The Autonomic Performance Prescription

Karin Högstedt,
AT&T Labs-Research, NJ[1]
{karin@research.att.com}

Doug Kimelman, VT Rajan, Tova Roth, Mark Wegman,
IBM T.J. Watson Research Center, Hawthorn, NY 10532
{dnk, vtrajan, tova, wegman}@watson.ibm.com

Nan Wang
University of Maryland[1]
{nwang@cs.umd.edu}

Abstract

Programmers increasingly program by component assembly. This is despite the fact that the use of components tends to decrease the performance of programs. The programmers who design the components do not know anything about the environment the components will be used in, and cannot, therefore, tailor the components for each specific use. The user of a component is, for modularity reasons, not supposed to know anything about the implementation. So, no one and no system chooses the most efficient implementation. Our work lays out a foundation for reformulating the implementation of components, so that modularity is preserved and yet the assembled system is optimized. Our approach amounts to automatically choosing among a number of candidate implementation alternatives for each of the data structures and communication mechanisms that the components use during their interactions. Some of our initial work in this area has shown how to transform one example of component interaction optimization into a graph problem, and has proposed analysis of dynamics and fast graph reduction heuristics as an efficient technique for solving the problem.

Our hope is that this work will inspire debate that ultimately will lead to a new area of investigation, within the field of compiler-performed optimization, with the potential to achieve orders of magnitude improvements in the performance of component-oriented programs.

1 Introduction

While computers have been getting exponentially faster as time goes on, the applications we use do not appear to be on the same improved performance curve. We believe a part of the problem is from what we have labeled the "Encapsulation Performance Problem." The Encapsulation Performance Problem (EPP) occurs when programmers write programs by assembling components. We do not want to discourage this style of programming. It is important both to make programmers more productive but also to ensure consistency among applications.

The Encapsulation Performance Problem is in some sense a question of who is in charge and can be boiled down to three observations:

1. A component writer cannot know the characteristics of the application the component will be used in.
2. The application writer for reasons of modularity and good software engineering should not know details of the implementation of a component.
3. The most efficient choice of component implementation is different under different circumstances.

Most writers of complex software will recognize this problem. Programmers have lost intellectual control of their programs, in a serious sense. It was possible in the 1980's to understand the details of an entire application. Moreover, in contrast to programs written predominantly before say 1990, most of the time spent in executing an application is spent in parts of a program not written by the application writer. We have observed that it is not uncommon today for an expert programmer who understands a framework in more detail than the original programmer to improve program efficiency by *decimal orders of magnitude*, by invoking different components, by invoking them in different ways, or even by altering the implementation of a component. For example, choosing among alternative dictionary, directory or registry components, without knowing about the internals of the components, may affect performance adversely. In some cases, anticipated use may dictate a component with a hash-based implementation, rather than a tree- or list-based implementation, while in other cases the overhead of such an implementation may not be warranted. Furthermore, developing a component without knowing anything about the applications in which it will eventually be used also can result in inefficiencies. For example, developing a component for a distributed application without knowing anything about the topology in which it will be deployed can lead to severe performance degradation due to inappropriate choices with respect to placement, caching, and replication. In 1968 Doug McIlroy, in his famous paper, ``Mass Produced Software Components'' succinctly presented his vision of software components [McIlroy68]:

> *The most important characteristic of a software components industry is that it will offer families of routines for any given job. No user of a particular member of a family should pay a penalty, in unwanted generality, for the fact that he is employing a standard model routine. In other words, the purchaser of a component from a family will choose one tailored to his exact needs. He will consult a catalogue offering routines in varying degrees of precision, robustness, time-space performance, and generality. He will be confident that each routine in the family is of high quality - reliable and efficient.*
>
> ----Doug McIlroy

The software industry has come a long way to realize some of McIlroy's vision. However users are paying substantial performance penalties for using the non-customized components. The C++ Standard Template Library (STL), Java Collection Class, Microsoft Foundation Class, and other packages provide high quality off-the-shelf reusable components. A purchaser of these packages must pick and choose components and tailor them

[1]Work done while at IBM T.J. Watson Research Center.

to his or her needs in order to achieve any degree of efficiency. Even the efficiency achieved is only because we have, over many years, managed to make tradeoffs in performance a part of the API and forced users to deal with it. Languages like Smalltalk avoid this. Moreover new applications like J2EE application servers are notoriously difficult to tune because the programmer does not understand the consequences of the different choices they make.

Traditional statement-level and procedure-level optimizations are inadequate for the kinds of software systems that are built from off-the-shelf components, because the differences in the implementations are often much more fundamental than traditional optimization deals with. Examples are changes in algorithms, data structures and placement. These different implementations must be specified by the component developer. Even languages like SETL which explored very high level optimizations only optimized "components" such as sets that were supplied by and known intimately by the compiler. New means are necessary to optimize component assemblies.

Optimization at the higher levels of programming as required by component assembly is a largely unexplored area in the field of compilers. The value of different implementations must be judged based on the frequency with which they do various operations and the frequency with which they interact with other objects. This paper is a first attempt at identifying and working out many of the issues needed for the kind of optimization appropriate for programming by component assembly.

To solve the EPP we propose the Autonomic Performance Prescription(APP). Here the answer to the question of who is in charge of the efficiency of the system is the system itself, hence the word "autonomic". In the APP we carefully orchestrate who is in charge of various decisions so as to preserve important software engineering values like information hiding and encapsulation. The basic steps of the APP, which we will detail in later sections of the paper are:

1. The component writer supplies multiple implementations of their component , together with a mechanism for inferring the cost of using those different implementations when a program is run.
2. A program using those components is run (or at least partially run) and the impact of using the differing implementations of the components is gathered.
3. An ideal set of implementations for the previous run is determined by the system
4. The system uses the ideal choices for the previous run to determine what choices should be made in the future.

Our main contribution is a framework in which a component developer provides multiple implementations of a component and a profile directed compiler/runtime system chooses the right implementation for a particular need and allowing optimizations. Within our framework we present the following mechanisms:

We gather profile information. (Section 2) We use the profile information to construct a graph, called the Object Affinity Graph(OAG), and show that the optimal choice of implementations for the profile run can be determined by solving a graph cutting problem on the OAG. (Section 3) A set of new heuristics that will systematically reduce the size of a graph while preserving optimality of the min-cut. In all the experiments we these done they have completely reduced the graph, but other reductions are possible at the possible expense of optimality. (Section 4)We show how to identify characteristics of objects from the profile run that can be used to distinguish between objects that should have different implementations in future runs. (Section 5)We show how the framework can be used to solve a number of related problems. (Section 6)Present some preliminary empirical result and experience that suggest our framework is feasible. (Section 7)

Section 8 discusses some of the related work, and finally Section 9 gives our conclusions.

In this one paper we cannot solve all the issues involved in optimizing systems built using components. The last section of this paper points to many new areas which when elucidated and solved could profitably be combined with the ideas presented in this paper.

2 Multiple Implementations of Library or Component

This section will show by example how a component or library writer can provide multiple implementations of their component. According to the APP, they will be responsible not only for providing multiple implementations but they will also be responsible for providing code that determines the cost of using those implementations. The cost comes in two flavors 1) the cost incurred by an object (an instance of the component) when using the implementation by itself and 2) the cost of the interactions of two objects when the objects have different implementations. At run time the system will be responsible for using the information gathered during the profiling runs from the costs functions to choose good implementations.

While in some cases the decision about the appropriate choice of implementation of an object can be made by examining the kind of use that is made of it, in other cases this decision would have to be coordinated with the corresponding decisions made for other objects that interact with this object. The later happens when the costs for interactions are relatively high. For example, the decision about the placement of an object in a network of computers is affected by the decision about the placement of other objects that interact with this object. The problem thus becomes a combinatorial one, which in our experience even an experienced programmer cannot make well without the kind of analysis we discuss in this paper.

We have provided a snippet of code implementing part of a general set class, in a language modeled on Java (see Figure 1). In this class there are two implementations, `Hashmap` and `Treemap`. Each implementation has the same three methods with the same arguments, though had we finished the code, the methods would have had different bodies. For each method in each class there is a corresponding cost. Again the signatures of the costs are the same, but their bodies are different.

Cost methods are side effect free. In other words you can and in fact need to run the cost methods for one implementation at the same time you actually use a different implementation. Moreover a cost method cannot use a value which is computed only in one implementation. Hence we declared `size` outside the scope of the two implementations.

The application programmer is supposed to be unaware of which implementation the system chooses. But, methods may take as arguments objects which have been given different implementations. Thus we also provide a coercion function to convert one to the other. When a method with a single argument, which can have multiple implementations, returns a value that can also have multiple values, we may also need to use a coercion as we do for the clone method in Figure 2. There is a cost, of course, to doing these coercions and that is computed by the coercionCost method. While we might need a different coercion for each pair of implementations., for simplicity we have here provided only one coercionCost method.

```
Class Map
size{}
     Implementation Hashmap {
       Methods {
               insert(val),
               find(val),
               findGreaterThan(val), etc. }
       Costs {
               insert { return( 50 ) }
               find { return( 45 ) }
               findGreaterThan
                       { return( 10*size of the
map) }
                 }
       }
     Implementation Treemap (Default){
       Methods {
               insert(val),
               find(val),
               findGreaterThan(val) etc. }
       Costs {insert, find, and findGreaterThan {
               return( 30*log of size of the map
) }  }   }
               Coercion  {
          Coerce( Map from, Map to) {
  // Needs to be provided for every pair of
implementations       }
          CoercionCost{ return( 150*size of the
map  )  }    } }
```

Figure 1

As in Figure 2, a client that uses the Map class will simply create instances of the Map class. Without our analysis the client will get the default implementation (Treemap in our case). The implementer of the set class in this case chose Treemap because it had the best worst-case performance and so was likely to run acceptably during the profile run. Using our framework a runtime system or a compiler will automatically ensure that the client, after the profile run, gets the best of the set implementations defined by the nested classes.

The code in Figure 2 creates a number of set instances, a, b, and c, and performs different kinds of operations on different instances. Using the analysis presented in this paper, a runtime system or a compiler will assign the right implementation for each of these set instances. During a profile run of the client code

```
Map a, b, c = new Map();
Loop 100 times {
       a.insert(rand);
       b.insert(rand);
}

Loop 200 times {
       a.find(rand);
       b.findGreaterThan(rand);
}

c=a.clone();
```

Figure 2

information is gathered. . During that run the default implementation (i.e., Treemap) will be used. The runtime system will invoke the cost methods for each implementation to compute the cost of using an implementation for a set instance. This cost includes the costs for invoking various operations that the client code invokes. After the first loop in Figure 2, the size for both a and b will be 100. So the cost of a single findGreaterThan in the second loop would be 30*log(100) if the implementation is a Treemap. It would be 10*100, or 1000 for a Hashmap. Since there are 200 iterations of the second loop the cost is really 200 times that.

In the rest of the paper we will show how to transform the above *decision problem* to a graph cutting problem and also use characterization information to efficiently choose the right implementation.

3 Building Object Affinity Graph

We will model the decision problem of determining the optimal implementation choices for the profiling runs using an Object Affinity Graph (OAG). A OAG is a weighted undirected graph with two kinds of nodes, called *implementation* nodes and *instance* nodes. Implementation nodes represent the different implementation of a class and the instance nodes represents different instances of a class. For the above client code there are two implementation nodes, one for each of the two different map implementations, and 3 instance nodes, one for each of the instances of map allocated as the program in figure 2 is run. In this simple case, each of the map objects can be labeled with a variable name, but this is for pedagogical reasons and labeling doesn't make sense when objects are allocated in the heap.

Skeleton OAG

To construct an OAG we first run the client program using sample data and generate a profile. During this phase we will use the default implementation (i.e., Treemap) for the set instances. For each instance created (in the profile) we create a instance node. We also insert an edge from each implementation node to each instance node when the first method with multiple implementations is invoked on the corresponding instance. In the graph in figure 3, we left off the edges from c, because no methods of c are executed. As discussed above the clone operation deals with two objects (the newly created object and the object it was created from). When an operation deals with two

objects and a coercion may be forced we insert an edge between the corresponding instance nodes. An edge between an implementation node and an instance node indicates that the set instance corresponding to the instance node can be given the implementation defined by the implementation for the map class. An edge between two instance nodes indicates that one instance of the map class interacts with another instance of the Map class.

Edge	***Cost***
A,C	150*size =150*100
Hashmap,A	MAXC - (100*50 + 200*45)
Hashmap, B	MAXC - (100*50 + 200*10*mapsize)
Treemap, A	$maxc-\sum_{i=1}^{100} 30*\log(i) + 200*30*\log(200)$
Treemap, B	$maxc-\sum_{i=1}^{100} 30*\log(i) + 200*30*\log(200)$
Hashmap, C	0 (edge not shown)
Treemap, C	0 (edge not shown)

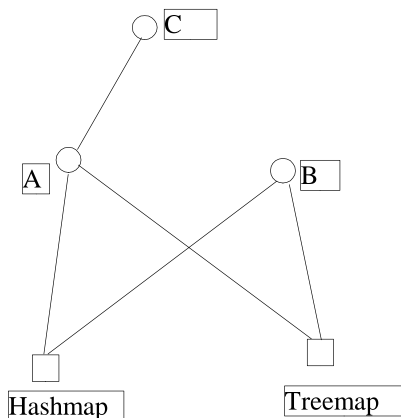

Figure 3

Affinity and Interaction Cost

An instance of a class can be assigned to any one of the implementation defined in the class. We model this by creating an edge from the corresponding instance node to every implementation node. We "decorate" the edges with weights that indicate "affinity" of an instance to an implementation. The value of an edge weight can range from a zero value, indicating no affinity, to an infinite value , indicating maximum affinity. Recall that we also insert edges between any two instances whenever there is an interaction between them (e.g., cloning interaction). The weights on such edges correspond to the amount of interaction between the corresponding instance nodes. A key question is: How to compute edge weights?

We used profiling for computing edges weights. Recall that a component developer also defines cost functions for different operations in different implementation and cost functions for transformations. The runtime system, during the profiling, will use these cost functions to compute edge weights. During profiling we gather information on the costs for each object. In particular we compute the sum (or were we to sample, the expected sum) of the costs for each object for each implementation. So, for a particular object o, we would sum up the values from all the invocations of o.insert and o.find etc. under one implementation and, separately, the sum for all the other implementations. We would also for each pair of objects sum all the costs generated by the coercionCost methods and determine the additional cost of using different implementations for those two objects.

These two types of costs have very different meanings. In one case it measures how much we want two objects to have the same implementation. In the other case it measures how much we don't want an object to have a particular implementations. We need to use the same meaning and hence we will compute the *affinity* between an object and another object as well as the affinity of an object for an implementation. We want affinity to be negatively correlated with the cost of using an implementation, but we want it to be a positive number. Hence we find the maximal cost, maxc, of implementation for an object (where that cost is the sum of the costs produced in the profiling runs). The affinity of an object for an implementation will be maxc minus the cost of that implementation for that object. The affinity of an object for another object will be the sum of the cost of the coercions done in profiling run.

In figure 3 we give an example of how the weights of the edges would be computed for the program in figure 2. The size of the maps a and b are both 100 after the first loop in figure 2. The coercionCost for the statement "`c=a.clone();`" is 150 times the size of the map or 150*100, This mean a cost of 15,000 will occur as the program is run if a and c are given different implementations. There are 100 insertions and 200 find operations on a. Thus the cost of a being a hashmap is 100*50 + 200*45. The edge (a,c) measures the cost of a and c being **different**. The cost of a being the **same** implementation as any other hashmap is 5000+9000.

We label the edge from a to Hashmap as maxc-(5000+9000). This is because we want the label on the edge from a to Hashmap to reflect the cost of a not having an implementation of a Hashmap. We arbitrarily increase the cost of using any implementation by maxc and then label the edge from a to the hashmap by Maxc minus the cost that would be incurred if it were the same as a hashmap.

4 Graph Partitioning

In this section first we describe how to partition the OAG so that we obtain an optimal assignment of class implementation to class instances. The cost of any assignment of implementations. to the instances will be equivalent to the sum of the weights of edges between nodes with different implementations., including the cost of the edges between a node and the implementations. it was not assigned. In graph theory this corresponds to the graph partitioning problem. Stone realized that graph partitioning was the algorithmic problem being solved when figuring out optimal distribution of functions on a network[Stone 77]. In our notation Stone recognized that when trying to give an object an implementation on one machine rather than an implementation on the other machine, you end up with a graph partitioning problem.

In graph theory our implementation nodes are called *terminal* nodes and our instance nodes are called *non-terminal* nodes. Two terminal nodes cannot be in the same partition.

In general graph partitioning is an NP-hard problem [Dahlhouse et.al] but we will give several transformation heuristics that systematically reduces the graph to smaller and smaller graphs. The time of complexity of the sequence of transformation is almost linear in practice, and we present experimental results to back this claim.

Each of our heuristics preserves optimality and reduce the size of the graph. If you only use these heuristics (perhaps repeatedly) and are able to reduce the graph to a point where the graph has only one possible partitioning, you have found the optimal solution.

We start by introducing some notation. An OAG graph $\mathbf{G} = \{\mathbf{N},\mathbf{E},\mathbf{T}\}$ is a graph with a set of *non-terminal nodes* **N** and a set of *terminal nodes* **T**, such that $\mathbf{N} \cap \mathbf{T} = \phi$. Recall that non-terminal nodes are the notation in mathematics for what we've been calling instance nodes, and terminals are the notation for implementation nodes. Each edge has a weight, denoted $\mathbf{w}(\mathbf{e})$ for an edge $\mathbf{e} \in \mathbf{E}$. Figure 1 illustrates such a graph. At the end of our partitioning algorithm we will have assigned each non-terminal node to a terminal node. A graph partition is a set of non-terminal node that is assigned to a terminal node. The set of edges between nodes assigned different terminal node is called the *cut set.* The *weight of a* cut is the sum of the weights of the edges of the cut set. Our problem is to find an assignment that minimizes the cost of the cut.

One of the basic primitives used in our algorithms is the act of contracting an edge. The contraction of $(\mathbf{x},\mathbf{y}) \in \mathbf{E}$ corresponds to replacing the vertex **x** and **y** by a new vertex **z**, and for each vertex **v** not in $\{\mathbf{x},\mathbf{y}\}$ replacing any edge $(\mathbf{x},\mathbf{v})$ or $(\mathbf{y},\mathbf{v})$ by the edge $(\mathbf{z},\mathbf{v})$. The rest of the graph remains unchanged. If the contraction results in multiple edges from node **z** to another node **w**, they are combined and their weights added. Contraction of edges is commutative. Hence, given a subset of edges of a graph, the result of contracting the edges is independent of the order in which they are contracted. [Motwani and Raghavan 95]. Note that whatever the minimum cost cut is, it *can* be obtained by contraction of some subset of the edges in **E**. Those edges that are not contracted form the cut.

The main idea of the heuristics is that little by little we reduce the size of the given graph. Suppose we know that a particular edge $(\mathbf{x},\mathbf{y})$ need not be present in a min-cut of the given graph. We can then contract the edge, and the weight of a min-cut of the resulting contracted graph is the same as that of the original, larger graph. Furthermore, the contracted graph has one less node, so it is simpler. The question is: how do we determine that a particular edge need not be in a min-cut? In the remainder of this section we present several lemmas for identifying conditions from which we can infer that a particular edge need not be in a min-cut of a given graph i.e. that there is at least one min-cut that does not include that edge.

We next give five heuristics (presented as lemmas) that allows us to reduce an OAG systematically to a smaller and smaller OAG. The proofs of these lemmas are given in Appendix A.

Lemma/Heuristic 1. Dominant Edge. Suppose graph **G** has a non-terminal node **n**, with edge **e** being the heaviest among the edges adjacent to **n**, with edges $e_1, e_2, \ldots, e_r$ being the edges connecting **n** to non-terminals, and with edges $e'_1, e'_2, \ldots, e'_k$ being the edges connecting **n** to nodes which are terminals. Further suppose that $w(\mathbf{e}) \geq \Sigma\ w(e_i) + Max(\ w(e'_1), w(e'_2), \ldots, w(e'_k))$. Then there is a minimum cut not containing edge **e**. That is, we can contract edge **e** to obtain the new graph **G′** that has a minimum cut with the same value as a minimum cut of **G**.

Lemma/Heuristic 2. Independent Net. If the communication graph between objects can be broken into two or more independent nets, then the min cut of the graph can be obtained by combining the min-cut of each independent net.

An Independent Net is a subgraph of the original graph such that there is no edge between the non-terminal nodes in this subgraph and nodes outside this subgraph.

Lemma/Heuristic 3. Terminal Cut. Let a terminal cut $\mathbf{T}_i$ be the set of all edges between a terminal $\mathbf{t}_i$ and non-terminal nodes **N.** Let $\mathbf{W}_i$ be the sum of the weight of all edges in the terminal cut $\mathbf{T}_i$. Let $\mathbf{W}_i$'s be sorted so that $\mathbf{W}_1 \geq \mathbf{W}_2 \geq \mathbf{W}_3 \geq \ldots$. Then any edge which has weight greater than $\mathbf{W}_2$ cannot be present in the minimum cut, and hence can be contracted.

Lemma/Heuristic 4 Zeroing. Assume that a non-terminal node **n** has edges to each of the **t** terminals in **T** with weights $\mathbf{w}_1 \leq \mathbf{w}_2 \leq \ldots \leq \mathbf{w}_t$. Node **n** may also have edges to other nodes in the graph. Reducing the weights of each of the *t* edges from **n** to the terminals **T** by $\mathbf{w}_1$ does not change the assignment of nodes for minimum cut. It reduces the cost of the minimum cut by $(t\text{-}1)\ \mathbf{w}_1$.

Zeroing reduces the weight of each terminal cut, thus permitting more edges to be contracted using the Terminal Cut heuristic based on Lemma 3.

Lemma/Heuristic 5. Articulation Point. Let **S** be a subset of the non-terminal nodes of **N** that would be disconnected from all of the terminals if non-terminal node **n** were deleted. Then none of the edges between the nodes in **S**, nor the edges between the nodes in **S** and **n** can be in the minimum cut and they can all be contracted.

Algorithm 1

```
1  While (the graph contains a non-terminal nodes) do
2      If (there exists a heuristic which can be applied) then
3         apply the heuristic to reduce the graph:
4      Else
5         choose an edge with maximum weight and contract it;
```

If a graph can be reduced to only terminal nodes by executing only the statement on line 3, then the graph can be partitioned optimally. The order of the heuristics affects the efficiency of the algorithm (see [Högstedt et al.] for details).

In practice, these heuristics have completely reduced numerous (realistically complex) graphs (see section 7). However, in cases where these heuristics do not completely reduce the graph, any traditional approach can be applied to the remaining derived graph (or contract an edge randomly as in step 5). Some traditional approaches, such as branch-and-bound, guarantee an optimal result, but their complexity is exponential. This might be acceptable if the remaining derived graph is small enough. Other approaches, such as the randomized algorithm [Karger and Stein 93] have lower complexity, but offer only a small (but finite) probability of optimality. In this algorithm an edge to be contracted is chosen at random with a probability proportional to

its weight. Additional independent trials can be used to increase the probability. In each case, after each contraction due to one of these algorithms, we can apply any applicable heuristics until no further reduction of the graph is possible. This could significantly enhance the performance of the algorithm.

5 Future choices based on past choices

When a program creates a fixed number of objects each time it is run and those objects behave the same way from run to run, the techniques described in the previous sections suffice to determine an optimal choice amongst implementations. However, in the much more common case, when a program creates a different set of objects on each run, we need to determine for each of the new objects what implementation it should use on this run based on the information from the profiling run. Because the implementation of an object for its lifetime is determined at create time we can only use characteristics of the object and its creation available at create time. We call the problem of determining the implementation of an instance from these characteristics the *characterization* problem.

At create time the runtime system knows 1) which statement is doing the allocation, 2) the objects which are the value of the "this" pointer for every call in the call chain leading to this allocation, 3) the arguments to every call on the stack, and it can also record various characteristics in any of the afore mentioned objects when they in turn are created. Ideally the system will, based on previous runs, determine that some combination of these gives sufficient information that when an object is allocated the right (or at least not disastrously wrong) implementation choice can be made.

```
Map a, b, c = new Map();
Loop 100 times {
       a.insert(rand);
       b.insert(rand);
}

Loop 200 times {
       a.find(rand);
       b.findGreaterThan(rand);
}
Map[] D = new Map[2];
       D[0] = a;
       D[1] = b;
Map[] E = new Map[2];
for (i = 0; i <2; i++)  {
1:     E[i] = D[i].clone;
       for (j = 0; j< smallno; j++)
       E[i].find(rand);
}
for (i = 0; i <2; i++) {
2:     E[i] = D[i].clone();
       for(j = 0; j < largeno ; j++)
              E[i].findGreaterThan(rand);
       }
```

Figure 4

Consider, for example, the following variation of the program given in figure 2 (figure 4). The program clones the variable D twice, assigning each time to E. On the first occasion, only a small number of find method calls are made to the variable. Therefore the variable E should choose the same implementation as the variable it clones from. On the second occasion, a large number of findGreaterThan methods are called, therefore the cloning should always choose the TreeMap implementation. We can characterize the choice of implementation of objects created at line 1 the object it is cloned from. That is to say, we use both line and the argument of the clone operation to do the characterization of the object created in statement 1, and just the line at statement 2.

There are three possible problems we could face in getting to the optimal partition based on the characterization. 1) There may not be enough information available at allocation time to determine the optimal implementation as defined by the partitioning of the OAG, 2) it may be combinatorially difficult to determine what information available yields an optimal choice of implementations, and 3) it may be that from previous runs it appears that certain information at create time is an important determiner of the implementation, but that is purely a coincidence and using that information is valueless. Point 3 is a classic problem called *over fitting* in the AI domain.

At create time there is a large amount of information available. We will not describe an algorithm to use all of it, in part because such an algorithm would have problems with over fitting. We will now describe a simpler set of information that we believe is very likely to be relevant. This is simpler but not simple enough to completely avoid a potential nasty combinatorial problem which we will address later in this section.

Many of the objects allocated in the program are given a characterization. Objects whose implementation choice we have to make are characterized by their implementation. Some objects, which do not have a choice of implementation, are also characterized to make the job of characterizing other objects easier. For example if an object allocates a number of strings which should be in ASCII, it might be labeled ASCII as its characterization, even though it is not a string. An object which is the first argument to the create function for a number of objects of a given implementation might also get that characteristic. Some objects which cannot be helpful in determining characteristics are not characterized.

For each set of objects allocated at a particular allocation site, we will construct an allocation strategy. A simple strategy might be that all objects allocated at that site, are given a particular implementation. A different strategy might be that they get an implementation based on their creator's characteristics. Two other strategies may make sense. The object might be characterized to have the same characteristics as one of the arguments to the creation function.

One last strategy is required by the prevalence of factories, abstract factories, and even class objects in languages like Java. Here all creation is done through an intermediate object. The information needed to characterize the objects being created exists at the call to the abstract factory but not inside the factory. For example, all objects created by a call to an abstract factory from a particular allocation site perhaps should have the same implementation. Thus, we consider all the strategies described in the previous paragraph but using the information one level up in the stack frame. In some cases in languages like Java it may be necessary to go two levels up in the stack frame to accommodate class objects as well as abstract factories.

On the assumption that we have enough information to characterize an object and get good performance, we present a simple greedy algorithm to do that characterization. To simplify the presentation of the algorithm we will assume that it is sufficient to use allocation site and the implementation of the object that causes the creation to occur. We will call that object the creating object to distinguish it from the created object. Moreover, we will assume that when we use the implementation of the creating object we are only using it so that the created object has the same implementation. The algorithm illustrates one way to choose implementations when those implementations may depend on other choices of implementation that need to be made.

The goal of the algorithm is to determine a set of characteristics for objects that on the profiling run would have resulted in objects having the implementation determined by the partitioning in the previous sections. If this goal is not achievable then we seek the characterization which gives as good a set of implementations as possible.

Our algorithm for this combinatorial optimization problem is a simple greedy one, starting with a partitioning of the objects based on their optimal implementation in the profile run, as described in the previous sections. As we determine that certain characteristics should determine an objects implementation we will move those objects that have the same characteristics to the same partition of the OAG. That may be less than the optimal determined by the partitioning algorithm, but we may have no choice.

I. We measure the costs over and above the previous partition of the strategy of assigning all objects allocated at one site to the same fixed implementation and letting the remaining objects stay where they were in the earlier partition.

II. We measure the costs for all objects created at a given statement being given the same implementation as their creating object. We do this either by moving the creating object to the partition of the created object or by moving the created objects to the creating objects partition(whichever is cheapest). Leaving all other objects in their previous partition.

III. We find the allocation site and strategy with the minimal cost computed above, move all objects created at that site (or their creating objects) to the partition determined by that strategy and use this as a new partition and repeat until all allocation sites have a strategy.

This algorithm seems to us to be overly simplistic. However, we have run it and where the information was available to it, it has always yielded an optimal solution. The major problem seems not to be the algorithm, but rather the problem of finding the information needed. Our experiments, so far, have ignored the characterization of objects and it may well be that when we add that information in, there will be sufficient information but the algorithm will fail to find it. If and when that happens we will work on constructing a better algorithm.

6 Generalizations and Other Applications

Our framework can accommodate a number of other problems using two extensions. 1)In the previous sections we discussed choosing one property of an object, namely, its implementation. This property is intrinsic to a class of the object. We can extend our framework to include properties that are external to a class. Thus objects of many classes can have the same property. Such extrinsic properties, for example, can be the location (i.e., machine) where an object is created. 2) In the previous sections we also discussed one kind of interaction, namely, cloning of an object to create a new object. We can once again extend our framework to include interactions such as communication among objects located on different machines.

In our model the cost of interaction of a pair objects which have the same property is lower than the cost of interaction of a pair of objects that have two different properties. The same solution techniques can find the optimal choice of a property of objects.

6.1 A Generalization for Distributed Programming

Consider the location of an object in a distributed system to be its property and communication between the objects as the interaction of interest. The location property is extrinsic to its class. Two objects located on the same machine have lower communication cost. Every time one object invokes a method on another they have an interaction. Each method invocation has a cost if the objects have different properties and when the code for the components is written the CoercionCost must reflect that.

6.2 Enterprise JavaBeans

The Enterprise JavaBeans (EJB) specification defines an API for creating and developing enterprise applications. There are two kinds of beans, called the session bean and entity bean. The EJB

Specification Version 2.0 requires that a bean developer explicitly identify beans that are local and beans that a remote (from the perspective of another bean). A bean developer essentially hardcodes the affinity relation. Typically a bean developer does not know the environment in which a bean will be deployed. Using OAG model we can provide a better and a more flexible approach for distributing beans across multiple machine. For this problem, terminal nodes correspond to locations of JVMs and non-terminal nodes corresponds to different beans. A bean deployer (rather than the bean developer) can enforce that certain beans should be deployed on certain JVMs. Using our analysis framework we can then deploy unconstrained beans on the most appropriate JVM. Edge weights between non-terminal nodes (i.e., between beans) correspond to migration/communication cost. If two beans that interact are on the same machine then the migration/communication cost is zero, otherwise we compute this cost using system parameters, such as latency, bandwidth, marshaling and unmarshaling cost , etc.

6.3 XML Document Processing

The eXtensible Markup Language (XML) is a universal format for representing structured documents and data. XML was designed to describe data, and to focus on what data is. XML uses DTD (Document Type Definition) or XML Schema to describe structured document and data. XSLT (XML Style Language Transformation) can be used to transform XML that conform one DTD to another XML that conform to another DTD. We can use OAG model to automatically choose which DTD is the right representation for an XML document. Terminal nodes correspond to different DTDs and XML documents correspond to non-terminal nodes. An edge exists between to non-terminal nodes whenever there is a need to transform the format (i.e., DTD) of the XML document. The cost would be zero if both non-terminal node use the same DTD otherwise we need to apply XSLT to transform the document format.

6.4 Other Applications

Register allocation for multiple register files is another application where terminal nodes correspond to multiple register files and non-terminal nodes correspond to program variables. Transformation cost consists of cost of moving values between different register files. Character format transformation (such as from ASCII to EBCDIC, etc.) is another application of OAG.

7 Experimentation and Empirical Results

We were drawn to this problem by our colleagues who worked on an IBM product the VisualAge Generator (VAG). This is a 4GL language used to coordinate processes running on multiple machines. The dominant performance issue is related to distribution, as are many of the problems that arise in middleware/enterprise applications. The problem for VAG was relatively simple, all the "objects" are quite coarse grained and are known at compile time, so the issues of characterization don't apply. Our first set of algorithms were based on Ford-Folkerson derivative algorithms and were quartic in the size of the graph. We were quite successful in this context. We found that we could cut the graphs that for sample applications that came with VAG and our cuts had roughly half the weight of suggested one in the sample. Our elapsed time was 0.206ms per message with the whereas with the implementation suggested by the creators of the sample the cost per message was 0.205. Presumably the difference (assuming that it is not a measurement error) is because some of the costs that is related to things other than distribution. In other words, our partition resulted in a program almost twice as efficient. We have encountered examples in other IBM enterprise applications where a wrong partition can resulted in a factor of 40x performance degradation

Our assumption when we wanted to scale up to real object oriented programs was that the partitioning algorithm was going to be the bottleneck. We ran several Java programs, using Jinsight to trace the program and report which objects were created and what messages were sent between them. The heuristics for reducing the graph, which we have described here seem to largely solve the partitioning problem. Though we have not been able to find and instrument enough good benchmarks for us to conclude we have a representative sample (we've done less than a dozen examples). We have also only experimented with alternate implementations which are related to which machines an object resides on. We were able to optimally partition the samples from the VAG systems, while our previous more expensive algorithm was not. We have been able to take some graphs resulting from SPECJBB which was the best benchmark we could find and perform an optimal reduction of a graph that had 17,000 nodes and over 200,000 messages between objects in the program. SpECJBB simulates a retail chain with a company, several warehouses and several stores. We tried various combinations of requiring that orders stay on stores, other things staying on the computers that hold a warehouses information etc. We also tried to partition several other programs. For all the graphs that we generated our graph partitioning algorithm always generated optimal solution. The general pattern seems to be that dominant edge heuristic collapses between 50% to 90% of the nodes. Then zeroing heuristic reduces the weight of edges adjacent to the terminals followed by the terminal cut heuristic that that combine of some very heavy nodes. Once those nodes are combined, there are more dominant edges and the process repeats. Sometimes the graph is reduced to a relatively few nodes, say 30, and some of the more sophisticated and costly heuristics can be invoked to compute optimal solutions.

However, since we were only trying to model distribution in our experiments, it may be that other implementation choices will result in quite different graphs. The major cause for worry is that there may in other cases be many more edges to the terminals. We have also found some algorithms in the literature, see [Schloegel 2000] which are used for partitioning meshes in scientific/engineering problems, that are usually have weight no more than 30% greater than the optimal partition. These algorithms have been applied in a wider domain than ours have been and seem to do an adequate job.

Our conclusion is that while we were originally quite worried about the combinatorics of graph cutting this is unlikely to be a major roadblock to achieving the vision of this paper.

Similarly we were initially concerned that the cost of an optimal characterization algorithm might be prohibitively expensive. Hand analysis of the greedy algorithm on the examples we've made show that when the characterization suffices the algorithm finds the optimal characteristics to use easily. That is to say once we have been given characteristics to look for, the algorithm seems capable of finding the optimal combination -- *if that combination exists.* But, using only the characteristics of allocation site and the implementation of the

creating object may not be enough. There are at least two issues we encountered. 1) Java has class objects and SpecJBB uses factories. This means that all objects of one class are created at the same allocation point even if they seem to have different new statements. All the new statements call the class object which does the allocation. We needed to replicate the class objects and we also replicated factories (which were side effect free) then ran our algorithms. In all but one case we could see that the characterization had in fact resulted in an optimally partitioned program. However 2) we saw a problem, which has not been adequately studied yet in one of the SpecJBB sample runs where we had multiple warehouses connected to multiple stores. Part of the problem may be that we did not use all the characteristics discussed above in our experiment and part of the problem may arise because in that case because of some technical difficulties we lost some of the creation information. But we are concerned that since each warehouse uses the same code the allocation site will not let one distinguish between which warehouse an object should reside on. Since we'd lost the information the location of the object causing the allocation was lost and so neither characterization was available to distinguish. We do believe in looking at this example that we will certainly want to be able to use additional characterizations such as the location of the arguments to the new statement.

8 Related Work

We have described a method for extensible, high-level optimization, using partitioning and characterization. We discuss related work on each of these areas.

High level optimization:

Languages like SETL [Sch75a, Sch75b] and VERS [Ear76] also propose alternative implementations for basic data types such as sets. SETL uses type-based static analysis to select the best implementation for a given instance of the set representation. Our solution was to model the problem as a graph problem and use dynamic profile information to compute edge weights and to characterize properties that are distinguishable from one run to another run. Both SETL and VERS rely purely on static information which probably will give them worse performance. Incidentally there is no actual implementation in SETL which allows a client to automatically select a particular implementation (although a methodology is described in [Cai Et Al]).

Extensible Optimization:

Hisgen [Hisgen 85] proposes languages mechanism using conditional statements that are evaluated at run-time. Implementation choices are made based on the value of the conditions. Vandevoorde and Guttag extend this work by using a specification language (rather than transformation rules via conditional statements) [Vandevoorde 94].

Distribution and partitioning:

We know of no work that looks at the automatic minimization of the transfer costs between data representations using dynamic information, except in the context of distributed parallel programs. That related work falls, loosely, into two general areas: work on partitioning applications in the domain of parallel computation usually for scientific/engineering applications, and work on partitioning applications in the domain of distributed systems.

Researchers in parallel and distributed computing have addressed a related problem which consists of how to optimally partition code/data so as to minimize communication costs. Although these problems can be modeled as graph partitioning problem they are fundamentally different in that these graphs do not have terminal nodes. Graph cutting without terminals reduces to the "k-Cut" problem — partitioning the graph into a given number, k, of components such that no component is empty and the communication between them is minimized. The k-Cut problem is in fact solvable in polynomial time for each fixed $k > 2$, and hence is a significantly easier problem than the k-terminal cut problem. Partitioning distributed object applications, on the other hand, with the requirement that certain objects be placed on particular machines of a heterogeneous collection of different machines, represented in the graph by terminal nodes, which must not be combined, reduces to the problem of multi-terminal graph cutting, or the "k-terminal cut" problem, which is MAX SNP-hard [Dahlhaus et al. 94]. Stone[Stone 77] was the first to observe in the context of distribution that minimal transformation cost is obtained when we have a min-cut of the graph.

In the distributed systems domain, the problem of application partitioning has been widely studied for non-object oriented programs. The number of objects in a distributed object application is orders of magnitude larger than the number of modules to be distributed in a classical distributed application, and the methods typically employed for traditional systems do not scale well to problems of this size.

Both systems use algorithms based on Kernighan-Lin algorithms [Kernighan and Lin 70]. The best current variation of Kernighan-Lin for the number of objects we create is [Schloegel 2000] which builds a hierarchy of graphs with nodes in a higher graph corresponding to a pair of nodes in the lower graph. Usually this is done by choosing edges to collapse based probabilistically on their weight. At each level of the hierarchy you try to exchange nodes to improve the cut. In our experiments Schloegel was rarely more than 30% off optimal. Our algorithms are of comparable complexity and achieve optimality. The capacity constraints and some of the added flexibility of those other algorithms may become important in our context and while algorithms like Schloegel are not as good as ours for what we were measuring, they seem to have acceptable performance and somewhat more flexibility. One of our contributions is to find that on our samples, Schloegel was not too far from optimal, which can't be said, unless you can prove you've achieved optimality.

Characterization

Characterization of objects based on what is known at allocate time to determine what behavior they match from a previous run is close to virgin territory in the academic literature The closest match we know of is in the context of garbage collection where certain allocation sites have been found to produce data that should be more frequently collected than others. These allocation sites cause objects that are characterized as prolific. Using different strategies for collecting prolific objects makes sense much like using different strategies for different generations of objects. [Shuf 2002].

9 Conclusion and Future Work

Optimizing compilers have focused on optimizing the abstraction layer between machine language and the programming language. This paper in contrast tries to allow the specification of optimizations at all levels of abstraction that the program is

written in. We have argued that this requires extensible optimizations based on alternate implementations. Traditional optimization from the seminal work of [Allen and Cock 72] to [Learner 2002], which really does a nice job of elucidating and generalizing this model work by having for each program statement a sequence of implementations and uses static analysis to choose the most specific implementation that is legal. In contrast, we don't have a fixed sequence, some implementations are more efficient in some contexts than others.

To solve this problem we describe a contract between the component writer and the system. The component writer offers alternate implementations and information about their costs. The system chooses which implementation to use based on profiles of past behavior. We model the profiles as a graph and then show how to solve a combinatorial problem to determine the optimal implementations for the past run. Then we show how to generalize to future runs based on characteristics of objects that should use different implementations.

Autonomic solutions are important when the complexity of a system has exceeded human capacity to deal with it as a whole. Here we preserve the encapsulation that is so important to humans and yet allow the system to tune itself across component barriers. We have shown that the methodology suggested works as a first attempt to solve this problem.

The same contract that allows the system to look at the profiles may well be able to modify an object while the program is running. We give coercion methods that could change the implementation of an object. We could sample the costs and change the implementation. The advantages of the system doing this are many fold. It is difficult to write instrumentation code that takes samples. If one object should change implementation that suggests the program has entered a new phase and all objects should require less evidence that they should change implementations. The perceptive reader will observe that there is a close relationship between dynamically changing implementations and automatic software caching as well as object migration.

Static analysis can play a significant part here as well. In our experiment section we described about the importance of replicating class objects. Generalizing any object which becomes immutable after some point can be replicated. Once replicated multiple implementations could be used, with the appropriate one used for least cost.

We indicated that work needs to be done to improve the characterization. Moreover there are cases where the graph model is inaccurate, such as when coercion costs vary depending on which implementations are being coerced from and to. We have a more accurate model but have not yet tried to determine the heuristics to optimize it.

Finally more experience needs to be gained with a wider variety of implementations. More work also needs to be done on how to specify the implementations.

While we've indicated that lots of work remains, we hope we've convinced the reader that large improvements are possible and that the future work in this direction has a chance of setting a part of the agenda for much of the future of optimization that programmers face in the world of components.

References

[Allen and Cocke 72] F.E. Allen and J. Cocke "A Catalogue of Optimizing Transformations", in R. Rusin, ed., Design and Optimization of Compilers, Prentice Hall, 1972.

[Aho et al. 72] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1975.

[Bellisard et al. 96] L. Bellisard, S.B. Atallah, F. Boyer and M. Riveill "Distributed Application Configuration", Proc. 16th International Conference on Distributed Computing Systems, May 1996, pp. 579-585.

[Calinescu et al. 98] G. Calinescu, H. Karloff, and Y. Rabani "An Improved Approximation Algorithm for Multiway Cut", Proc. 30th STOC, 1998, pp. 48-52.

[Chern et al. 89] M. Chern, G.H. Chen and P Liu "An LC Branch-and-Bound Algorithm for the Module Assignment Problem", Information Processing Letters 32, 1989, pp. 61-71.

[Cheruki 96] C.S. Cheruki, Andrew Goldberg, D.R. Karger, M.S. Levin, and C. Stein, Experimental Study of Minimum Cut Algorithms, MIT Research report.

[Chopra and Rao 91] S. Chopra and M.R. Rao "On the Multiway Cut Polyhedron", Networks 21, 1991, pp. 51-89.

[Dahlhaus et al. 94] E. Dahlhaus, D.S. Johnson, C.H. Papadimitriou, P.D. Seymour, and M. Yannakakis "The Complexity of Multiterminal Cuts", SIAM Journal on Computing 23, 1994, pp. 864-894. Preliminary version appeared in STOC '92.

[Darbha and Agrawal 95] S. Darbha and D.P. Agrawal "A Fast and Scalable Scheduling Algorithm for Distributed Memory Systems", Proc. 7th IEEE Symposium on Parallel and Distributed Processing, October 1995, pp. 60-63.

[Demeure and Nutt 94] I.M. Demeure and G.J. Nutt "Prototyping and Simulating Parallel, Distributed Computations", Journal of Parallel and Distributed Computing 23(1), October 1994, pp. 1-9.

[DePauw and Sevitsky 99] W. De Pauw and G. Sevitsky "Visualizing Reference Patterns for Solving Memory Leaks in Java", Proc. 13th European Conference on Object-oriented programming, Lisbon, Portugal, June 1999, LNCS Vol. 1628, Springer, pp. 116-134.

[Ear76] J. Earley. High level iterators and a method for automatically designing data structure representation. J. of Computer Languages, 1976.[Fox et al. 94] G.C. Fox, R.D. Williams and P.C. Messina "Parallel Computing Works!", Morgan Kaufman Publishers, Inc., 1994.

[Gilbert et al. 95] J.R. Gilbert, G.L. Miller and S.H. Teng "Geometric Mesh Partitioning: Implementation and Experiments", 1995.

[Hisgen 85] A. Hisgen "Optimization of User-Defined Abstract Data Types: A Program Transformation Approach" Ph.D. Thesis CMU 1985 see: http://reports-archive.adm.cs.cmu.edu/anon/1985/abstracts/85-166.html

[Hunt and Scott 99] G.C. Hunt and M.L. Scott "The Coign Automatic Distributed Partitioning System", Proc. 3rd OSDI Symposium on Operating System Design and Implementation, February 1999, pp. 187-200.

[IBM 1996] IBM Corporation "Introducing VisualAge Generator Version 2.2", IBM Order No. GH23-0225-00, 1996.

[Junger et al. 00] M. Junger, G. Rinaldi and S. Thienel "Practical Performance of Efficient Minimum Cut Algorithms", Algorithmica 26, 2000, pp. 172-195.

[Karger and Stein 93] D.R. Karger and C. Stein, An $O(n^2)$ Algorithm for Minimum Cuts. In Proc. 25th Annual ACM Symposium on Theory of Computing, pages 757-765, 1993.

[Kernighan and Lin 70] B.W. Kernighan and S. Lin "An Efficient Heuristic Procedure for Partitioning Graphs", The Bell System Technical Journal, February 1970, pp. 291-307.

[Kimelman] Kimelman, T. Roth, H. Lindsey and S. Thomas "A Tool for Partitioning Distributed Object Applications Based On Communication Dynamics and Visual Feedback", COOTS '97 Conference on Object-Oriented Technologies and Systems - Advanced Topics Workshop, June, 1997.

[Kumar et al. 94] V. Kumar, A. Grama, A. Gupta and G. Karypis "Introduction to Parallel Computing: Design and Analysis of Algorithms", Benjamin/Cummings Publishing Company, Redwood City, CA 1994.

[Lerner 2002] Lerner S, Grove D. Chambers C, "Composing Dataflow Analyses and Transformation" POPL, Jan 2002

[Lo 88] V. Lo "Heuristic Algorithms for Task Assignment in Distributed Systems", IEEE Transactions on Computers 37(11), November 1988, pp. 1384-1397.

[Motwani and Raghavan 95] R. Motwani and P. Raghavan "Randomized Algorithms", Cambridge University Press, 1995.

[Padberg and Rinaldi 90] M. Padberg and G. Rinaldi "An Efficient Algorithm for the Minimum Capacity Cut Problem", Math. Prog. 47, 1990, pp. 19-36.

[Purao 98] S. Purao, H. Jain and D. Nazareth "Effective Distribution of Object-Oriented Applications", Communications of the ACM 41(8), August 1998, pp. 100-108.

[Ramanujan et al. 96] R.S. Ramanujan, J.C. Bonney, K.J. Thurber, R. Jha and H.J. Siegel "A Framework for Automated Software Partitioning and Mapping for Distributed Processors", Proc. 2nd International Symposium on Parallel Architectures, Algorithms, and Networks, June 1996, pp. 138-145.

[Schloegel 2000] K. Schloegel, G. Karypis and V. Kumar "Graph Partitioning for High Performance Scientific Simulations", to appear in J. Dongarra, I. Foster, G. Fox, K. Kennedy and A. White, eds., CRPC Parallel Computing Handbook, Morgan Kaufmann 2000, http://www-users.cs.umn.edu/~karypis/publications/Papers/PDF/gpchapter.pdf .

[Sch75a] Schwartz J, "Automatic Data Structure Choice in a Language of Very High Level" CACM December 1975, pp 722-728.

[Sch75b] Schwartz J, "Optimization of Very High Level Languages, Parts I, II" J. Of Comp. Lang, 1975 pp. 161-218.

[Shuf 2002] Shuf Y., Gupta M., Bordaweker R., Singh J.P., "Exploiting Prolific Types for Memory Management and Optimization" POPL 2002.

[SPEC 2000] SPECjbb2000, A Java Business Benchmark, http://www.spec.org/osg/jbb2000/docs/whitepaper.html .

[Stone 77] H.S. Stone "Multiprocessor Scheduling with the Aid of Network Flow Algorithms", IEEE Transactions on Software Engineering SE-3(1), January 1977, pp. 85-93.

[Stoyenko 96] A.D. Stoyenko, J. Bosch, M. Askit, and T.J. Marlowe, "Load Balanced Mapping of Distributed Objects to Minimize Network Communication", Journal of Parallel and Distributed Computing 34, 117-136, 1996.

[Tom and Murthy 97] A. Tom and C.S.R. Murthy "An Improved Algorithm for Module Allocation in Distributed Computing Systems", Journal of Parallel and Distributed Computing 42, 1997, pp. 82-90.

[Vandevoorde 94] Vandevoorde M, Guttag J , "Using Specialized Procedures and Specification-Based Analysis to Reduce the Runtime Costs of Modularity", SIGSOFT 94 Dec 1994 pp 121-127.[Welch et al. 95] L.R. Welch, B. Ravindran, J. Henriques and D.K. Hammer "Metrics and Techniques for Automatic Partitioning and Assignment of Object Based Concurrent Programs", Proc. 7th IEEE Symposium on Parallel and Distributed Processing, October 1995, pp. 440-447.

[Zhou 93] H.B. Zhou "Two-stage M-way Graph Partitioning", Parallel Computing 19, 1993, pp. 1359-1373.

Appendix A: Proofs of Lemmas

Lemma 1. Dominant Edge. Suppose graph **G** has a non-terminal node **n**, with edge **e** being the heaviest among the edges adjacent to **n**, with edges $e_1, e_2, \ldots, e_r$ being the edges connecting **n** to non-terminals, and with edges $e'_1, e'_2, \ldots, e'_k$ being the edges connecting **n** to nodes which are terminals. Further suppose that $w(\mathbf{e}) \geq \Sigma\, w(e_i) + Max(\, w(e'_1), w(e'_2), \ldots, w(e'_k))$. Then there is a minimum cut not containing edge **e**. That is, we can contract edge **e** to obtain the new graph **G′** that has a minimum cut with the same value as a minimum cut of **G**.

Proof: Consider an optimal assignment of nodes (corresponding to a minimum cost cut). Assign all the nodes other than **n** to their partition in that optimal assignment and contract them with the terminal in that partition. At this point, we have the node **n** connected only to terminals. It can only be contracted with one, and hence the optimal partition would have it contracted with the terminal with which it shares the heaviest edge.

Each member of the set of the edges to the terminal $\mathbf{E}'' = \{e''_1, e''_2, \ldots, e''_m\}$, is created by combining none or one of the edges $e'_1, e'_2, \ldots, e'_k$ from **n** to terminals (and **e** if other end of **e** is a terminal node) with a subset consisting of zero or more of the edges $e_1, e_2, \ldots, e_r$ from **n** to other nodes (and **e** if other end of **e** is not a machine node). Optimal assignment of **n** is obtained by contracting the largest edge in the set **E″**.

The weight of the edge in the set **E″** containing **e** as a part of it is greater than or equal to $w(\mathbf{e})$. The upper bound on the weight of any edge in **E″** not containing **e** is less than or equal to $\Sigma\, w(e_i) + Max(\, w(e'_1), w(e'_2), \ldots, w(e'_k))$. (This upper bound is obtained when the largest of the edges $e'_1, e'_2, \ldots, e'_k$ is combined with all of the edges $e_1, e_2, \ldots, e_r$.) Therefore, if the conditions of the lemma are satisfied, then the largest edge in the set **E″** would be the one containing **e** . And this would be the edge that is contracted. Hence **e** is contracted in the optimal assignment of nodes to machines. **QED**

Lemma 2. Independent Net. If the communication graph between objects can be broken into two or more independent nets, then the min cut of the graph can be obtained by combining the min-cut of each independent net.

Proof. A net is a set of connected nodes. We say two nets $\mathbf{Net}_1$ and $\mathbf{Net}_2$ are independent if there is no edge between non-terminal nodes in $\mathbf{Net}_1$ and non-terminal nodes in $\mathbf{Net}_2$. Let $\mathbf{Net}_1$, $\mathbf{Net}_2$, ... $\mathbf{Net}_n$ be the independent nets in the graph. The weight of any cut of the graph is the sum of the weights of the cuts in the nets, since there are no edges between the nets. Hence minimizing the weight of the cuts in each of the nets will minimize the weight of the overall cut. **QED**

Lemma 3. Terminal Cut. Let a terminal cut $\mathbf{T}_i$ be the set of all edges between a terminal $\mathbf{t}_i$ and non-terminal nodes **N.** Let $\mathbf{W}_i$ be the sum of the weight of all edges in the terminal cut $\mathbf{T}_i$. Let $\mathbf{W}_i$'s be sorted so that $\mathbf{W}_1 \geq \mathbf{W}_2 \geq \mathbf{W}_3 \geq \ldots$. Then any edge which has weight greater than $\mathbf{W}_2$ cannot be present in the minimum cut, and hence can be contracted.

Proof: We will prove this lemma by contradiction. Assume that an edge **e** with the weight $w(\mathbf{e}) > \mathbf{W}_2$ is present in the optimal cut between terminal node $\mathbf{t}_i$ and terminal node $\mathbf{t}_j$. Let $\mathbf{W}_j \leq \mathbf{W}_i$. We know that $\mathbf{W}_j \leq \mathbf{W}_2$ since $\mathbf{W}_2$ is the second largest of the set $\{\mathbf{W}_1, \ldots \mathbf{W}_t\}$.

Now consider what happens if all nodes that are in the same optimal partition with terminal $\mathbf{t}_j$ are moved to terminal $\mathbf{t}_i$ The weight of the cut is reduced by at least $w(\mathbf{e}) - \mathbf{W}_j$. We know that $w(\mathbf{e}) - \mathbf{W}_j \geq w(\mathbf{e}) - \mathbf{W}_2$. Since this is a positive number it contradicts our assumption that the earlier cut was a min-cut. Therefore the edge **e** cannot be a part of a minimum cut. **QED**

Lemma 4 Zeroing. Assume that a non-terminal node **n** has edges to each of the **t** terminals in **T** with weights $\mathbf{w}_1 \leq \mathbf{w}_2 \leq \ldots \leq \mathbf{w}_t$. **n** may also have edges to other nodes in the graph. Reducing the weights of each of the t edges from **n** to the terminals **T** by $\mathbf{w}_1$ does not change the assignment of nodes for minimum cut. It reduces the cost of the minimum cut by $(t\text{-}1)\ \mathbf{w}_1$.

Proof: Zeroing reduces the weight of each terminal cut, thus permitting more edges to be contracted using the Terminal Cut heuristic based on Lemma 3.

Consider the graph after all nodes but **n** have been assigned to terminals. Now consider what happens if each of the t edges connecting **n** to the various terminals **T** is reduced in weight by $\mathbf{w}_1$. The effect on the weight of a particular assignment of **n** to a terminal is to reduce the weight of the cut by $\mathbf{w}_1(t\text{-}1)$, since the edges cut are the same, and t-1 cut edges have their weights reduced by $\mathbf{w}_1$ each. It reduces the weight of cut by $(t\text{-}1)\ \mathbf{w}_1$.

The edge that was the heaviest of the last t edges before reducing the weights stays the heaviest after the reduction. The assignment associated with the optimal cut has therefore not changed, since contracting the heaviest edge always preserves optimality. **QED**

Lemma 5. Articulation Point. Let **S** be a subset of the non-terminal nodes of **N** that would be disconnected from all of the terminals if non-terminal node **n** were deleted. Then none of the edges between the nodes in **S**, nor the edges between the nodes in **S** and **n** can be in the minimum cut and they can all be contracted.

Proof: If all of the nodes in **S** are assigned to the same terminal as **n**, there would be no contribution to the cut from the edges in **S**. If any node **s** in the set **S** is assigned to a different terminal, then at least one edge in the path connecting **s** to **n** has to be cut, and will therefore make a positive contribution to the cut. Hence, the minimum cut would involve the nodes in **S** assigned to the same terminal as **n,** and hence all the edges in the subgraph defined by **S** and node **n** can be contracted. **QED**

Appendix B: More on Experimental Results

In this section we describe two sets of preliminary experiments we did to test the applicability of our framework. In the first experiment we ran our tool on small applications developed using VisualAge Generator. In the second experiment we used SPECjbb.

B1. VisualAge Generator Experiments

IBM VisualAge® Generator is a powerful high-end, rapid application development environment for building and deploying e-business applications. Developers with little or no Java expertise can implement end-to-end Java e-business systems. We used test suite from the material IBM uses to teach customers how to build applications from templates provided with this fourth generation language. These test suite is a representative in structure (if not scale) of applications typically developed by customers in the field. A 3-tier version of the VisualAge Generator Templates application was generated, compiled, and run on a small test bed. topology consisting of 16 Mb/s token-ring network with three modest Windows NT 4.0 workstations, serving as a client machine, a middle-tier logic server and a database server. The database was DB2 2.1.2. Transaction completion times were measured in milliseconds using the NT _ftime() function.[2]

A representative scenario consisting of a sequence of short transactions was run. For the purpose of this study, three different partitions were configured, run, and measured in order to compare their actual performance: A "naive" partitioning, in which all logic components were placed on the middle tier, GUI components on the client and database components on the server; a manual partitioning in which some components were moved from the middle tier to the client or server based on programmers intuition and automatic partitioning in which our tool decided where each component went on the basis of the measured communication profile.

The results are shown in the following table.

Partitioning	Cut Cost (messages Between machines)	Run Time (ms)	Run Time/Cut Cost (ms/message)
Naive	53	10.23	0.193
Manual	42	8.62	0.205
Automatic	23	4.75	0.206

We can see from the table that the automatic partitioning significantly improves the performance of the program and that the cut cost (the number of messages sent between machines) is almost exactly proportional to the running time of the program. If the computational cost were a significant factor, then we would have seen significant effects of moving the components from faster server machines to slower client machines as we went from Naive to Manual to Automatic partitioning. Instead we see from the data in this case that performance is primarily determined by communication costs and we believe that for a significant class of programs, communication costs is the dominant factor in system performance.

B2. SPECjbb Experiments

In a second experiment we ran SPECjbb2000 [SPEC 2000], a Java benchmark derived from TPC-C, an industry standard transaction processing benchmark that models a supply chain with warehouses, districts, etc. We have also ran experiments against its predecessor, pBOB. The two Java benchmarks have been run with numbers of objects ranging from 2,000 to 11,000.

2

At the time we were running these experiments we were attempting to decrease communication costs between machines. So, our constrained entities correspond to objects that must reside on certain machines. Our unconstrained entities are objects that could exist on any machine, but we did not presuppose mobile objects that could move themselves as the program entered a new phase (see section 7). Typically, our improved dominant edge heuristic alone will reduce the graph to between one half and one third of its original size. In some cases, it reduces the graph to as little as one tenth of its original size. At that point, the zeroing heuristic will reduce the weights of edges adjacent to terminals, and then the most complex nodes can be contracted with the terminal cut heuristic. Dominant edge will then find opportunities for additional reduction. Ultimately, in most cases, the heuristics all working in concert are able to reduce the graph totally. In those few very complex cases that remain, the graph has been so reduced that it is amenable to either branch and bound backtracking or simple approximation algorithms that often produce the optimal cut.

We ran the Java programs on a single machine in a single JVM. We monitored the communication between the objects using an instrumented JVM in Jinsight [Wim et al] that produces traces of Java runs consisting of a list of objects allocated by the JVM and a list of calls from one object to another. Since this instrumented JVM is two orders of magnitude slower than regular JVM, and produces large trace files, we traced the transactions of SPECjbb2000 for only a short time. This includes hundreds of transactions and results in hundreds of thousands of messages between thousands of objects.

From this trace we inferred the effect of running this program on multiple machines. We had to make several assumptions to make this inference. Since each machine will have its own JVM and class files, we assumed that all class method calls to be local. Thus if an object A calls any class method which creates or calls object B, we counted this as a message from object A to object B. We constructed a graph with the objects instances as the nodes, and message count between them as the weight of the edges between them.

In one SPECjbb2000 experiment, we chose the four transaction managers, which are surrogates for users initiating transactions, as terminals, and one warehouse, which are objects that keep track of the items that would be stored in a physical warehouse. Our partitioning places roughly uniformly distributes most objects on each of the transaction managers, and the warehouse.

The following table compares our work with the algorithm described in [Schloegel 2000] (though it is our own implementation of that algorithm) to our own in a variety of these runs. We give our results -- the optimal ones and results we would get without using the Dalhouse heuristic (which in these graphs only collapses one edge) because we are concerned that there may be some scaling problems with that heuristic.

data	Spec1	Spec2	Spec3	Spec4
Number of entities	1,972	3,317	6,197	11,478
Number of edges	2,844	4,896	9,444	17,878
Number of messages	29,323	53,954	109,503	210,889
Weight of optimal cut	1,418	2,611	5,288	10,901
Weight w/o Dalhouse	1,418	2,642	5,437	10,914
Weight Schloegel's algorithm gets	2,061	3,710	5,754	13,070