Middleware and Multicore Architecture: Challenges and Potential Enhancements from Software Engineering Perspective

Saleh M. Alnaeli
Department of Computer Science
University of Wisconsin-Fox Valley
Menasha, Wisconsin, USA
saleh.alnaeli@uwc.edu

Amanda D. Ali Taha
Department of Computer Science
University of Wisconsin-Fox Valley
Menasha, Wisconsin, USA
rice9147@students.uwc.edu

Sam B. Binder
Department of Computer Science
University of Wisconsin-Fox Valley
Menasha, Wisconsin, USA
BIND8695@students.uwc.edu

Abstract—An empirical study that examines challenges middleware software systems have to take full advantages of multicore technology is presented. The study is conducted on 9 open source middleware systems containing over 3.39 million lines of code. Each system is analyzed and the inhibitors to parallelization are detected and presented. Additionally, some challenges in analyzing, adapting, and re-engineering middleware systems to better utilize modern multi-core architectures are determined including function side effects. Function side effects categorized based on their types and the complexity they pose in conducting inter-procedural static analysis. The data shows that the most prevalent inhibitor by far is functions called within for-loops that have side effects. Moreover, the study shows that parameters by reference and global variables modification are the most prevalent side effects that poses the greatest challenges in re-engineering middleware systems to improve their parallelizability to better utilize multi-core architectures. That is, conducting accurate program analysis with existing software engineering tools becomes exigent and impractical with those side effects. The study suggests some software engineering techniques (e.g., refactoring) that have the potential to improve the parallelizability of middleware systems.

Index Terms—Middleware, software engineering, multicore architecture, parallelization inhibitors, function side effects.

I. INTRODUCTION

The demand for the distributed systems that run on different machines, operating systems, underlining architectures, and networks, (e.g., the best gaming software, or just about any software “on the cutting edge”) has exploded in recent years. As these applications and systems are getting more complex and sophisticated, middleware is being utilized as major building block for the development of future software systems. Systems and software producers have started relying on middleware as a solution to this demand since it involves minimal programming. Therefore, they can produce cost-effective solutions within a short time and minimal risk [1].

In the software engineering community, middleware has shown to greatly enhance developer’s productivity by hiding much of the intricacy associated with writing complex applications, rather than relying purely on programming language aids and customized library support [2, 3]. Recently, middleware software systems have become available at an ever-increasing rate and have enhanced our way of programming and thinking. No more is middleware just about bringing together different hardware and software components, rather, it is about saving time and helping developers focus on the specific purpose of their application. By allowing them to concentrate on the most important aspects of the systems, developers can save time and still deliver high quality.

With the explosion of modern multicore architectures and their omnipresence in today’s devices, middleware software engineers have been pushed to rethink how the code they write can better utilize the underlying hardware toward multicore-capable middleware. Since most of the existing middleware systems were developed with sequential processors in mind, they typically make inefficient use of the multicore technology. The problem gets worse as the number of cores increases, which may reduce individual core speed, causing drastic slowing in sequential software speed. As a result, many developers are starting to feel that they are at the mercy of middleware designers to ensure acceptable performance for their systems when they use middleware. Knowing this is motivating middleware developers to modify their products to enable parallel execution to take advantage of underlying hardware.

The process of parallelizing a software system is typically done with one of the standard APIs such as OpenMP. These APIs provide the developer with a set of tools to parallelize loops and take advantage of multiple cores and shared memory [4, 5]. Current C/C++ compilers can do a limited amount of automatic parallelization. That is, loops with fixed iteration bounds (i.e., for-loops) can, in certain situations, be directly parallelized by the compiler. Loops without fixed iteration bounds cannot, in general, be parallelized. The auto-parallelization can also take place via a tool prior to compiling. These tools look for for-loops that do not contain any parallelization inhibitors [6, 7].

For middleware software systems, many obstacles can prevent it from being parallelized. Those challenges vary in their impact and nature. Some difficulties are caused by the way developers write their source code and others are due to the environments the middleware work on. However, in the end, the full potential of multicore processor architectures will require a deeper analysis and assessment of the middleware application to adapt the source code for scalable concurrency. Middleware designers that address these concerns will be able to produce a middleware software system that will benefit the most and take
all advantages of the contemporary multicore architectures that are available in almost every computer and device today [8].

TABLE 1. THE 9 OPEN SOURCE MIDDLEWARE SYSTEMS USED IN THE STUDY, AND FUNCTIONS, LOOP CONSTRUCTS FOUND IN THE 9 SYSTEMS

<table>
<thead>
<tr>
<th>System</th>
<th>KLOC</th>
<th>Functions</th>
<th>For</th>
<th>While</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAO</td>
<td>1,543,805</td>
<td>39,720</td>
<td>5,063</td>
<td>2,192</td>
<td>10,000</td>
</tr>
<tr>
<td>ACE</td>
<td>609,033</td>
<td>12,685</td>
<td>2,530</td>
<td>1,133</td>
<td>3,049</td>
</tr>
<tr>
<td>openDDS3.8</td>
<td>326,471</td>
<td>9,185</td>
<td>1,572</td>
<td>671</td>
<td>1,779</td>
</tr>
<tr>
<td>ofono</td>
<td>242,153</td>
<td>7,331</td>
<td>699</td>
<td>307</td>
<td>527</td>
</tr>
<tr>
<td>ApacheAXIS2</td>
<td>221,083</td>
<td>4,213</td>
<td>552</td>
<td>337</td>
<td>807</td>
</tr>
<tr>
<td>CIAO</td>
<td>191,535</td>
<td>5,937</td>
<td>388</td>
<td>69</td>
<td>1,044</td>
</tr>
<tr>
<td>DanCE</td>
<td>102,568</td>
<td>5,292</td>
<td>485</td>
<td>128</td>
<td>345</td>
</tr>
<tr>
<td>xmlBlaster</td>
<td>92,929</td>
<td>2,590</td>
<td>323</td>
<td>152</td>
<td>429</td>
</tr>
<tr>
<td>omniORB</td>
<td>64,451</td>
<td>2,227</td>
<td>320</td>
<td>238</td>
<td>235</td>
</tr>
<tr>
<td>TOTAL</td>
<td>3,394,028</td>
<td>89,180</td>
<td>11,932</td>
<td>5,227</td>
<td>18,215</td>
</tr>
</tbody>
</table>

Here we have undertaken an empirical examination of a variety of middleware open source software systems to better understand the challenges and roadblocks for parallelizing middleware systems.

We are particularly interested in determining the most prevalent inhibitors that occur in these middleware applications and any general trends. This work serves as a foundation for understanding the problem requirements in the context of a broad set of middleware applications. Moreover, the focus of this research is on potential challenges in the most common situations occurring within typical middleware software systems. A count of each of the challenge types are tabulated and this data is then presented to compare the different systems and uncover trends, and make other general observations.

We also examine the prevalence of types of side effects in middleware systems. Furthermore, we are interested in knowing the roadblocks to software engineering static analysis techniques that can be applied to analyze middleware systems. All in pursuance to better understand and eliminate the function side effects, thus improving overall system parallelizability.

This work contributes in several ways. First, it is one of the only large studies on the potential to parallelize middleware software systems. Our findings show that function calls represent the vast majority of inhibitors occurring in these systems. Moreover, this study shows that passing arguments by reference and global variables modifications are the more prevalent types in middleware systems. This knowledge will assist researchers in formulating and directing their work to address those problems for better multicore-capable middleware.

The remainder of this paper is organized as follows. Section 2 presents related work on the topic of middleware performance and parallelization. Section 3 describes the middleware challenges. Section 4 describes the methodology we used in the study along with how we performed the analysis to identify each inhibitor and side effect. Section 5 presents the data collection processes. Section 6 presents the findings of our study of 9 open source middleware, followed by conclusions in section 7.

II. RELATED WORK

The bulk of previous research on this topic has focused on the importance of middleware systems and need to always produce a middleware that meets the industry business requirements. Some other studies were conducted to compare the performance of middleware systems on different platforms varied from hardware specifications to system architectures.

However, no study has been conducted to show the actual challenges in middleware systems to be able to get advantages of multicore architectures on the source code level, nor from a software engineering perspective.

Jik-Soo Kim et al. [2] conducted a study that shows the benefits and overheads associated with middleware systems, in different computational environments and with different workloads. Their study aimed finding a way that can help in making better decisions for tuning the application environment, for selecting the appropriate middleware, and also for designing more powerful middleware systems for more efficient applications in both parallel and distributed computing environments. The study shows none of the potential challenges from the software development perspective.

Yuanfang Zhang et al. [3] conducted an empirical study concerning the real-time system software and their middleware for multicore platforms. Their study presents MC-ORB, the first real-time object request broker (ORB), designed to exploit the features of multicore platforms, with admission control and task allocation services that can provide schedulability guarantees for soft real-time tasks on multicore platforms. The study has not evaluated the source code design nor the obstacles or challenges in parallelizing real-time middleware.

Valerie Issarny et al. [1] conducted a survey that presents related challenges for the middleware in general and related impact on the software design. They show that future applications will need to fit with advanced non-functional properties such as mobility. However, they have not shown any findings related to the middleware and multicore architecture and programming problem domain.

S. Alnaeli, J. Maletic et al [7] conducted an empirical study that examines the potential to parallelize large-scale general-purpose software systems. They found that the greatest inhibitor to automated parallelization of for-loops is the presence of function calls with side effects and they empirically proved that this is a common trend. They recommended that more attention needs to be placed on dealing with function-call inhibitors, caused by function side effects, if a large amount of parallelization is to occur in general purpose software systems so they can take better advantage of modern multicore hardware.

Parallelizing compilers, such as Intel’s [9] and gcc [10], have the ability to analyze loops to determine if they can be safely executed in parallel on multi-core systems, multi-processor computers, clusters, MPPs, and grids. The main limitation is
effectively analyzing the loops [16]. For example, compilers still cannot determine the thread-safety of a loop containing external function calls because it does not know whether the function call has side effects that would introduce dependences.

M. Porter [8] published an article on the journal of military electronics and computing presents the emerging need for techniques that can help parallelizing middleware systems for better utilization of multicore technology. However, the author has not provided any practical way nor facts about the inhibitors or the challenges in parallelizing the middleware systems.

The work presented here differs from previous work on middleware performance and parallelization in that we conduct an empirical study of actual inhibitors to parallelization in the source code level. We empirically examine a number of systems to determine what roadblocks exist to developing better parallelizable middleware that can better work on multicore architecture. The study also evaluates the challenges middleware has from the software engineering perspective in terms of analysis difficulty based on the usage of virtual functions and function pointers, as well as function side effects.

III. CHALLENGES IN MIDDLEWARE PARALLELIZATION

We now discuss the potential challenges in middleware parallelization in the source level code. Inhibitors to parallelization are discussed first followed by the roadblocks to static analysis of the middleware source code. Then, the challenges that middleware might run into if they were parallelized and forced to run on a multicore supported platform with other parallelizable systems (oversubscription).

In this study, a for-loop is considered a free-loop if it does not contain any parallelization inhibitors that are not already solvable with OpenMP. That is, a free-loop does not contain any of the following inhibitors: data dependency, function calls with side effects, or jumps outside of the loop.

A. Inhibitors to Parallelization

Now, we describe the different inhibitors to the middleware parallelization process and the challenges that prevent middleware from running in parallel for better multicore exploitation. Particularly, we are interested in for-loop parallelization inhibitors because, in most applications, the extensive computation is carried out in loops and parallelization APIs like OpenMP can parallelize only for-loops. However, not all for-loops are parallelizable. For example, for-loops whose results are used by other iterations of the same loop will not work properly, and can lead to unexpected and incorrect results [11, 12]. Inhibitors can prevent for-loop parallelization. While some of these are solvable, others are not.

Some inhibitors have a direct solution in Application Programming Interfaces such as OpenMP, and others cannot be solved and demand more complex (conservative) approaches. In this study, a for-loop is considered a free-loop if it does not contain any parallelization inhibitors that are not already solvable with OpenMP.

The data dependency is discussed first followed by function calls with side effects, and then jump statements (e.g., break, goto).

1) Data Dependency

Data Dependency is a well-studied condition which inhibits software systems from parallelization. In many situations, the order of statement execution within the body of the for-loop must be preserved to gain the same expected results from middleware system, as when executed in sequential order. That is, all loop iterations must be independent and no dependency relation should exist between two different iterations.

Several data dependency tests have been developed based on approximation. They are well covered in the literature. All methods are conservative in the case of dependency suspension, or when it is difficult to prove the opposite, so that no unsafe parallel transformation is done.

Example 1: Fibonacci sequence:
// 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
1: array[1]=0;
2: array[2]=1;
3: for(int idx=3; idx<10000; ++idx)
4: array[idx] = array[idx-1] + array[idx-2];

Example 2:
for (i=1; i<100; i++)
{ M[i*2] = Data1[i-1]*0.25;
  Data2[i] = M[2*i-4] + Data3[i+1];
  t = i + 4;
  Data2[i-1] = t*i;
  Data3[Data4[t]] = fun(Data1[Data2[t-1]]);
  Temp = Data4[i];
  constRef[7] = constRef[3];
}

Cases that are reported as data dependency:
1) M[i*2] and M[2*i-4] → Flow-dependence
2) Data3[Data4[t]] and Data3[i+1] → Anti-dependence, assuming that Data4[t] is less than i+1
3) Data2[i] and Data2[i-1] → Output-dependence

Fig. 1. Examples of data dependency detected by the tool ParaStat tool [6, 7]

The main purpose of data-dependency analysis is to detect if the same memory position is used in more than one loop iteration. The majority of dependency analysis algorithms are focused on array references. There are three types of dependency based on the way and the sequence of accessing a memory location. They are 1) flow-dependence (aka true dependence), 2) anti-dependence, and 3) output dependence. A fourth type, input dependence is not considered here because it does not meet the condition that at least one access is a write to memory [13].

Fig. 1 presents two examples that show the data dependency types in loop constructs. In the first example, if the loop is parallelized evenly on 2 cores with two threads (500 iterations each), when the item 501 is calculated it would result into a wrong answer because the items 499 and 500 have not been calculated yet if the second thread acquires the CPU before the first thread. That is, items that come before current item need to be calculated first.
In this work, we take a conservative approach to detecting data dependency and detect all potential dependencies. That is, if we cannot prove that a loop is a free of dependency, we consider it a potential data dependency holder.

2) Function Calls with Side Effects

Another challenge middleware can have in the parallelization context is calling functions or routines that have side effects within a for-loop. Today's compilers cannot parallelize any loop containing a call to a function or a routine that has side effects. A side effect can be produced by a function call several ways, all related to any modification of the nonlocal environment, such as modification of a global variable, or passing arguments by reference [14]. Moreover, a function call in a for-loop or in a call from that function can introduce data dependence that might be hidden [15]. The static analysis of the body of the function increases compilation time; hence this is to be avoided.

As such, it is usually left to the programmer to ensure that no function calls with side effects are used and the loop is parallelized by explicit markup using an API. Generally, a function has a side effect due to one or more of the following: the function modifies a global variable; modifies a static variable; modifies a parameter passed by reference; performs I/O; or calls another function that has side effects. Our approach for calls using function pointers and virtual methods is to assume that all carry side effects. At the onset, this may appear to be a problematic, however conservative, limitation. However, this assumption is supported by empirical analysis we undertook in a previous study [7].

3) Goto and Break statements (Jumps)

Breaks and goto statements are inhibitors to parallelization of for-loops. That is, the loop must be a basic block, meaning no jumps outside the loop are permitted. As such, the occurrence of one of these statements prevents parallelization of the loop. It is very simple to detect all occurrences of break and goto statements in source code so counting them is accurate. A call to exit() can be handled by OpenMP so we do not consider these as loop inhibitors. Also, the same applies to exception handling. Exceptions thrown in a parallel region and caught within the same region are safe for parallelization. Catches can be inserted into those regions automatically if they do not exist. Since there is a known solution for exceptions, we do not consider them as inhibitors [16].

B. Roadblocks to static analysis and inhibitor elimination

In order to improve and assess parallelizability of middleware systems- in the source code level- and eliminate some of detected inhibitors, software engineers need to conduct system analysis. The analysis is usually achieved based on many different approaches. In software engineering, static analysis is one of the most commonly used techniques. The level of difficulty of the static analysis process differs from one system to another based on many factors. For example, systems that are overwhelmed with indirect function calls using function pointers and virtual functions tend to be hard to analyze and expensive in terms of time complexity [17]. In this section, we discuss the challenges middleware systems can have when it comes to inhibitor eliminations and deeper analysis using software engineering tools and techniques built on the top of static analysis approach. Of course, it is important to determine the expected difficulty for better time management should refactoring projects be planned.

C. Oversubscription and ThreadParceling

Another challenge we believe that needs to be considered when talking and studying middleware and multicore architecture and technology is oversubscription and thread parceling. A potential problem of fully parallelized middleware run on a multicore platform is the creation of more threads than there are available cores, which demands more than what the hardware resources can actually serve [11, 12]. The problem gets worse with a system that uses the middleware need to fulfill its processing tasks within limited time budget (e.g., games framing) [11]. When integrating parallel middleware into a parallel software system, care must be taken to ensure that the middleware's parallel approach doesn't negatively affect main software system performance, or with other middleware already in use. According to Intel, those situations are unavoidable and results in an expensive performance penalty. Consequently, software engineers and middleware producers need to consider the hardware limitations when developing parallel systems and middleware.

New modern middleware systems need to be able to handle all expected situations caused by similar issues. General-purpose middleware that does not take to the account the number of the available cores may not be practical anymore. Thus, we believe that the time for customized middleware has come. That is, today’s middleware software engineers should start thinking about practical solutions that fit well with today’s modern computer hardware and architectures, where new discoveries and ideas are being presented in rapid succession. That can be achieved using the capability that APIs like openMP offer for controlling number of cores that middleware can generate during running session. That is, programmers should make their systems adaptable with the available hardware on the system that is hosting the middleware.

D. Programming styles impact on parallelizability

Most of the problems that lead to inhibitors to middleware parallelization are caused by programmers and software engineers. That is, programming styles that make building the system easier can cause problems in parallelizability of the system. For instance, a side effect can be produced by function call in multiple ways. Basically, any modification of the nonlocal environment is referred to as side effect [14, 15, 18].

The static analysis of the body of the function increases compilation time; hence, this is to be avoided. So while a programmer thinks that using global variables and passing parameters by reference can make it easier to do programming, new challenges are introduced in other contexts such as middleware system parallelization [16, 18].

In this study, we also analyze the nine systems in terms of function side effects distribution and prevalence, showing the impact of side effects of the parallelizability of middleware systems. That can help software engineers prioritize their future
refactoring plans to start with the most prevalent side effect, thereby achieving better results.

IV. METHODOLOGY FOR INHIBITORS, SIDE EFFECTS, AND INDIRECT-CALLS

A for-loop is considered a free-loop if it does not contain any parallelization inhibitors that are not already solvable with OpenMP. We used a tool, ParaStat, developed by one of the main authors and used in [7], to analyze loops and determine if they contain any inhibitors as defined in this section. First, we collected all files with C/C++ source-code extensions (i.e., .c, .cc, .cpp, .cxx, .h, and .hpp). Then we used the srcML (www.srcML.org) toolkit [19] to parse and analyze each file. The srcML format wraps the statements and structures of the source-code syntax with XML elements, allowing tools, such as ParaStat, to use XML APIs to locate such things as for-loops and to analyze expressions. Once in the srcML format, ParaStat iteratively found each for-loop and then analyzed the expressions in the for-loop to find the different inhibitors.

A count of each inhibitor per loop was recorded. It also recorded the number of free-loops found. The final output is a report of the number of free-loops and for-loops with one or more types of inhibitors. Finally, all functions were deeply analyzed and side effects were detected and counted and their distributions determined.

Findings are discussed later in this paper along with limitations of our approach.

V. DATA COLLECTION

Software tools were used, which automatically analyze functions and loops and determines if they contain any inhibitors or side effects. The srcML toolkit produces an XML representation of the parse tree for the C/C++ systems we examined. ParaStat analyzes the srcML produced using XML tools to search the parse tree information using system.xml from the .NET framework. The body of each loop and function is then extracted and examined for each type of inhibitor in loops or side effects in functions. For the loops, if no inhibitors exist in a for-loop it is counted as a free loop otherwise the existence of each inhibitor is recorded. The systems that were chosen in this study were carefully selected to represent a variety of middleware open source systems developed in C/C++. These are well-known middleware systems to both academia and research communities.

VI. FINDINGS, RESULTS AND DISCUSSION

We now study the challenges in parallelizability of nine open-source middleware systems including inhibitors to parallelization and function side effects. TABLE 1 presents the list of middleware systems examined along with number of files, and LOCs for each system. TABLE 1 shows a count of how many for-loops were found in each system and for comparison the number of while loops and number of functions in each middleware system. One item of interest in is that, all of the middleware systems show a much larger use of for-loops than while-loops. This gives promise for potential parallelization through the use of APIs such as OpenMP.

<table>
<thead>
<tr>
<th>System</th>
<th>Function with Side Effect</th>
<th>Jumps</th>
<th>Data Dependency</th>
<th>Free for-loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAO</td>
<td>667 (13%)</td>
<td>360 (7%)</td>
<td>296 (6%)</td>
<td>3,850 (76%)</td>
</tr>
<tr>
<td>ACE</td>
<td>303 (12%)</td>
<td>412 (16%)</td>
<td>131 (5%)</td>
<td>1,765 (70%)</td>
</tr>
<tr>
<td>openDDS</td>
<td>210 (13%)</td>
<td>99 (6%)</td>
<td>56 (4%)</td>
<td>1,247 (79%)</td>
</tr>
<tr>
<td>ofono</td>
<td>259 (36%)</td>
<td>191 (27%)</td>
<td>42 (6%)</td>
<td>303 (43%)</td>
</tr>
<tr>
<td>ApacheAXI</td>
<td>439 (80%)</td>
<td>66 (12%)</td>
<td>9 (2%)</td>
<td>98 (18%)</td>
</tr>
<tr>
<td>CIAO</td>
<td>18 (5%)</td>
<td>12 (5%)</td>
<td>12 (3%)</td>
<td>342 (88%)</td>
</tr>
<tr>
<td>DanCE</td>
<td>5 (1%)</td>
<td>19 (4%)</td>
<td>26 (5%)</td>
<td>442 (91%)</td>
</tr>
<tr>
<td>xmlBlaster</td>
<td>152 (47%)</td>
<td>29 (9%)</td>
<td>24 (7%)</td>
<td>150 (46%)</td>
</tr>
<tr>
<td>omniORB</td>
<td>9 (3%)</td>
<td>69 (22%)</td>
<td>83 (26%)</td>
<td>183 (57%)</td>
</tr>
<tr>
<td>Average</td>
<td>24%</td>
<td>12%</td>
<td>7%</td>
<td>63%</td>
</tr>
</tbody>
</table>

A. Design of the empirical study

This study focuses on three aspects regarding middleware parallelizability in terms of inhibitors and challenges in system analysis. First, the percentage of for-loops containing one or more inhibitors; this gives a handle of how much of the system could be readily parallelized by a compiler or other automated tool. Second, we examine which inhibitors are most prevalent. Third, we seek to understand the cause of parallelizability inhibitors and as a case study, we focus on function side effects.

We determine the percentage of functions containing one or more side effects; and then we show the distribution of the side effect types in each system. This can give middleware software engineers an idea about the most prevalent side effect type to consider as priority, should they plan for refactoring operations in aims to improve their systems. We propose the following research questions as a more formal definition of the study:

RQ1: What is a typical percentage of for-loops that are free loops (have no inhibitors)?

RQ2: Which types of inhibitors are the most prevalent?

RQ3: Which types of function side effect are the most prevalent? What are their distributions? To what extent are side effects being present in middleware systems?

We now examine our findings within the context of these research questions.

B. Percentage of free for-loops

Table 2 presents the results collected for the nine middleware systems. We give the total number of free for-loops along with the percentage of all for loops we detected.

Table 2 shows the percentage of free-loops computed over the total number of for-loops. As can be seen, free loops account for between 18% and 91% of all for loops in these systems, with an overall average of 63%. However, in general, the percentage is high for most of the middleware systems compared to the situation in general purpose systems that were studied in our previous study [7]. That is, on average, a big portion of the
detected for-loops in these middleware systems could potentially be parallelized. This addresses RQ1.

![Chart](chart.png)

Fig. 2. The average percentage of free for-loops and non-free-for-loops that contain at least one inhibitor of the given type, over all 9 systems. Loops may contain more than one inhibitor type.

**C. Parallelization inhibitors Distribution**

We now use our finding to address RQ2. Table 2 presents the details of our findings on the distribution of inhibitors in studied middleware systems.

It clearly counts each of the inhibitors that occur within for-loops. Many of the for-loops have multiple inhibitors (e.g., a data dependency and a jump). As can be seen, function-call inhibitors are by far the most prevalent across all systems. For most of the systems this is followed by jumps and then data dependency, thus addressing R2. We lumped the jumps together. This is similar to our findings with general purpose systems, though middleware systems show better results by noticeable margin [7]. The findings show that DanCE has great potential to be parallelized and take advantage of multicore architecture. In contrast, ApacheAXIS seems to have a big problem when it comes to parallelization context.

Fig. 2, presents the average percentage, over all 9 middleware systems, of for-loops that are not free and contain one or more inhibitors. This gives a clear view of which inhibitor occurs most frequently. We see that the trend is likewise similar via this perspective. Function-call inhibitors are by far the most prevalent. On average, we see the next most prevalent inhibitor is the jumps followed next by data dependency.

**D. Function Side Effects**

In order for us to address RQ3, we examined the nine middleware systems with respect to types of function side effects that cause the most prevalent parallelization inhibitor (calls with side effect within loop body), and their distribution. Fig. 3, presents the average percentage, over all 9 middleware systems, of functions that are not clear and that contain one or more side effects. This gives a clear view of which side effect occurs most frequently in all middleware systems. We see that the trend is not similar through this perspective.

For TAO, ACE, opJAO, and xmlBlaster, global variable modification is the most prevalent. In comparison, ofono, ApacheAXIS2, DanCE, and omniORB have more parameters that are sent by reference. We are not sure why these trends are inconsistent but we will consider this for future work. On average, we see the most prevalent side effect is the parameters by reference followed next by global variable modification. Next, the percentage of functions that call another functions that hold side effects, and then functions that contain in/out operations.

![Bar Chart](bar_chart.png)

Fig. 3. The distribution of side effect in all 9 middleware systems. Numbers represent percentages. Function may contain more than one side effect type.

**VII. CONCLUSION**

This study empirically examined the challenges middleware software systems have to take full advantage of multicore technology. The study was conducted on 9 open source middleware systems containing over 3.39 million lines of code. There are no other studies of this type currently in the literature.

We found that the greatest inhibitor to parallelizing middleware systems is the presence of function calls with side effects. As such, more attention needs to be placed on dealing with function-call inhibitors if a large amount of parallelization is to occur in middleware software systems so they can take better advantage of modern multicore hardware. Our results
show some indication that this is a common trend. Additionally, we empirically showed that coding style can play a big role in advancing a system’s parallelizability, with developers of middleware systems causing more challenges to the parallelization process by using parameters by reference in their functions and having their functions modify global variables that can be easily handled outside of the functions. That is at least partially due to development teams and organizations not focusing on developing software in a way that could one day take advantage of parallel architectures. However, the recent ubiquity of multicore processors gives rise to the need to educate middleware producers and developers and make them more aware of the problems and inhibitors to parallelizing their code.

From the results of this work, we recommend the middleware developers community to develop techniques that assist in removing jumping statements along with the identification of functions with side effects (in the context of parallelization). We recommend that the middleware community start thinking about customized middleware.

Finally, we believe that new modern middleware systems need to be able to handle all expected situations caused by oversubscription and thread parceling.

General-purpose middleware that does not take into account the number of the available cores may not be practical anymore. Therefore, today’s middleware software engineers should start working on the development of parallelizable middleware that can fit well with today’s modern computer hardware and architectures. Nearly all of today’s systems are parallelizable and new discoveries and ideas are being presented in rapid succession.

This can be achieved using the capabilities APIs like openMP offer for controlling the number of threads that middleware can generate during running session, making their systems adaptable to the available hardware on the host system so as to not negatively affect other systems performance.

ACKNOWLEDGMENT

This work was supported in part by UW-Fox and CSEPA.

REFERENCES


