Can We Make It Faster? Efficient May-Happen-in-Parallel Analysis Revisited

Congming Chen1 2  Wei Huo1  Lung Li1  Xiaobing Feng1
1 State Key Laboratory of Computer Architecture, CAS  
2 Graduate University of Chinese Academy of Sciences  
Beijing, China  
{chencongming, huowei, fxb}@ict.ac.cn

Kai Xing
School of Computer Science  
University of Science and Technology of China  
Anhui, China  
kxing@ustc.edu.cn

Abstract—May-Happen-in-Parallel (MHP) analysis is a very important and fundamental mechanism to facilitate concurrent program analysis, optimization and even concurrency bug detection. However, the inefficiency in its design and implementation keeps MHP analysis away from being practical and effective. In this paper, we investigate the state-of-art of iterative data flow based (IDFB) MHP analysis and propose a new design and corresponding systematic implementation. Specifically, we address the most severe efficiency problems in node process order of the work-list in the original approach, and resolve them in our design and implementation by using the concept of parallel level to avoid redundant node visits. Our intensive experimental study shows that the proposed design and implementation have a relative speed up of 29.02x compared with the original implementation, moreover, it achieves a relative speed up of 10.00x comparing to the state-of-art of non-IDFB approach which is claimed to be more efficient than the original IDFB approach. Our design and implementation are capable of achieving an order of magnitude efficiency improvement comparing to both IDFB and non-IDFB approaches.

Keywords- Concurrent Program, Control Flow Analysis, Conservative, Efficiency, Precision

I. INTRODUCTION AND MOTIVATION

May-Happen-in-Parallel (MHP) analysis is an important and fundamental mechanism to facilitate concurrent program analysis. It is used for determining whether a given pair of statements in a concurrent program can be executed in parallel or not. MHP analysis serves as a cornerstone of other static and dynamic analysis techniques, such as concurrent control flow and data flow analysis, concurrent program optimization, concurrent program debugging and synchronization anomalies detection including data race and dead lock checking [1][2].

It is difficult to precisely determine whether all pairs of statements in a concurrent program may happen in parallel or not[3]. Taylor[4] has proved that this problem is NP-complete given that all control paths in all threads are executable. Thus, MHP algorithms are computing an approximate result. There are many outstanding approaches which have been proposed in previous works, and the majority of them are iterative data flow based (IDFB)[5][7][6][7][9][10][11]. There are also some non-IDFB MHP algorithms which either use hierarchical data structures[12] or take advantage of highly abstracted concurrent syntax[13][14][15] to facilitate the analysis, making MHP analysis more efficient, but difficult to process complex concurrent operations to a certain extent. For example, very few non-IDFB approaches can handle inter-threaded synchronization operations.

Since MHP analysis is essentially a control flow problem, there is a natural demand for iterative data flow based MHP algorithm. Meanwhile, compared with non-IDFB approaches, IDFB approaches are more practical, precise, and extensible. For example, different kinds of concurrent operations including inter-threaded synchronization operations can be easily integrated into the data flow equations of IDFB approaches. The IDFB MHP approaches are playing very important role in MHP analysis.

However, existing IDFB approaches are not efficient enough to analyze large scale real world concurrent programs. Generally, the problem is caused by the complexity of these approaches, e.g. the state-of-art IDFB MHP algorithm is proposed by Naumovich[9] for concurrent Java program model, with a complexity of $O(N^3)$, where $N$ is the number of statements in a program. However, the implementation strategy of some pivotal parts of the algorithm may also be an inducement to the efficiency problems as we will address in section III. The scale of existing analyzed programs in previous works is far smaller than the scale of real world concurrent programs. Therefore, there is an imperative demand for solving the efficiency problems in analyzing large scale programs.

In this paper, we present a new design and implementation of the state-of-art IDFB MHP algorithm. In our design, we address and solve the most severe efficiency problems in node process order of the work-list in the original approach. Our implementation achieves an order of magnitude efficiency improvement compared with both the original approach and even the state-of-art non-IDFB approach which declares to be more efficient than the state-of-art IDFB approach. The contributions of our work can be concluded as follows:

1) We address the two most severe efficiency problems in the original state-of-art IDFB MHP approach in this paper: non-topological node processing order which leads to repeated computation, and eagerly update of node information caused by symmetry of MHP analysis leads to redundant computation. Neither of these problems was discovered and discussed before.

2) We illustrate the efficiency improvement by significant experiment, which shows that our implementation has a relative speed-up of 29.02x comparing to the original approach and a relative speed up of 10.00x comparing to the state-of-art non-IDFB approach, which is an order of magnitude improvement on efficiency.
The rest of this paper is organized as follows. We’ll review some basic concepts of the state-of-art IDFB MHP approach in section 2, and then state its efficiency problems and illustrate our implementation in section 3. Section 4 presents our experimental data and section 5 draws a conclusion of our work.

II. BASIC CONCEPTS

The state-of-art IDFB MHP analysis approach is proposed by Naumovich[9], with a graph-based program representation using the so-called Parallel Execution Graph (PEG). The following are some basic concepts of this approach.

A. Program Representation

This approach is based on Java model of concurrency. Several so-called thread communication methods are processed in their approach and relative data flow equations are designed according to their concurrent syntax. For example, method start() is used to create a new thread, and method join() is used to join an existed thread. Moreover, inter-threaded synchronization methods such as notify() and wait() are also processed in this approach.

The graph-based program representation PEG is constructed on the inter-threaded concurrent control flow graph: each thread has its intra-threaded control flow graph and different intra-threaded control flow graphs are connected by different kinds of edges, which are defined according to their concurrent syntax, including: thread spawn edge, inter-threaded communication edges. Each node in PEG is a triple <object, name, caller>, where name represents a method, and object is an object who owns the method name, caller is the calling thread.

B. Data Flow Equations

Different data flow equations are defined for thread communication methods according to their concurrent syntax. For each PEG node n, it has a set M(n) to preserve all nodes which may happen in parallel with n, a set OUT(n) to preserve nodes may happen in parallel with successors of n. The essential data flow equation is defined as:

\[ OUT(n) = (M(n) \cup GEN(n)) \setminus KILL(n) \] (1)

The GEN set and KILL set are defined for PEG nodes contain relative concurrent syntax. For example, a PEG node with start() method has a GEN set since a new thread is created after this method is executed, and new happen in parallel relations are thus created. This GEN set is defined to contain the first PEG node of the created thread. Another example is that, given a PEG node with join() method which will terminate an existing thread specified by the field object of this node after this method is executed, a KILL set of this node is defined to include all PEG nodes of the terminated thread. In addition, the KILL set and GEN set will not change during the iterative analysis process, so they can be pre-computed.

Based on the above essential data flow equation, several data flow equations are defined for all kinds of thread communication methods, including inter-threaded synchronization methods, such as notify() and notifyAll(). Finally, the iterative process is based on these pre-defined data flow equations.

C. Iterative Process

The IDFB algorithm is based on a work-list implementation. Existing nodes in the work-list will be processed one at a time and new nodes may be added into this list according to certain rules. The whole iterative process reaches a fixed point until the M set and OUT set of each node won’t be changed any more, and, at the same time, the work-list becomes empty.

The M set and OUT set of each node will be changed on the fly during the iterative process, and to check possible update of these sets, two original node sets are preserved before each single iteration, represented as M_{old} and OUT_{old} respectively. New nodes are added into the work-list primarily according to the following two rules:

a) \( \forall n \in PEG, \text{ if } M(n) \neq M_{old}(n), \text{ then } \forall m \in \{M(n) - M_{old}(n)\} \)

b) \( \forall n \in PEG, \text{ if } OUT(n) \neq OUT_{old}(n), \text{ then } \forall m \in \{OUT(n) - OUT_{old}(n)\} \)

Before starting on discussing our implementation, we will illustrate the efficiency problems in the original approach, which will be solved with our implementation.

A. Efficiency Problems

Nodes in the work-list are processed in a First-In-First-Out (FIFO) way in the original algorithm[10]. It seems that it is a natural order to process these nodes. However, we will see in the following motivation example that the process order of nodes in the work-list has a significant influence on algorithm efficiency. A simple PEG which involves two threads is shown in figure 1. Node n2 contains start() method to create a new thread t2, and node n3 contains join() method to terminate thread t2. We adopt three orders to process nodes in the work-list. TABLE 1 shows how these orders affect the iterative process. These three orders includes: FIFO, which stands for the original First-In-First-Out order. STOP1 and STOP2 respectively stand for two different im-
implementations of a kind of topological order of nodes in the work-list. The difference between these two implementations will be elaborated later. The second column in the table shows the iterative process details, and the third column shows the number of iterations in the whole process.

![Diagram of PEG]

**TABLE I. EFFECT OF DIFFERENT PROCESS ORDERS**

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Iteration Details</th>
<th>Iter#</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>n2, n3, n6, n4, n7, n6, n5, n3, n8, n7, n6, n4, n3, n8, n7, n5</td>
<td>19</td>
</tr>
<tr>
<td>STOP1</td>
<td>n2, n3, n6, n4, n5, n7, n3, n4, n5, n8, n3, n4, n5</td>
<td>13</td>
</tr>
<tr>
<td>STOP2</td>
<td>n2, n3, n4, n5, n6, n7, n8, n3, n4, n5</td>
<td>10</td>
</tr>
</tbody>
</table>

Naumovich has proved that the result of the algorithm “does not depend on the order in which nodes are placed on and removed from the work-list”[10]. Thus, the three orders listed above do produce the same results, but have different iteration processes.

In the above three orders, FIFO has the worst efficiency, as we have noticed that many nodes are repeatedly processed during the iterative process. This is mainly because this strategy has two potential problems which will lead to inefficiency:

i) **Non-topological Order.** Nodes are added into the work-list from different threads alternatively in FIFO order according to the original algorithm. This might cause a node to be processed before its predecessor nodes. As a result, the current node needs to be processed again if the OUT sets of its predecessor nodes are updated. For example, in table I, among the first 6 nodes in the iteration details of FIFO order, n3 and n6 are added into the work-list according to rule b) in section II since they are the immediate successors of n2. Then again, according to rule b), n4 and n7 are added into the work-list respectively since OUT(n3) and OUT(n6) are updated. After that, M(n4) will be updated during the iterative process, and n6 is added into the work-list again because of the symmetry requirement of MHP analysis. As a result, n7 is processed before n6, and each time M(n6) has been updated, all of its successor nodes including n7 and n8 need to be processed again. However, we can avoid this kind of repeated computation by defining a topological order which can enforce that a node will never be processed before its predecessors.

ii) **Eager Update.** According to the symmetry requirement of MHP analysis in the original algorithm, each time a node n is added into the M set of another node m, m will be added into M(n) as well, and n will be added into the work-list. This will also lead to inefficiency if we process nodes in the work-list in FIFO order, because the processing frequency of node n may depend on the number of nodes which may happen in parallel with n. Again, take the first 6 nodes in the iteration details of FIFO in table I as an example, the first time n6 is added into the work-list is because n6 is propagated into M(n3) from OUT(n2). Then, the second time n6 is added into the work-list is because n6 is propagated into M(n4) from OUT(n3). Each time n6 is propagated into the M set of another node, n6 is added into the work-list and processed again. However, we do not need to eagerly update n6 in this way; for instance, we can delay processing n6 in the work-list until n6 has been propagated to all of the successors of n3. It is worth noting that this problem cannot be avoided by just defining a topological order, in fact, same problem exists in STOP1.

B. Design and Implementation

Our solution to tackle the first efficiency problem is to define a topological order between nodes in PEG and process nodes in the work-list according to this topological order. As we all know, topological order of a directed acyclic graph is not unique. According to the definition of topological order and the so-called source removal algorithm [16], if there is more than one node that has no incoming edges, then the topological order depends on which node may be chosen as the next node. We define a special kind of topological order in our implementation, and it depends on the following principle definition and algorithm.

**Definition 1: Parallel Level (PL)**

∀t∈PEG, t is a thread. We assign a number to t as the parallel level of t, which is represented as PL(t). PL is defined as follows:

1) Given two threads t₁, t₂∈PEG, if thread t₂ is created by thread t₁, which is represented as: t₁ spawn→t₂, then PL(t₂) = PL(t₁) + 1.

2) Suppose main stands for the main thread of the program, then we define PL(main)=0. Hence, considering 1), we have ∀t∈PEG, PL(t)≥PL(main).

Based on definition 1, we define the topological order generated by following algorithm 1 as the strict topological order. Before performing algorithm 1, we transfer a PEG into a directed acyclic graph (DAG) by traversing the whole PEG to find out all the strongly connected components (SCCs) and merge each SCC into a single node. The static thread model adopted in approach [9] guarantees that the SCC merging operation has no influence on the final MHP analysis results.
Algorithm 1:
S : An empty stack used for keeping temporary nodes
L : Empty list which will contain sorted elements
R : Root node of PEG
push R → S;
while(S is not empty)/
    n = pop(S);
delete n from PEG;
    add n → L;
    if a new thread is created by n
        find out immediate successor node m’ of n which satisfies:
            PL(m’.caller)>PL(n.caller)
            delete <n, m’> from PEG
            push m’ → S;
    end if
for each edge e=<n, m> in PEG
    delete e from PEG;
    if indegree(m) is 0
        push m → S;
    end if
end for
return L;

In fact, strict topological order is still not unique. As the step marked with (*) in algorithm 1 indicates, each selection strategy of the next node will lead to a different strict topological order if node n has multi-successors. However, this kind of difference has no impact on the iterative process and the final MHP results because it is intra-threaded.

Our solution to tackle the second efficiency problem lies in the implementation of STOP2. Both STOP1 and STOP2 are based on strict topological order, nodes are added into the work-list according to this order, but they are different in the style of removing nodes from the work-list. In STOP1 order, each time we remove a node from the work-list, we always try to pick the first node in current work-list, we call this way as first-priority style. While in the implementation of STOP2 order, we try to get the nearest next node of newly removal node from the work-list, and we call this way as next-priority style. As a result, in STOP1 order, we preferentially process nodes in threads with lower parallel level according to the definition of strict topological order in algorithm 1. Thus, each time these nodes are propagated into the M set of node with higher parallel level, they are added into the work-list and processed eagerly. As the process detail of STOP1 in table 1 indicates, nodes including n3, n4, n5 are processed three times since there are three nodes in thread t2. However, in STOP2 order, although these three nodes are added into the work-list due to the symmetry requirement of MHP analysis, they are only processed until all nodes in thread t2 are processed. The key point here is that we delay the update of nodes with lower parallel level in the implementation of STOP2.

Finally, the efficiency problems caused by eager update and non-topological order can be elegantly solved by the implementation of STOP2, making it as our final solution to improve efficiency of the original approach.

IV. EXPERIMENT

We implemented our MHP algorithm based on the PThreads concurrent programming model[17] which is one of the most popular concurrent programming models. This programming model has been adopted in many large scale real world applications, especially in web server development, and thus makes our implementation more general and practical. The porting process from Java to PThreads is trivial since these two programming models have similar concurrent syntax, each concurrent operation in Java has a correlative operation in PThreads, so we do not discuss details of building PEG based on PThreads program here.

Our implementation is integrated in the open source compiler Open64[18]. We evaluated our implementation on a hardware platform with 1.86GHz CPU and 16GB memory, using CentOS-5 as the operation system. We compared the above three orders and the state-of-art non-IDFB approach which is proposed in [12]. Since the primary program representation defined in this approach is Thread Creation Tree (TCT), we represent this approach as TCT in our experiment. Benchmarks used in our implementation are mainly come from SPLASH2[19]. Experimental data is shown in table II and figure 2. In table II, we list benchmark lines, the number of threads modeled in our experiment, the overall number of nodes and number of edges in PEG respectively from column 2 to column 5. We show the pure MHP analysis time of different orders, including FIFO, STOP1 and STOP2 in the so-called “IDFB MHP TIME” category in table II. We do not include the time of building PEG or TCT, since the building time is much shorter comparing with the MHP analysis time, we just simply ignore this. In the category called “NODE VISITING NUM MAX#/AVERAGE#”, we list the maximum and average number of visits to a node during the iterative data flow analysis process for different orders. For a given test case, a larger number of visits to a node generally mean that this node needs more time in iterative process. From the data listed in table II, we can find out that data in the previous two categories are consistent with each other. We list the analysis time of TCT approach[12] in the last column, which is one of the state-of-art non-IDFB approaches, and it is declared to be more efficient than the original PEG approach[9].

We set a time limitation of 43200 seconds (12 hours) in our experiment, since large cases such as “radiosity” take a long time to be analyzed when FIFO or STOP1 order are adopted. Cases run out of this time limitation are marked as “OOT” in category “TIME”, which means “Out Of Time”, meanwhile, number of visits to a node in these cases is also marked as “OOT” in related category.

We also show the relative speed up between different orders by normalizing the analysis time of our solution STOP2 in figure 2, data are represented in logarithmic form, the lower the better. The case “radiosity” is not included in this figure, since it runs out of time for orders of FIFO and STOP1. As the speed up indicates, for all test cases, STOP2 is definitely more efficient than FIFO which is the order used in the original algorithm, and for all cases, STOP2 yields an geometric average relative speed up of 29.02×.
<table>
<thead>
<tr>
<th>TEST CASES</th>
<th>KLOC</th>
<th>T#</th>
<th>N#</th>
<th>E#</th>
<th>IDFB MHP TIME(s)</th>
<th>NODE VISITING NUM MAX#/AVERAGE#</th>
<th>TCT TIME(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIFO STOP1 STOP2</td>
<td>FIFO STOP1 STOP2</td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td>1.47</td>
<td>4</td>
<td>1602</td>
<td>1946</td>
<td>11.71 22.89 0.75</td>
<td>229/146 1069/481 2/1</td>
<td>4.35</td>
</tr>
<tr>
<td>radix</td>
<td>1.43</td>
<td>6</td>
<td>1885</td>
<td>2295</td>
<td>10.8 42.34 1.17</td>
<td>178/95 1431/660 2/1</td>
<td>5.07</td>
</tr>
<tr>
<td>lu-con</td>
<td>1.39</td>
<td>5</td>
<td>1250</td>
<td>1529</td>
<td>3.64 10.88 0.27</td>
<td>129/70 817/340 2/1</td>
<td>2.29</td>
</tr>
<tr>
<td>lu-non</td>
<td>1.16</td>
<td>5</td>
<td>1186</td>
<td>1451</td>
<td>3.61 10.71 0.27</td>
<td>129/74 817/359 2/1</td>
<td>1.98</td>
</tr>
<tr>
<td>cholesky</td>
<td>5.69</td>
<td>2</td>
<td>23740</td>
<td>28711</td>
<td>5062 1743.98 54.36</td>
<td>466/310 9819/4063 2/1</td>
<td>692.32</td>
</tr>
<tr>
<td>barnes</td>
<td>3.51</td>
<td>4</td>
<td>1396</td>
<td>1687</td>
<td>7.29 18.11 0.42</td>
<td>208/120 1081/501 2/1</td>
<td>2.67</td>
</tr>
<tr>
<td>fnm</td>
<td>5.43</td>
<td>2</td>
<td>9102</td>
<td>10526</td>
<td>53.7 1471.67 10.9</td>
<td>428/238 4529/2256 2/1</td>
<td>116.32</td>
</tr>
<tr>
<td>ocean-con</td>
<td>8.18</td>
<td>2</td>
<td>5255</td>
<td>6437</td>
<td>312.7 319.2 3.53</td>
<td>947/551 2529/1219 2/1</td>
<td>35.29</td>
</tr>
<tr>
<td>ocean-non</td>
<td>6.25</td>
<td>2</td>
<td>5066</td>
<td>6222</td>
<td>274.46 279.42 3.23</td>
<td>943/532 2400/1139 2/1</td>
<td>33</td>
</tr>
<tr>
<td>raytrace</td>
<td>11.04</td>
<td>2</td>
<td>5714</td>
<td>7081</td>
<td>10.51 32.83 0.5</td>
<td>113/24 769/104 2/1</td>
<td>51.45</td>
</tr>
<tr>
<td>volrend</td>
<td>5.49</td>
<td>2</td>
<td>1130</td>
<td>1347</td>
<td>2.81 3.41 0.12</td>
<td>205/80 408/152 2/1</td>
<td>0.93</td>
</tr>
<tr>
<td>water-nsq</td>
<td>3.1</td>
<td>2</td>
<td>1560</td>
<td>1866</td>
<td>11.61 11.88 0.32</td>
<td>304/188 733/347 2/1</td>
<td>2.99</td>
</tr>
<tr>
<td>water-spa</td>
<td>3.66</td>
<td>2</td>
<td>1726</td>
<td>2092</td>
<td>14.19 14.83 0.38</td>
<td>309/194 798/371 2/1</td>
<td>3.72</td>
</tr>
<tr>
<td>radiosity</td>
<td>24.52</td>
<td>2</td>
<td>160712</td>
<td>183679</td>
<td>OOT   OOT 4693.05</td>
<td>OOT OOT 28124.97 2/1</td>
<td></td>
</tr>
</tbody>
</table>

An interesting phenomenon we can find in figure 2 is that FIFO is generally more efficient than STOP1. Specifically, FIFO has an average relative speed up of 1.82× compared with STOP1. This is a little bit different from what we see in the sample program, however, this is not a surprising result. Even though we can use STOP1 to avoid the first efficiency problem caused by non-topological order, the eager update problem still exists in this implementation, and it is more serious than that in the FIFO implementation. As a result, the STOP1 order leads to even more redundant computation in our experiment, and performs much worse than FIFO order.

A notable improvement of our implementation compared with the original algorithm is that we significantly improved scalability of IDFB MHP approach. For the largest test case in table II, only TCT approach and our implementation can get the final results within the time limitation. Moreover, STOP2 is even more efficient than TCT.

### V. RELATED WORK

Many approaches have been proposed to compute MHP information with different concurrent programming models. Callahan and Subhlok[5] proposed a prototype tool called PTOOL for analyzing asynchronous parallel loops, their analysis is called B4 analyzing. Duesterwald and Soffa[6] extended B4 analysis to be inter-procedural, and applied this analysis to Ada programs with rendezvous synchronization model. Masticola and Ryder[7] proposed an approach called non-concurrency analysis to compute a conservative statement set which cannot happen in parallel with a specified statement based on an iterative approach for Ada programs. The worst complexity of their algorithm is $O(S^5)$, where $S$ is the number of statements in a program. After that, Naumovich and Avrunin[8] proposed a conservative data flow algorithm to compute MHP results for all statements. Their approach is also based on the rendezvous model of concurrent Ada programs, for a set of concurrent Ada programs,
their algorithm can get a higher precision than previous approaches with a complexity of $O(k^2)$.  

Naumovich[9] then proposed a MHP algorithm for Java programs, which is the state-of-art IDFB MHP approach discussed in this paper. Li[11] then proposed several optimization techniques to reduce the analysis cost of Naumovich’s algorithm[9], such as simplifying procedures without concurrent operations into single nodes. This optimization is also applied in our implementation.

Barik[12] proposed an efficient algorithm to compute MHP information for Java programs based on Thread Creation Tree (TCT). Complexity of their algorithm is $\Theta((kN)^2)$ in which k is the number of abstract threads according to their novel static thread model, and N is the number of statements per abstract thread. It gets an average speed up of 1.77× in analysis time comparing with Naumovich’s approach[9]. It is the state-of-art non-IDFB approach which we compared in our experiment. The tree structure is also adopted in Agarwal’s algorithm[13] which focuses on computing MHP information for X10 programs. Kamil[14] proposed a concurrency analysis algorithm for parallel programs written with Titanium language, where Titanium is a single program, multiple data global address space variation of Java. The analysis takes advantage of two key features of Titanium language including textually aligned barriers and single-valued expressions. Lee[15] proposed a type system to compute MHP information for featherweight X10 language with async-finish parallel primitives, and they proved the correctness of their type system.

VI. CONCLUSION

We present a new design and systematic implementation of the state-of-art IDFB MHP approach in this paper. Two most severe efficiency problems about node process order in the work-list in the original approach are resolved in our implementation. Our implementation has a relative speed up of 29.02× compared with the original approach, moreover, it also achieves a reasonable speed up of 10.00× compared with the state-of-art non-IDFB MHP approach. Both have an order of magnitude of improvement on efficiency.

ACKNOWLEDGMENT

This work is supported in part by a national Natural Science Foundation of China (61100011), a National High Technology Development 863 Program of China (2012AA010901), a National Grand Fundamental Research 973 Program of China (2011CB302504), and a national Natural Science Foundation of China (60921002).

REFERENCES


[18] Open64 open source compiler. http://www.open64.net/