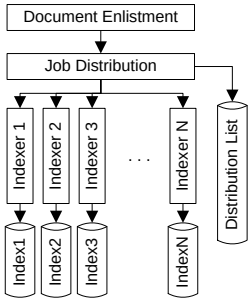

Fig. 2. Multithread Indexer Design.

- Storing raw indexes is more scalable than processed one since various weighting schemes may

Cite this article as: O. Karnalim, "Improving Scalability of Java Archive Search Engine Through Recursion Conversion and Multithreading", CommIT (Communication & Information Technology) Journal 10(1), 15–26, 2016.

Algorithm 5 Recursive Method Expansion Algorithm.

```
function expand(beforeList : Data) : Data
| Data afterList
| for each method with id curID in beforeList
| | methodExpansion(curID, beforeList, afterList)
| return afterList
end function
procedure methodExpansion(curID : MethodID, beforeList : Data, afterList : Data)
| if(afterList does not contain method with curID)
| | ExpandedMethod mtt
| | ScoreTuple[] sList = get score tuple list of method with curID from beforeList
| | for each ScoreTuple st in sList
| | | if(st is string literal)
| | | | mtt.add(st)
| | | else // st is method call
| | | | MethodID nID = st.getMethodID()
| | | | if(beforeList contains method with nID) // need to be expanded
| | | | | if(afterList does not contain method with nID) // not expanded yet
| | | | | | methodExpansion(curID, beforeList, afterList)
| | | | | end if
| | | | | copy all term in method with nID to mtt// also included as CA part
| | | | end if
| | | end if
| | end for
| | afterList.add(mtt)
| end if
end procedure
```

Algorithm 6 Iterative Method Expansion Algorithm.

```
function expand(beforeList : Data) : Data
| Data afterList
| for each method with id curID in beforeList
| | methodExpansion(curID, beforeList, afterList)
| return afterList
end function
procedure methodExpansion(curID : MethodID, beforeList : Data, afterList : Data)
| if(afterList does not contain method with curID)
| | IterTuple cur
| | cur.id = curID; cur.caller = null
| | cur.sList = get score tuple list of method with cur.id from beforeList
| | while(true)
| | | if(cur.sList has unprocessed ScoreTuple)
| | | | ScoreTuple st = cur.sList.getNextUnprocessedScoreTuple()
| | | | if(st is string literal)
| | | | | cur.mtt.add(st)
| | | | else// st is method call
| | | | | MethodID nID = st.getMethodID()
| | | | | if(beforeList contains method with nID) // need to be expanded
| | | | | | if(afterList does not contain method with nID) // notexpanded yet
| | | | | | | IterTuple next
| | | | | | | next.id = nID; next.caller = cur
| | | | | | | next.sList = get score tuple list of method with next.id from beforeList
| | | | | | | cur = next
| | | | | | else
| | | | | | | copy all term in method with nID to cur.mtt
| | | | | | endif
| | | | | endif
| | | | endif
| | | else
| | | | afterList.add(cur.mtt)
| | | | if(cur.caller != null)
| | | | | insert all term in method with cur.id to cur.caller.mtt at its respective pos
| | | | else break
| | | | end if
| | | end if
| | end while
| end if
end procedure
```

Algorithm 7 Greedy Load-Balance Algorithm.

```
function splitJob(path : String, n : int)
| Stack[] jobs = new Stack[n]
| for each file f in path
| | if(f is Java archive)
| | | minIdx = get lowest sJobs index
| | | jobs[minIdx].add(f)
| |end if
| end for
end function
```

be applied without re-indexing.
- Index error correction may be simplified since it only focuses on errorneous indexes and its respective documents.
- Many additional indexes can be embedded during retrieval phase without modifying preexisting indexes.

H. Multithread Retriever

In this research, two retrieval models are redersigned in multithread manner. These retrieval model are standard and low-rank vector space model. Low-rank VSM (LRVSM) is extended VSM that utilizes semantic relatedness in order to improve its recall. VSM is selected since this model is a benchmark of many other retrieval models whereas LRVSM is the most effective extended VSM found in previous research [2, 3]. Both retrievers takes directory path which consists of all indexed files as its input and build in-memory retrieval models.

I. Multithread VSM

On VSM retriever, multithreading is conducted at reading indexes and retrieving documents. Both tasks assume each index as one job and each job is handled by one retriever (N indexes = N jobs = N retriever). Multithread VSM index reader can be seen in Fig. 3. Each index is listed and read separately in different jobs. After all indexes are read, their respective terms are weighted using tf-idf scoring and stored in separate retrievers. Multithread VSM document retriever can be seen in Fig. 4. Since each retriever is responsible for an index, the number of jobs is equivalent to the number of indexes. Each retriever retrieves its relevant documents based on query input and merges it to global result.

J. Multithread LRVSM

Multithread LRVSM works quite similar with multithread VSM except LRVSM involves pre-calculated semantic relatedness. Since semantic relatedness between terms is stored in binary file, index reader should load that file and document retriever should involve it

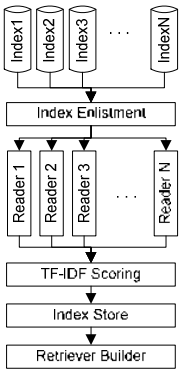

Fig. 3. Multithread VSM Index Reader.

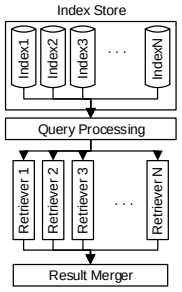

Fig. 4. Multithread VSM Document Retriever.

on its retrieval mechanism. Multithread LRVSM index reader can be seen in Fig. 5. This module enlists all indexes and relatedness file in order to load it on memory. Following VSM index reader design, each file is considered as one job and all index terms are weighted using tf-idf scoring before stored in memory. Multithread LRVSM document retriever can be seen in Fig. 6. Relatedness data are stored in array and shared among all retrievers. This mechanism is thread-safe since only read action permitted on shared data.

K. Multithread and Multiprocess Semantic Relatedness Calculator

Since semantic relatedness between terms is precomputed and takes a long time, this module is also redesigned by involving multithread and multiprocess

which can be seen in Fig. 7. This module takes all indexes as its input, generates distinct terms, and calculates semantic relatedness on separate processes. Separate processes is implemented by executing many standalone executable files. Processes are chosen instead of threads because:

- Many third-party semantic relatedness libraries involve synchronized and static methods which may yield bottlenecks if implemented only in separate threads (e.g., Ws4J: WordNet Similarity for Java [18]). Multithread design with bottlenecks may yield longer processing time than the naive one (single-thread) since multithreading needs additional time to split and merge jobs.
- Standalone executable files may be built in many programming language other than Java as long as it follows its input and output template.

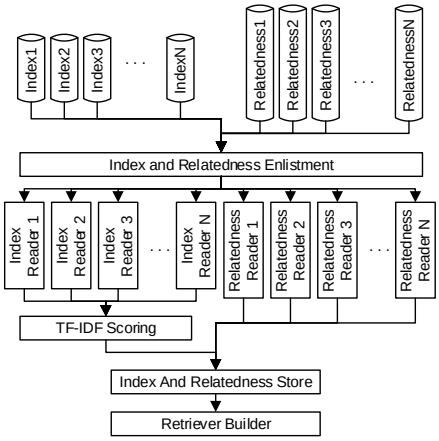

Fig. 5. Multithread LRVSM Index Reader.

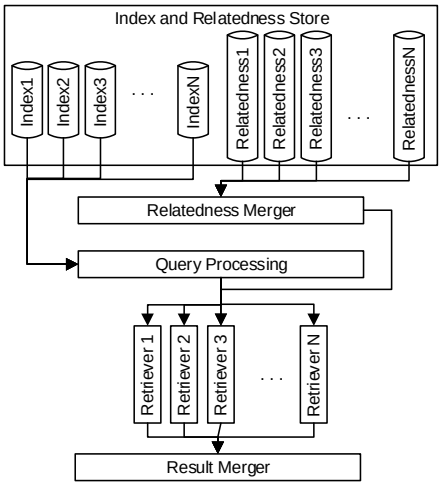

Fig. 6. Multithread LRVSM Document Retriever.

- Semantic relatedness calculator may be freely designed by its developer. Internal saving mechanism may also be added as additional feature since calculating semantic relatedness takes a long time (calculating semantic relatedness of 40.978 distinct terms in previous research using single thread takes about five days).

Standalone executable file built for this module must follow input and output template. It takes six input arguments which are source file (CSV file which consists all distinct terms), lower and upper bound of first and second job, and target file. The program should take all distinct term from source file, calculate semantic relatedness between terms on given job, and store it to target file in binary format of hash map. Target hash map uses double as its value and string as its key. The key represents concatenated term pair separated by vertical bar ("|") whereas its value represents term pair relatedness.

Since distinct terms are stored in array and each term i is only paired with remaining terms with larger index than i, semantic relatedness calculation at the end of array should be faster than the beginning part. To distribute term calculation tasks evenly, each process (executable file) is given two jobs. First one is from the beginning of an array and the second one is from the end of an array. Distinct terms are split to $2 \times N$ jobs and each process i is assigned to job i and $N - i$

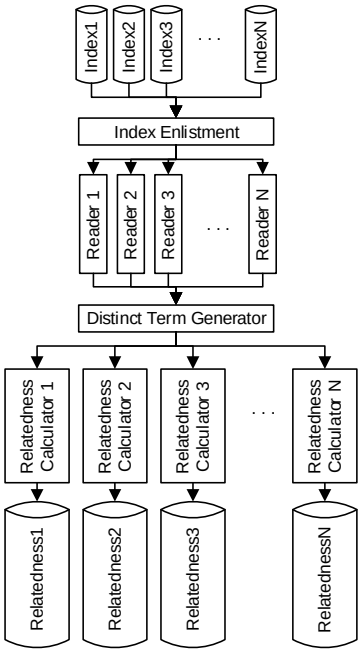

Fig. 7. Multithread Semantic Relatedness Calculator.

where N represent the number of processes.

III. RESULTS AND DISCUSSION

Efficiency and effectiveness are two major measurements which are commonly used to determine the feasibility of a research. Since this research focuses on module conversion in order to improve its scalability, efficiency is measured based on processing time and scalable design whereas effectiveness is measured on converted modules correctness. Evaluation conducted in this research uses default dataset from Ref. [1] as a benchmark. For clarity in each table, blue mark represents the best result whereas red mark represents the worst result for each factor.

A. Evaluating Recursion Conversion

To evaluate recursion conversion, some schemes are augmented which are shown in Table II. Each type consists of three symbols which represent module implementation (I = Iteration and R = Recursion). Its symbol order is equivalent to module order shown in Table II columns. RRR and III schemes are the benchmarks of this evaluation since RRR represent recursive approach in all modules and III represent iterative approach in all modules. Type IRR, RIR, and RRI are used to measure conversion impact of certain module.

Indexing time of each schemes on default dataset can be seen in Table III. Since processing time is operating system dependent, each scheme is measured five times using the same dataset and its average result is assigned as its result to reduce its dependency bias.

As seen in Table III, each scheme that involves iterative implementation takes more processing time rather than RRR scheme. Following recursive logic in iterative approach may yield longer processing time since it requires many additional objects on its process (e.g. IterTuple in Iterative Tarjans SCC algortihm). Although its processing time is less efficient, it is more scalable than recursive approach. Stack overflow error rarely appears on iterative approach since the number of function call is greatly reduced. RRR scheme on default dataset generates a stack overflow error when run on JRE 1.8 whereas it runs well on JRE 1.7. This is caused by many update in JRE 1.8 which detects some processes in this program as endless recursion (although it is not). Iterative form is also a better approach than recursive form since the number of recursive calls conducted on dataset is uncertain. III scheme takes the longest indexing time since all modules are implemented iteratively.

Recursion conversion correctness is measured by comparing the result of iterative and recursive approaches. Its correctness is proved when both implementation yield similar results on various dataset. In this evaluation, 11 dataset are tested where the first dataset is default dataset in Ref. [1] and the rest are sub dataset split from default dataset. Sub dataset are resulted by splitting default dataset to 10 parts evenly. Since these modules are indexer part, result comparison is conducted based on indexes generated by both implementation. As a result, both implementation yield similar results on all schemes.

TABLE II
RECURSION CONVERSION EVALUATION SCHEMES.

Type	Module		
	Loop Encapsulation	Recursive Method Elimination	Method Expander
RRR	Rec	Rec	Rec
IRR	Iter	Rec	Rec
RIR	Rec	Iter	Rec
RRI	Rec	Rec	Iter
III	Iter	Iter	Iter

B. Evaluating Multithread Design

Multithread indexer, VSM, EVSM, and semantic relatedness calculator are measured in term of efficiency and effectiveness. These evaluation are conducted in Windows 7 Ultimate 32-bit with 4 GB RAM, and Intel(IR) Core™ i7-3770 CPU @ 3.40 GHz and 3.90 GHz as its processor. Each evaluated module excepts semantic relatedness calculator is tested by various number of jobs which are $1, N, N \times 2, N \times 5, N \times 10, N \times 20, N \times 40$, and 552. N is the number of physical cores -1, which is 3 in this evaluation environment whereas 552 is the number of Java archives indexed in our dataset. To reduce its dependency bias, each evaluation scheme is measured five times using the same dataset and its average result is assigned as its result.

TABLE III
INDEXING TIME OF RECURSION CONVERSION SCHEMES.

Type	Indexing Time (s)
RRR	417.297
IRR	423.964
RIR	436.958
RRI	429.307
III	433.995

C. Evaluating Multithread Indexer

The quantitative efficiency measurement of multithread indexer can be seen in Table IV. The best indexing time is gained at $N = 3$ since all physical cores are utilized (4 cores, 1 core is used for main thread and the rest are used for side threads). $N = 1$ takes the longest indexing time since it only utilizes 1 side thread (which is quite similar to sequential process). Index size is increased proportionally with the number of jobs since each job generates an index file and each index has its own header file.

Since the contents of generated indexes for each scheme are equivalent with the contents of index generated in previous research, multithread design on this module is proved correct. Time reduction in multithread indexer also proves that multithreading may improve scalability since it enables this system to process larger dataset.

D. Evaluating Multithread VSM and LRVSM

Multithread VSM and LRVSM are measured with similar factors since both of them are retriever modules. Measured retriever's efficiency factors are index load time and average query latency whereas effectiveness is only measured based on its correctness.

The time efficiency measurement of multithread VSM and LRVSM can be seen in Tables V and VI. Since LRVSM evaluation is assumed to take relatedness file as a single file, both results yield similar conclusions which are:

- Index load time runs fastest at $N = 3$ since all physical cores are utilized. The number of jobs is proportional to index load time since job transfer in thread takes time. $N = 1$ takes longer time than N=3 since it only utilizes one thread instead of three.
- Average query latency is affected by the number of index partitioning. Retrieving a term from one big chunk of index takes longer time than retrieving it from many small chunks since large index may yield more complicated structure than the small one. Small indexes are also easier to be cached in memory. But too many small indexes may yield longer processing time since it needs to iterate through these indexes. As seen in Table VI, average query latency at $N = 552$ takes longer time than $N = 120$ although it has more indexes with smaller size.

Conversion correctness are proven by the fact that both retriever yields similar result with its respective sequential form for all queries from default dataset (1860 queries). Multithread VSM and LRVSM is also more scalable since shorter processing time may enable system to handle larger data.

TABLE IV
TIME AND MEMORY EFFICIENCY OF MULTITHREAD INDEXING.

N	Efficiency Measurement	
	Indexing Time (s)	Index Size (MB)
1	326.851	4,894
3	153.913	5,213
6	164.822	5,559
15	169.354	5,860
30	172.387	6,525
60	174.642	6,973
120	180.711	7,623
552	240.053	9,142

TABLE V
TIME EFFICIENCY OF MULTITHREAD VSM.

N	Efficiency Measurement	
	Indexing Time (s)	Ave. Query Latency (s)
1	1.749	0.462
3	0.831	0.099
6	0.926	0.045
15	0.961	0.024
30	1.118	0.011
60	1.310	0.007
120	1.572	0.007
552	2.558	0.024

TABLE VI
TIME EFFICIENCY OF MULTITHREAD LRVSM.

N	Efficiency Measurement	
	Indexing Time (s)	Ave. Query Latency (s)
1	56.041	0.091
3	55.563	0.047
6	55.759	0.046
15	56.603	0.043
30	57.128	0.042
60	57.287	0.041
120	57.884	0.039
552	59.208	0.152

TABLE VII
SEMANTIC RELATEDNESS EVALUATION DATASET.

Java Archive	Distinct Terms	Related Term Pairs
javassist-2.5.1.jar	646	22.770
lucene-1.2.jar	553	15.434
jxl-2.4.2.jar	1.102	63.415

TABLE VIII
PROCESSING TIME OF SEMANTIC RELATEDNESS CALCULATOR (S).

Java Archive	Single Thread Design	Multithread Design
javassist-2.5.1.jar	2.237,249	1.355,221
lucene-1.2.jar	1.457,311	828,732
jxl-2.4.2.jar	5.249,131	3.372,989

E. Evaluating Multithread Semantic Relatedness Calculator

Since semantic relatedness calculator requires standalone executable files for completing its task, a Java-based standalone executable file (JAR) is build for this research. This program follows algorithm template given by multithread semantic relatedness calculator. Since calculating semantic relatedness is time consuming, multithread design of this module is only tested to three Java archives instead of entire default dataset. These Java archives characteristics can be seen in Table VII.

Processing time of multithread and single-thread design of semantic relatedness calculator can be seen in Table VIII. Each scheme is evaluated five times and its average result is considered as that scheme result. Evaluation is conducted by split distinct terms to three jobs and utilize Lin's semantic relatedness algorithm described in Ref. [19]. As seen in Table VIII, multithread design is more scalable since it takes less time than single-thread by utilizing all physical cores. This design also yields the same result as single-thread design which proves its correctness.

IV. CONCLUSIONS

Based on evaluation in this research, scalability of Java archive search engine can be improved through recursion conversion and multithreading. Recursion conversion improves scalability by avoiding stack overflow error whereas multithreading improves scalability by reducing its execution time (which enables system to process larger dataset).

Recursion conversion conducted in this research involves three main recursive modules which are loop encapsulation, recursive method elimination, and method expansion. Although these modules are inconvenient enough to be redesigned as iterative one, it still can be converted by following its recursive pattern with the help of caller reference. Recursive pattern in iterative implementation takes more time than its recursive form since these modules are logically recursive and require many additional objects during its execution. The correctness of recursion conversion described is also proved by the fact that both implementation yield similar result.

Multithread design has been successfully implemented in Java archive search engine which involves indexer, VSM retriever, LRVSM retriever, and semantic relatedness calculator. This mechanism cuts off its respective processing time since it utilizes all physical cores. Index partitioning may yield faster retrieval model although too many small indexes may also yield longer processing time. All multithread modules are also proved correct by black box testing its results.

REFERENCES

[1] O. Karnalim and R. Mandala, "Java archives search engine using byte code as information source," in *Data and Software Engineering (ICODSE), 2014 International Conference on*. IEEE, 2014, pp. 1–6.

[2] O. Karnalim, "Extended vector space model with semantic relatedness on java archive search engine," *Jurnal Teknik Informatika dan Sistem Informasi*, vol. 1, no. 2, 2015.

[3] W. B. Croft, D. Metzler, and T. Strohman, *Search engines: Information retrieval in practice*. Addison-Wesley Reading, 2010, vol. 283.

[4] D. Grune, K. Van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern compiler design*. Springer Science & Business Media, 2012.

[5] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[6] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[7] M. Carrano and T. Henry, *Data Abstraction & Problem Solving with C++: Walls and Mirrors*, 6th ed. Prentice Hall, 2012.

[8] A. S. Tanenbaum, *Modern Operating Systems*, 4th ed. Prentice Hall, 2014.

[9] W. Liu and T. Wang, "Index-based online text classification for sms spam filtering," *Journal of Computers*, vol. 5, no. 6, pp. 844–851, 2010.

[10] W. Premchaiswadi and A. Tungkatsathan, "Online content-based image retrieval system using joint querying and relevance feedback scheme," *WSEAS Transactions on Computers*, vol. 9, no. 5, pp. 465–474, 2010.

[11] C. Bonacic, C. Garcia, M. Marin, M. Prieto, F. Tirado, and C. Vicente, "Improving search engines performance on multithreading processors," in *High Performance Computing for Computational Science-VECPAR 2008*. Springer, 2008, pp. 201–213.

[12] C. Bonacic and M. Marin, "Simulation study of multi-threading in web search engine processors," in *String Processing and Information Retrieval*. Springer, 2013, pp. 37–48.

[13] V. Skylarov, I. Skilarova, and B. Pimentel, "Fpga-based implementation and comparison of recursive and iterative algorithms," in *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, pp. 235–240.

[14] J. Miecznikowski and L. Hendren, "Decompiling java using staged encapsulation," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 368–374.

[15] M. Naftalin and P. Wadler, *Java generics and collections*. " O'Reilly Media, Inc.", 2006.

[16] C. Kustanto and I. Liem, "Automatic source code plagiarism detection," in *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. 10th ACIS International Conference on*. IEEE, 2009, pp. 481–486.

[17] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1, no. 1.

[18] H. Shima. (2015) Ws4j : Wordnet similarity for java. Accessed on November 24, 2015. [Online]. Available: https://code.google.com/p/ws4j/

[19] D. Lin, "An information-theoretic definition of similarity." in *ICML*, vol. 98, 1998, pp. 296–304.