Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems

Dionisio de Niz and Raj Rajkumar
dionisio@cs.cmu.edu, raj@ece.cmu.edu
Real-Time and Multimedia Systems Laboratory
Electrical & Computer Engineering
Carnegie Mellon University

***Abstract*— Embedded real-time systems must satisfy not only logical functional requirements but also *para-functional* properties such as timeliness, Quality of Service (QoS) and reliability. We have developed a model-based tool called `Time Weaver` which enables the modeling of functional and para-functional behaviors of real-time systems. It also performs automated schedulability analysis, and generates glue code to integrate the final runtime executable for the system. Its extensive glue code generation capabilities include the ability to insert inter-processor communications code at arbitrary software boundaries. In other words, from a functional point of view, a software component may be viewed as a single logical entity but from the tool point of view, the component can be partitioned into two or more pieces running on different nodes. This capability opens up many different possibilities to map (partitioned) software components to hardware nodes. The objective of this deployment is to minimize hardware requirements while satisfying the timing constraints of the software. The classical approach to addressing this problem is to use bin-packing techniques. In this paper, we study *Partitioning Bin Packing*, an extension to bin-packing algorithms to exploit the capability of partitioning software modules into smaller pieces. We analytically show that the number of bins required can be reduced. We also evaluate a number of heuristics to minimize not only the number of processors (bins) needed but also the network bandwidth required by communicating software modules that are partitioned across different processors. We find that a significant reduction in the number of bins is possible. Finally, we show how different heuristics lead to different tradeoffs in processing *vs* network needs.**

I. Introduction

Embedded real-time systems are tightly coupled with the physical world. This tight coupling imposes para-functional requirements (such as timeliness, jitter, fault-tolerance, and security) that go beyond functional (logical) behaviors. We have developed a model-based embedded system development tool called `Time Weaver` [6], which provides new abstractions that enable the modeling of both functional and para-functional behaviors. Each functional behavior and para-functional behavior is captured in a separate view. Each view focuses on a single concern enabling a domain expert (e.g. a signal processing expert, a control system expert, a real-time systems expert, a fault-tolerance expert) to focus on the concern of her expertise. Interactions among the multiple views are automatically handled by the tool. For instance, a software-defined radio[1] model presents a functional view to the signal processing engineers to compose the mathematical transformation of signals in order to modulate/demodulate them. A second view models the Fault-Tolerance (FT) structure (called *Fault-Tolerance* view) and can be modified by the FT expert. In this second view, the FT expert defines which functional modules should be replicated to tolerate faults, how to synchronize their execution such that their internal state does not deviate from one another, and how to ensure failure independence (e.g., by deploying them on different processors). A third expert, the Real-Time (RT) Systems expert, can be working on a *Timing* view defining timing relationships among components (e.g. how the rate of execution of a component relates to that of another). Another important view in our tool is the deployment view, where the hardware is described and the deployment of software to hardware is modeled. The tool automatically projects the impact of changes in one view to the others. For instance, the addition of a new component in the functional view is propagated to all the other views (Timing, Fault-Tolerance, and Deployment), the replication of modules is propagated to the Deployment view to be able to deploy the new replicas, and the timing relationships among components are propagated to the deployment view to evaluate whether the available processors have enough cycles to execute the software within their timing constraints. This automatic synchronization across the multiple views gives these views strong semantics, and hence we refer to them as *semantic dimensions*. Once the multiple concerns of the system and their mechanisms are modeled in the different views of the tool, the final executable code of the system can be automatically generated, integrating the functional code with the code for the specified para-functional behaviors.

In order to be effective, model-based technology must take the design decisions down to the running code. One of the important decisions in the design of distributed real-time systems is the allocation of software modules to hardware nodes. Given the constraints and cost sensitivity of many embedded systems, this allocation must reduce hardware costs while satisfying all the functional and para-functional requirements of the application. The automatic assignment of software modules (or tasks) to hardware nodes is the focus of this paper.

[1] A software-defined radio is a radio that implements the modulation and demodulation of radio transmissions in software.

The traditional approach to allocate software modules to hardware nodes is to use a bin-packing scheme (or its cousin, load-balancing). For example, tools such as RapidRMA from Tri-Pacific Corporation and TimeWiz from TimeSys Corporation support various kinds of bin-packing and load-balancing techniques. These techniques are appealing because (a) they perform well in practice in reducing the number of bins required, and (b) they are computationally efficient. However, these techniques and tools generally treat each software module as an unbreakable entity with each being deployed as one piece. In this paper, we show that the flexibility to "partition" a software module across two or more processors can lead to a (sometimes significant) reduction in the number of processors needed. We provide both analytical and simulation results. Analytically, we show that the number of processors ("bins") can be reduced in the worst case. Our simulation approach shows that our heuristics work well in practice. For instance, we show cases where our approach needs only 201 processors when the classical approach requires as many as 244 processors.

A. Our Approach

In our allocation scheme, we model the software as a directed acyclic graph of modules that communicate through messages. Modules are characterized as consuming CPU cycles (periodically) and messages as consuming bandwidth (bits per second). A second graph is used to represent the hardware architecture. In this graph, the nodes are processors and networks, and the links represent connectivity among them. (We assume only simple hardware architecture graphs such as bus-based or switch-based networks.) Processors are considered to provide a processing capacity expressed in cycles per second and networks are considered to provide a communication capacity in bits per second. Given these two graphs, the allocation problem is posed as a packing problem that assigns modules to processors and messages to networks (when the communicating parties are not on the same processor). We also assume that the Earliest Deadline First (EDF) scheduling algorithm [10] is used. When an object is partitioned and deployed on two (or more) processors, any timing constraint that applies to the object must now span those processors. The choice of intermediate deadlines and related latency issues are outside the scope of this work. However, our approach can be extended in rather straightforward fashion to include classical real-time scheduling techniques to deal with these issues. When an object is partitioned across multiple processors, we assume that a communication bandwidth requirement may be imposed between the parts of the objects. However, we assume that no timing penalty is incurred in terms of execution time (i.e. object size) due to the partitioning. An "inflation factor" to account for this overhead can be added later if necessary.

The ideal objective of the packing problem is then to use the minimum number of processors and network links to deploy a software graph. We can think of this problem as the generic problem of packing a list of objects into the minimum number of bins. This generic problem is known as bin-packing. The optimal solution to this problem is known to be NP-complete [8] but multiple near-optimal algorithms have been studied over the years. However, these solutions assume that objects are independent of one another (e.g. software modules do not communicate) and hence have limited practical application. In this paper we present our extensions to bin-packing algorithms to allow partitioning of software graphs to minimize the number of processors needed to run the system while attempting to reduce the network links needed (due to messages across processors) as much as possible. A formal analysis of these algorithms proves that the worst-case behavior with partitioning improves over the worst-case behavior of previous solutions that did not allow partitioning. In particular, we show that in the worst-case we can save at least two bins. We conjecture that the worst-case behavior is improved significantly more, but leave it as an open problem. A new algorithm, derived from the formal properties is also designed. In addition we perform several experiments to evaluate their average-case performance and explore the "bad" cases. In these cases, we observed a difference against the non-partitioning scheme of over 20% (i.e. 20% of the processors were saved out of 244). Furthermore, our polynomial scheme is near-optimal in that it incurs only an average difference of just 4% over the optimal scheme which requires an exponential amount of computation. In summary, the partitioning scheme improves both the worst-case and average-case behavior of bin-packing, and is near-optimal despite its polynomial complexity.

B. Related Work

Multiple research efforts have been conducted relating to deployment of software modules to hardware. We discuss below the most representative among these studies. Additional research with useful results on bin-packing will be discussed in the context of our analysis in Sections II and III.

In [1], Abdelzaher et al. propose a period-based load partitioning scheme that minimizes a metric called the *system hazard*, which is defined as the maximum response time among all tasks. These tasks are real-time tasks with precedence constraints and dependent execution times. A branch-and-bound algorithm was developed to cluster hardware nodes and software modules with the objective of performing an hierarchical allocation that minimizes the system hazard. In contrast, our work does not minimize the system hazard but the number of processors needed for a specific software graph. Our work, however, does verify that enough CPU cycles are allocated to software modules to honor their deadlines (assuming an EDF scheduler).

In [11], Mutka and Li presented a tool to allocate real-time tasks to processors. Their tasks are modeled as independent tasks and a fixed-priority scheduler with RMS [9]. A branch-and-bound algorithm is used to find a feasible allocation. In our scheme, we allow components to have communication dependencies.

In [13], Tindell et al. developed a simulated annealing algorithm for real-time task allocation. Tasks communicate with each other using a token protocol and the task's deadlines are modified to take into account the delay of such communication. Audsley's [2] response time analysis was used to calculate the "energy" component of the algorithm. In our

case, we assume that the deadlines have already been adjusted for any delays due to dependencies and focus on the allocation of software to processors and messages to networks.

Related to bin-packing, an algorithm to allocate divisible objects to bins was developed in [4]. However, the size of the objects are restricted to be in a special divisible sequence, where the size of object o_1 is divisible by the size of object o_2, which in turn is divisible by the size of object o_3 and so on and so forth. Such a restriction makes this approach difficult to apply in practice. Our work does not have this restriction and allows components to be partitioned into *any* size.

In the networking arena, Naaman presents in [12] a bin-packing algorithm to fit packets into a TDMA network. These packets can be fragmented but the fragmentation adds an overhead factor representing the additional bits that need to be sent. The purpose of the algorithm is to pack network packets compactly, avoiding fragmentation as much as possible. Even though this work also deals with object partitioning, in our case, we formally analyze the improvements on the worst-case performance of bin-packing algorithms due to partitioning, assuming that such partition does not impose any overhead.

C. Organization of the Paper

The rest of the paper is organized as follows. Section II introduces the most popular bin-packing schemes and their performance characteristics. Section III presents our partitioning schemes, and analyses some of their properties. Section IV presents extensions to our schemes to reduce the network bandwidth requirement due to communicating modules deployed on different processors. Section V provides evaluation results of heuristics based on the analyses. Section VI presents the concluding remarks. Finally, in the appendix, we provide a brief description of our `Time Weaver` tool.

II. Bin-Packing Algorithms

Our deployment algorithms are based on a new paradigm that may "partition" software modules into two or more pieces. In practice, partitioning pieces will lead to communication code insertion at the partitioning points. We have built a model-based design tool for embedded real-time systems called `Time Weaver`. With `Time Weaver`, communications code required to connect a partitioned object can be automatically generated. Also, both functional and para-functional behaviors can be verified with the assistance of built-in or external analyses. Hence, object partitioning to reduce the number of processors ("bins") is a very desirable objective.

Our partitioning algorithms are derived from the traditional bin-packing algorithms. Bin-packing algorithms try to solve the problem of packing a set of objects into the minimum set of bins. In the basic version, the maximum size of each object is 1, and each bin is of size 1. A bin can be filled with objects until the total size of all the objects in the bin sum up to 1. Objects cannot be partitioned and must be allocated as whole. Previous research [8] has already shown that finding the minimum number of bins to pack a set of objects (namely the bin-packing problem) is NP-complete. However, multiple near-optimal algorithms of polynomial complexity have been designed. The most popular and best understood is Best Fit Decreasing (BFD). This algorithm orders the objects in decreasing order of size and the bins in increasing order of available space (gaps). BFD then tries the allocation of the first object in each of the bins in order. This scheme has two attributes. One, it assigns the largest objects first when it is more likely to find a larger gap. Two, BFD seeks to leave the minimum gap possible by assigning the largest object to the smallest gap that accommodates it. This second attribute is the basis of the less complex algorithm called Best Fit (BF).

Other less complex algorithms such as First Fit and First Fit Decreasing have also been used for bin-packing. First Fit (FF) does not order the bins or the objects, and assigns an object to the first bin where it fits. First Fit Decreasing (FFD) is an improvement on FF where the objects are ordered in decreasing order to reduce the gaps in the processors. We will not use these two algorithms because the worst-case performance of BFD is better than FF (as we will discuss). In the analysis of these algorithms, we will use XFD when the properties discussed apply to both BFD and FFD, and XF when they refer to either BF or FF.

Our framework is designed to support distributed real-time systems, where software modules on one processor may communicate with modules on other processors. If such communicating modules can be assigned to the same processor (as a composite one), their communication requirements across a network will be minimized or even eliminated. However, not all modules that communicate with each other may fit into the same processor. Furthermore, due to the (recursive) encapsulation of software components into composite ones, each component can became potentially large in size, to the extent that it may not fit into one processor. The bin-packing scheme will therefore fail for such large objects. We avoid these problems by allowing objects to be partitioned (or split) into smaller objects[2] down to individual elementary components if required. In this section, we will propose some extensions to bin-packing schemes to deal with partitioned objects, and will analyze their worst-case performance.

For these schemes we make the following assumptions. First, each bin is assumed to be equal to 1. Secondly, each object ranges in size from 0 to 1[3]. Finally, if an object is partitioned, the sum of its parts equals the size of the original object.

A. Performance of the Basic Bin-Packing Algorithms

Different authors have analyzed the worst-case performance of bin-packing algorithms. To define this performance, Johnson [7] uses $FF(L)$, $BF(L)$, $FFD(L)$, and $BFD(L)$ to denote the worst-case number of bins used by each of these algorithms to accommodate a list L of objects to be assigned. Similarly, $XFD(L)$ denotes the number of bins used by any of these algorithms. In addition, L^* is defined as the optimal (minimal) number of bins to accommodate the list L. The performance

[2]The compositing of components into larger ones is a software abstraction designed to enhance reuse and facilitate the construction of systems of systems.

[3]Larger composites are assumed to have been already partitioned to sizes less than 1.

of an algorithm is then defined as the ratio $\frac{\langle alg\rangle(L)}{L^*}$ where $\langle alg\rangle$ can be replaced by any of the algorithms. The maximum value that this ratio can achieve over all lists with $L^* = k$ is represented by $R_{\langle alg\rangle}(k)$. The worst-case performance is then defined as: $\lim_{k\to\infty} R_{\langle alg\rangle}(k)$[4]. These ratios are achieved for small values of k, and so their asymptotic results can be applied to all values of k. These ratios effectively define an upper bound on the worst-case performance of these algorithms.

Johnson et al. [8] proved that the bound for FF and BF is $\frac{17}{11}L^* + 2$ and Johnson [7] proved that the bound for FFD and BFD is $\frac{11}{9}L^* + 4$. Later, Baker [3] reduced the bound for FFD and BFD to $\frac{11}{9}L^* + 3$. Some of these proofs tend to be very long (some exceeding 75 pages). We will use selected theorems and lemmas from these proofs as a basis for analyzing our partitioning schemes.

III. BFD With Partitioning ($PBFD$)

When objects are partitioned, communication needs are likely to be introduced between components assigned to different processors. Therefore, in addition to the classical bin-packing objective of minimizing the number of processors, one can also seek to minimize the network bandwidth across processors. We refer to this multiple objective problem as a BFD packing problem with partitioning ($PBFD$). Mathematically, each software component can be considered to be a node in a graph G with an edge added between two nodes if the two nodes represented by these nodes communicate. Then, each connected graph component is treated as a composite object that can be partitioned and assigned by the bin-packing scheme.

In classical bin-packing, bins are of unit size. One starts with (zero bins or) 1 bin and additional empty bins are added when necessary. In our context, adding bins on demand can lead to unnecessary and premature partitioning of objects. For instance, consider a list of objects $\{w, x, y, z\}$ to be deployed with sizes $(0.6, 0.6, 0.2, 0.2)$ in that order. If we add bins on demand, we would start with a single bin b_1 and allocate object w of size 0.6. At this point, we would consider object x and decide that it does not fit in bin b_1 and, since we do not have another bin, we proceed to split the component, perhaps in halves x_1 and x_2 putting them back in the list ($\{x_1, x_2, y, z\}$). Now, the next object x_1 can be assigned to bin b_1. Assuming that no other object can be partitioned, we would need to allocate a new bin b_2 to continue with the deployment. This bin can now hold the rest of the objects in the list ($\{x_2, y, z\}$), leading to an assignment as follows: $b_1 = \{w, x_1\}, b_2 = \{x_2, y, z\}$. Now, if instead of allocating bins on demand, we start with two bins b_1 and b_2. Next, we can allocate w and x to each of these bins. Then, allocate objects y and z to bins b_1 and b_2 leading to the assignment: $b_1 = \{w, y\}, b_2 = \{x, z\}$. The resulting network bandwidth requirement is zero given that no object was partitioned. We therefore avoid the premature partitioning of objects by initializing the bin-packing scheme differently (but equivalently). Specifically, we pre-allocate bins such that the sum of the capacity of these bins is $\lceil load\rceil$, where $load$ is the sum of the sizes of all the objects. $\lceil load\rceil$ represents a lower bound on the number of bins required. One of two potential partitioning schemes will be used:

1) **Early Partitioning**. This scheme partitions an object immediately if it does not fit into any of the pre-allocated bins. The goal is to partition the largest objects as early as possible when there is a higher likelihood of finding larger gaps.
2) **Late Partitioning**. This scheme defers the partitioning of the objects that do not fit into any of the bins until we are done deploying all the objects that can be deployed without partitioning. The goal of this algorithm is to ensure that we deploy the largest number of objects without partitioning to try to minimize the penalty of the partition, i.e. communication between processors.

It must be noted that if we are able to accommodate all the objects in the pre-allocated bins, then our algorithms behave like vanilla BFD.

The $PBFD$ algorithm can be summarized as follows. First, we pre-allocate a number of bins equal to $\lceil load\rceil$ as discussed above. Secondly, the allocation order of BFD is followed. Specifically, we order the bins by non-decreasing order of gaps and try the deployment of composites in decreasing order of size. Thirdly, if partitioning is needed (as defined by the partitioning scheme in use) the selected composite is partitioned into two parts that can be fit into the largest gaps[5] available. If this partitioning is possible, these parts are *returned* to the list of objects to be deployed and the list is reordered (in descending order of object/piece sizes). In other words, the object is partitioned but instead of being deployed, the pieces are added to the list of objects to be deployed. If, on the other hand, the composite cannot be partitioned into parts that fit into any gap, then a new bin is added, and the *unpartitioned* composite is assigned to the new bin. Finally, the deployment continues until all objects have been assigned.

We now show that the partitioning schemes improve the worst-case performance of the bin-packing algorithms.

We first prove that the worst-case performance of bin-packing algorithms XFD can be improved when objects can be partitioned into two parts[6] whose sizes are below a threshold value. We then prove that even when the two parts are partitioned into equal halves, this improvement holds.

Remark: Some aspects of the worst-case analysis may be somewhat confusing at first. The theorems on bin-packing focus only on the *worst-case* behavior of the algorithms, and determine the sizes of the objects that play a key role in increasing the number of bins required for packing as compared to (say) the optimal algorithm. A similar behavior manifests with partitioning bin-packing. Objects must satisfy some constraints on size to increase the number of bins

[4]It turns out that a higher ratio can be achieved for lists that can be packed in two or three bins. However, this is deemed to be a special case that is not reproducible in larger lists where an automatic algorithm is likely needed. This observation long made by bin-packing studies (e.g. [8]) also leads us to focus on lists of a large number of objects.

[5]One just has to compare the parts with the largest gaps available.

[6]Partitioning into more pieces is just an extension to this assumption given that the resulting pieces can be partitioned later into pairs.

required. For arbitrary lists of objects whose sizes do not satisfy these constraints, the number of bins required would be better in relation to the optimal algorithm.

We consider a family of $PBFD$ algorithms where we start with an initial number of bins equal to $\lceil load \rceil$. We refer to these bins created at initialization time as the **primary bins**. New bins are added only when the next object to be allocated (even after partitioning it into two parts) does not fit into any available gap. We refer to these bins as the **supplementary bins**. To prove the worst-case behavior of our schemes, we first restate some lemmas, facts, and theorems from previous authors and develop some new ones.

Johnson et al. [8] proposed the following lemma to prove that small objects can be disregarded without modifying the worst-case behavior of bin-packing algorithms. We use the following notation as defined and used by Johnson to minimize ambiguity:

- L denotes the list of objects to be packed,
- L^* denotes the number of bins used by the optimal packing algorithm (theoretical) to pack list L,
- $L\backslash x$ denotes the deletion of object x from list L,
- P denotes a packing of a list of objects,
- P^* denotes the optimal packing of a list of objects,
- a_i denotes the size of object i,
- B_i denotes bin i, and
- B_{last} denotes the last bin of the packing as ordered by BFD.

  As stated earlier, $FF(L)$, $BF(L)$, $FFD(L)$, and $BFD(L)$ denote the worst-case number of bins used by each of these algorithms to accommodate a list L of objects to be assigned. Similarly, $XFD(L)$ denotes the number of bins used by any of these algorithms.

Lemma 3.1: Suppose L is a list such that $XFD(L) > rL^* + d$ with r,$d \geq 1$. The list $L' = L\backslash x | x \leq \frac{r-1}{r}$ also has $XFD(L') > rL'^* + d$.

Proof: Let P be the packing of L and P' the packing of L' using K and K' bins respectively. If $K > K'$, then no element in the last bin of P can be larger than $\frac{r-1}{r}$, and hence all but the last bin must have levels exceeding $\frac{1}{r}$, since neither BF nor FF will start a new bin with an element which would fit in a previous bin. Thus, $L^* \geq \sum a_i > \frac{1}{r}(K-1)$ and so $K < rL^* + 1$, contrary to hypothesis. Hence $K' \geq K$ and the lemma is proved. ∎

Corollary 3.2: Only lists L with objects of sizes in the range $(\frac{2}{11}, 1]$ need to be considered for the worst-case analysis of XFD.

Proof: This follows from Lemma 3.1 and by the bound of $XFD(L) = \frac{11}{9}L^* + 3$ proven by Baker [3]. ∎

Based on Corollary 3.2, Baker in [3] states the following fact:

Fact 3.3: In an XFD packing P of list L, if the size of the smallest object x in L is $\frac{1}{2} \geq x > \frac{1}{3}$, then pieces of size greater than $(1-x)$ that, in turn, are greater than $\frac{1}{2}$ are packed alone in bins in both P and P^*, while the remaining pieces, which are of size at most $\frac{1}{2}$, are paired in bins of both P and the optimal packing P^* (except for a possible odd piece); hence $XFD(L) = OPT(L)$. So the smallest object x of any list L that exhibits the worst-case performance must have a size $\frac{2}{11} < x \leq \frac{1}{3}$.

With this fact, we can formulate the following new lemmas.

Lemma 3.4: In any list L such that $XFD(L) = L^* + n$ with $n \geq 1$, the largest element s_{L^*+i} in bin $L^* + i$, $1 \leq i \leq n$, in the packing P_{XFD} is less than or equal to $\frac{1}{3}$.

Proof: Suppose that we are currently assigning object a_i. If a_i is the first object assigned to bin B_{L^*+1} in XFD, then it means that it does not fit in any bin B_j for $j < L^* + 1$. It also means that $a_r \geq a_i | r < i$. Hence, all objects deployed on this and any subsequent bin would be less than or equal to a_i. As a result, a list $L' = L\backslash a_j | j > i$ exhibits $XFD(L') = L^* + 1$ with a_i as the only element in bin B_{L^*+1}. Hence, by Fact 3.3 we know that $a_i \leq \frac{1}{3}$ and we know that every subsequent object $a_r \leq a_i$ so bins B_{L^*+i} for $1 \leq i \leq n$ can only have objects $\leq \frac{1}{3}$. ∎

Lemma 3.5: In any list L such that $L^* = \lceil \sum a_i \rceil$, all objects a_j partitioned by $PBFD$ are smaller than or equal to $\frac{1}{3}$.

Proof: By the definition of $PBFD$, any object that fits into the primary bins is not partitioned. Hence if $L^* = \lceil \sum a_i \rceil$, then the number of primary bins is equal to L^*. Therefore, all the objects that are partitioned would have been allocated by BFD into bins $B_{L^*+i}, i \geq 1$. By Lemma 3.4, any object allocated by BFD into a bin $B_{L^*+i}, i \geq 1$ is smaller than or equal to $\frac{1}{3}$. ∎

Let us consider a simple example before interpreting the above lemma.

Example: Consider three objects each of size 0.51. We have $\lceil \sum a_i \rceil$ = 2. However, *no* non-partitioning scheme including the optimal scheme can use less than 3 bins. In contrast, with PBFD, we will start with 2 primary bins and the first two objects will be assigned to those 2 bins. The third object will then be partitioned into two pieces of (say) sizes 0.49 and 0.02 and assigned to the two primary bins respectively. In other words, PBFD can do better than the optimal non-partitioning scheme! (We will return to this in the next section).

Remark: Note that Theorem 3.5 applies only when $L^* = \lceil \sum a_i \rceil$. It is *not* always true that $L^* = \lceil \sum a_i \rceil$ as in the example above. This implies that when $L^* > \lceil \sum a_i \rceil$, PBFD can partition objects that are greater than $\frac{1}{3}$ again as in the above example. In these cases, the performance of PBFD will lean away from the worst-case behavior.

Lemmas 3.4 and 3.5 allow us to state our theorems about the worst-case behavior of partitioning schemes. In these theorems, we prove that $PBFD$ can save bins that BFD would require. A generic theorem is stated for partitions of any size and a special case for partitioning in halves.

Theorem 3.6: For all lists L such that $BFD(L) > rL^* + d$, $d \geq 1$, if list L' is a copy of the original list L with every object in B_{last} partitioned into two parts x_1 and x_2 with each part being less than or equal to $\frac{r-1}{r}$, then $BFD(L') \leq rL'^* + d$.

Proof: By Lemma 3.1, if $BFD(L') > rL'^* + d$ then for a list $L'' = L'\backslash x | x \leq \frac{r-1}{r}$, $BFD(L'') > rL''^* + d$ should hold, meaning that such objects do not contribute to the worst case. But we know that $x \leq \frac{r-1}{r}$ for $x \in B_{last}$ and such a bin only contains this kind of objects, meaning that this last bin was freed in packing P_{BFD}. Hence, it must be that

$BFD(L') \leq rL'^* + d$. ■

Note: As mentioned, Baker [3] showed that $BFD(L) > \frac{11}{9}L^* + 3$, i.e. we have $d = 3$ above. Since the statement of Theorem 3.6 requires only that $d \geq 1$, it can be applied twice with partitioning until $d = 1$. In other words, two processors can be saved even in the worst-case configuration of Baker by using PBFD.

Theorem 3.7: In any list L such that $L^* = \sum a_i$ and $BFD(L) > \frac{11}{9}L^* + d$ with $d \geq 1$, if all the objects of any one bin B_{L^*+i} for $i \geq 1$ is partitioned into halves, then the bin can be freed.

Proof: By Lemma 3.5, all the objects that are partitioned are smaller than or equal to $\frac{1}{3}$. Hence, if we split them in halves, each would be of size x, $x \leq \frac{1}{6} < \frac{2}{11}$. By Lemma 3.1, we can delete them and free up that bin. ■

A. Redefining Optimality With Partitioning

We now introduce the novel view that different levels of optimality can be defined in solving the bin-packing problem. Specifically, when partitioning is allowed, three different optimal packings can be defined:

- **Fluid-Flow Optimal (L^*_{ff})**. This refers to an ideal but impractical packing P^*_{ff} if all the objects can be partitioned into arbitrarily fine-grained sizes that could be perfectly accommodated leaving no gaps except in the last bin. In other words, $L^*_{ff} = \lceil \sum a_i \rceil$.
- **Optimal without Partitioning (L^*)**. This refers to the optimal packing P^* when objects cannot be partitioned. This packing could leave gaps in the bins but it yields the minimum number of bins possible given a list L, but no partitioning of objects.
- **Optimal with Partitioning (L^*_p)**. This refers to the optimal packing P^*_p if each object can be broken into a finite set of pre-defined sizes. This packing could leave gaps in any or all of the bins but it yields the minimum number of bins possible under the allowed partitioning constraints.

The relationship between the above optimal packings is that $L^*_{ff} \leq L^*_p \leq L^*$. Let us illustrate this relationship with an example. In this example, we only represent objects by their size (all sizes are less than one and all bins are of size 1), and composite objects by a list of the sizes of the composing objects. For instance, (0.5) represents an object of size 0.5 that cannot be partitioned. Similarly, $(0.3, 0.1)$ represents a composite object of total size 0.4 which can be partitioned into two pieces of size 0.3 and 0.1. Now, consider an object list L with 10 composites of type $a = (0.25, 0.6)$, 10 objects of type $b = (0.35)$, 10 objects of type $c = (0.7)$, and 1 object of type $d = (1.0)$. The optimal packing without partitioning yields $L^* = 26$, allocating one bin for the object of type d, one bin for each of the objects of type a (10 of them), one bin for each of the objects of type c (10 of them), and five bins for objects of type b (with each bin containing two objects of type b). The optimal packing with partitioning yields $L^*_p = 21$. This packing partitions objects of type a into objects of types $a' = (0.25)$ and $a'' = (0.6)$, then creates pairs of one object of type a' with an object of type c (adding up to $0.25 + 0.7 = 0.95$) and allocates them in a bin, using 10 bins to accommodate them all. Then, it creates pairs of one object of type a'' and one object of type b (adding up to $0.35 + 0.6 = 0.95$) and allocates them in a bin, using 10 bins. The last bin is then assigned the only object of size 1.0. Finally, the fluid-flow optimal would have the same packing as the optimal with partitioning for the first 20 bins. However, the last object of size 1.0 is split into 20 fragments of size 0.05[7] and puts these fragments into each of the initial 20 bins, leading to a packing with only $L^*_{ff} = 20$ bins. Contrary to the optimal bin packing with partitioning (that does not incur in any penalty for partitioning) our algorithms avoid partitioning objects as long as they fit into a bin to prevent the creation of messages across processors. This behavior and the decreasing order (by size) of BFD lead to the partitioning of the smallest objects. However, these algorithms can still partition objects that are still too big to be accommodated in the primary bins. Another algorithm called *Excess-bin Early Partitioning Best-Fit Decreasing* is specifically designed to avoid breaking objects that could produce fragments too big to be accommodated in the primary bins. We describe this algorithm next.

B. Excess-bin Early Partitioning Best-Fit Decreasing (EEPBFD)

We refer to bins that are allocated in addition to the number of bins allocated by the optimal algorithm with partitioning (i.e. in excess of L^*_p) as *excess bins*. This definition allows us to define a new partitioning algorithm that limits its partitioning only to objects that would have been allocated to the excess bins by the optimal packing. Specifically, this algorithm limits its partitioning to objects of sizes $\leq \frac{1}{3}$. In other words, if the algorithm runs out of primary bins, it will partition an object that does not fit into any gap only if its size is $\leq \frac{1}{3}$ and would put the pieces back into the list to be packed. Otherwise, it will allocate a new bin and allocate the object to that new bin.

Theorem 3.8: In any list L such that $BFD(L) > \frac{11}{9}L^* + d$ for $d \geq 1$, if all the objects partitioned by $EEPBFD$ are split in halves, then it will save at least one bin.

Proof: Given that all the objects partitioned by $EEPBFD$ are $\leq \frac{1}{3}$, then their halves are $\leq \frac{1}{6} < \frac{2}{11}$. By Lemma 3.1, we can delete these halves. By Lemma 3.4, we know that all excess bins will have objects $\leq \frac{1}{3}$ and therefore partitioning all the objects that fill up one bin will free up this bin. ■

C. Conjecture about PBFD

There are several degrees of freedom in designing a partitioning bin-packing scheme of polynomial complexity. These degrees of freedom include the number of bins a scheme starts with, the points when a new bin can be added, the partitioning choices for an object, and the sizes of partitioned pieces of an object. In other words, there is a large family of partitioning bin-packing algorithms.

We now make the following conjecture that the worst-case performance of partitioning bin-packing algorithms is significantly better than that of BFD schemes.

[7]Fluid-flow is allowed to partition into arbitrary sizes.

Conjecture: $PBFD(L) > \frac{10}{9}L^* + k$ where $k \geq 0$.

In other words, we believe that an appropriate partitioning bin-packing scheme can go beyond reducing the constant factor, and reduce even the worst-case proportional fraction $\frac{11}{9}$ to $\frac{10}{9}$ relative to an optimal non-partitioning scheme. Note that our conjecture characterizes the *worst-case* performance of PBFD. We already know that PBFD can actually perform better than even an optimal non-partitioning scheme in some cases as shown in an example earlier.

D. Summary of Theoretical Results

We now summarize the three theoretical results presented in this section.

First, we proved that $PBFD$ can save the bins in excess of $\frac{11}{9}L^* + 1$ with respect to BFD in the worst case. We prove this for the general case when the objects in the bins being saved can be partitioned to sizes less than or equal to $\frac{2}{11}$. In the average case, the savings can be much higher as will be seen next in our experiments.

Secondly, we showed that when the sizes of the objects are strictly partitioned into equal halves, the processor savings still hold.

Thirdly, we showed that if we limit the partitioning to objects of sizes less than or equal to $\frac{1}{3}$, at least one bin can be saved.

Finally, we also conjectured that the worst-case performance of a partitioning bin-packing scheme can be significantly better than that of a classical bin-packing scheme.

IV. Bandwidth Management

The partitioning of objects that PBFD performs does carry a cost: the bandwidth of the messages sent between the parts a composite was broken into. In this section, we extend the basic PBFD algorithm to minimize the number of partitions needed, and if possible the bandwidth generated by such partitions. Specifically, we present variations of bin-packing schemes guided by our earlier analysis to take into account the bandwidth that partitions generate. The design of these schemes is followed by its empirical evaluation. We start the discussion with the early partitioning approach and follow it by the design of two late-partitioning algorithms.

A. Cycle-Driven Early-Partitioning Bin-Packing (CEP)

This approach attempts to minimize both the CPU cycles and the network bandwidth needed for the modules. To minimize CPU cycles, the traditional approach of adding the largest module to where it fits best is used. To minimize bandwidth, the approach is to partition modules to the largest size that can fit into the processor with the most available cycles. This scheme assumes that partitioning a large object generates a large bandwidth demand between its parts. As a result, as soon as it finds the largest object that does not fit any available gap, it immediately partitions this composite to generate the largest bandwidth requirement as early as possible. The assumption is that it is more likely to find a larger gap in the network capacity at this early stage.

B. Bandwidth-Driven Late-Partitioning Bin-Packing (BLP)

The objective of this algorithm is to minimize both the number of processors and link capacities. To achieve this, we go through the processors in decreasing order and fill each one as much as possible. To fill the processors, the software graph is divided into disconnected sub-graphs. These sub-graphs are deployed in decreasing order of size until no more sub-graphs can fit into the processors. At that time, we select a sub-graph and partition it into further communicating sub-graphs, adding the pieces back to the list of sub-graphs to be deployed. We use the following efficient heuristic to pick the sub-graphs. The selection of the sub-graph to be partitioned has the purpose of reducing the size of the graph cut (which represents the bandwidth requirement between partitions). To perform this selection, we mark each sub-graph (at creation time) with a *bandwidth compression factor* that will be described shortly. The bandwidth compression factor represents the bandwidth of the messages exchanged among internal nodes. Because these messages are internal to the processor, they do not demand link bandwidth if the sub-graph is deployed on a single processor and hence is compressed away from the total demand. Moreover, when partitioning a sub-graph, the larger this factor, the larger the potential bandwidth demand generated by such a partition. With late partitioning, we partition composites that are more likely to generate more bandwidth earlier when it is more likely to find larger gaps in the network.

The bandwidth compression factor is calculated as follows. First, partitioning of sub-graphs is done starting from the node with higher bandwidth demand and adding the neighbors with which it shares the highest bandwidth. The neighbor addition is repeated until the appropriate partition size is reached. In general, this partitioning scheme has a tendency to decrease the bandwidth demand of the new parts (local increases are not unusual, but the general trend is to slope down). Figure 1 plots an idealized decreasing curve of the bandwidth demand from the partition to other pieces. At the origin of the X axis is the first partition when the size is small and the communication of that small part is the largest toward all the other parts. On the other extreme of the X axis, the bandwidth demand drops to zero when all the composing elements have been added to the part and all the internal communication has been compressed away. Second, on top of this curve, a fading slope line is drawn to simplify calculations. Finally, the area under this line is calculated and used as the bandwidth compression factor.

C. Cycle-Driven Late-Partitioning Bin-Packing (CLP)

In bandwidth-driven late-partitioning bin-packing, bandwidth demands were used as the primary metric to determine partitions. The CLP (Cycle-Driven Late-Partitioning Bin-Packing) scheme is designed to use the sub-graph size as the primary determinant for making partition decisions. The rationale is that the largest composites are likely to be allocated early when there are multiple big gaps available in the bins.

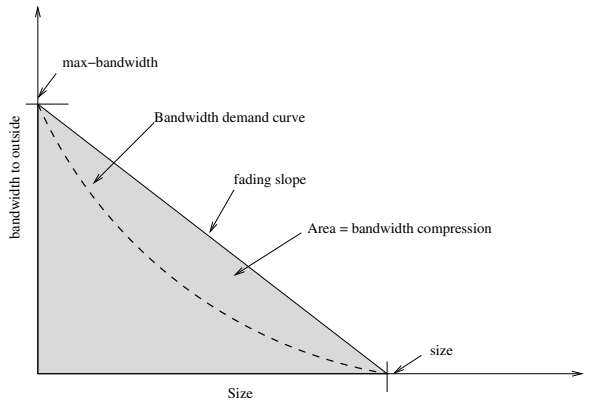

Fig. 1. Bandwidth Compression Factor

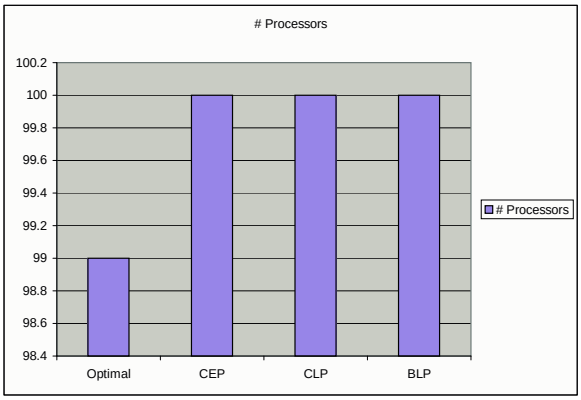

Fig. 2. Average Performance - Num. Processors

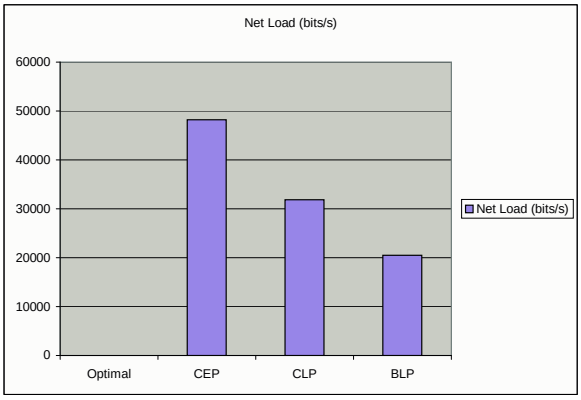

Fig. 3. Average Performance - Network Load

V. Experiments

We now evaluate the above 3 schemes for their efficacy in minimizing the number of bins and network bandwidth. We compare each scheme against an optimal allocation. Given the exponential complexity of finding the optimal allocation without partitioning, our experiments are designed such that an optimal allocation is known a priori by construction. Specifically, we pick a desired number of bins, and for each bin, we assign randomly chosen modules until it is nearly full. Then, we fill the gap that remains with an object of exactly that size. We then randomly assign communications and message lengths among modules in the bin. Objects that communicate become composites. This configuration now represents an optimal packing. The workload of composites in the configuration is then given to the bin-packing algorithms to be deployed and their performance is compared against the optimal (which by construction corresponds to the number of bins = total workload and zero communication bandwidth).

A. Average-Case Behavior

For the average case, random module sizes are created. The parameters of our experiment are:

- 1 Mbps links
- 1 GHz Processors
- Module sizes vary with uniform distribution between 1 - 45 % of the processor utilization,
- Messages bandwidth varies uniformly between 64 to 50,000 bits/s, and
- A Module load of 99 processors.

Figure 2 depicts the average over 30 runs on the number of processors used by the different algorithms. As can be seen, in the average case, all the algorithms behave well, allocating only one extra processor relative to the optimal scheme. The difference between the algorithms lies in the amount of bandwidth used for the messages across processors. Figure 3 depicts this difference when the message bandwidth across modules is randomly chosen within a range from 64 to 50,000 bits/s. This large bandwidth range highlights the difference in the allocation algorithms when bandwidth is considered. In this case, the difference between the Cycle-driven Early Partitioning and the Bandwidth-driven Late Partitioning algorithms is more than twice with respect to bandwidth used. However, if the bandwidth is chosen from a narrow range, the performance difference is reduced. Figure 4 depicts an experiment where the messages are allowed to range between 10,000 and 50,000 bits/s. In this case, the difference between the network bandwidth allocated by the Cycle-driven Early Partitioning and the Bandwidth-driven Late Partitioning algorithms is reduced to about 25%.

B. Worst-Case Behavior

The experiments presented in this section are based on a worst-case workload pattern presented in [8]. This pattern forces a performance ratio of $\frac{11}{9}$ between BFD and the optimal algorithm. The modules from the worst-case pattern (assumed to be composite modules) can be partitioned and further messages are added among the parts every time they are partitioned (to encourage their joint deployment). Finally, the modules are put in a list of modules to be deployed and the deployment algorithms are run.

The objective of this pattern is to test our algorithms under "bad" cases. We know the analytical worst case (verified experimentally) and we also know that on average, all the algorithms behave fairly well. Hence, we want to evaluate them in a region of bad cases. To perform this evaluation, an optimal module load is created and further partitioned randomly to enable a variety of module sizes and messages

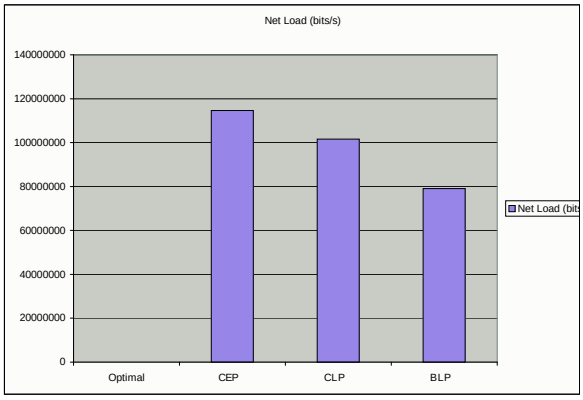

Fig. 4. Average Performance - Network Load Small Range

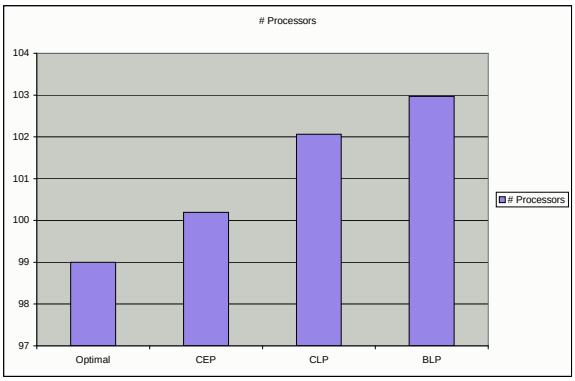

Fig. 6. Worst-Case Pattern Number of Processors Allocated As **Compared to 121 Processors without Partitioning**

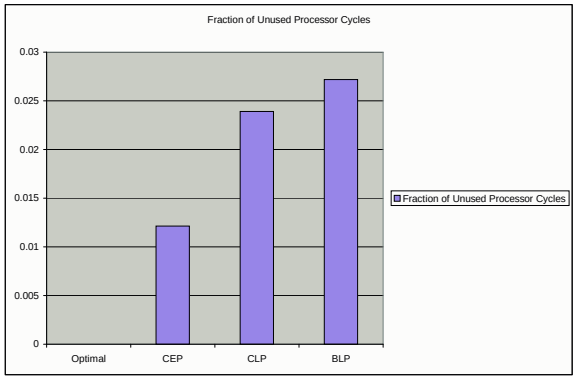

Fig. 5. Worst-Case Pattern- Fraction of Unused Processor Cycles

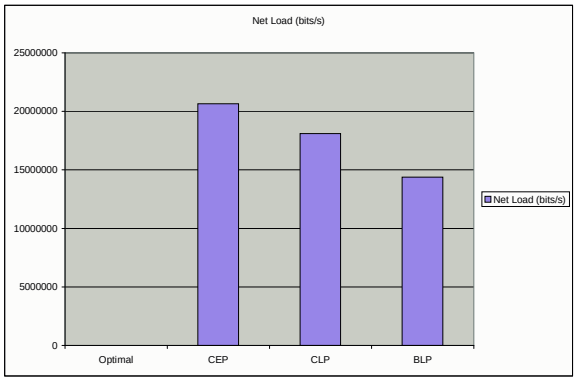

Fig. 7. Worst-Case Based - Network Load

among them. Then, the algorithms are run to evaluate their performance while deploying the modules. Thirty runs are averaged and the following measurements are evaluated:

- *Fraction of Unused Processors*. This figure represents the total fraction of processors that was unused and left idle.
- *Number of Processors*. The total number of processors allocated.
- *Network Message Bandwidth*. The bandwidth required by the messages sent across a network. Given the configuration of the optimal module load, the optimal algorithm does not require messages sent across the network.

Figure 5 depicts a bar chart that compares the unused processor cycles for the Bandwidth-Driven Late Partitioning, Cycle-Driven Late Partitioning, Cycle-Drive Early Partitioning, and Optimal algorithms. Figure 6 depicts the actual number of processors allocated. As can be seen, early partitioning has an advantage both in the fraction of unused processors and the number of processors. This is explained by the fact that the earlier the modules are partitioned, the larger are the gaps available to accommodate large modules. However, the fraction of unused processors in all three cases is under 3%.

Figure 7 depicts the network load in bits per second due to messages among modules on different processors. This figure depicts the effect of partitioning early, i.e., it creates a larger load in the network. On the other hand, among the late partitioning algorithms, when partitioning is done based on bandwidth, the network load generated is lower that when partitioning by CPU cycles.

C. Verification of EEPBFD Improvement

We next conduct an experiment to evaluate Excess-bin Early Partitioning BFD ($EEPBFD$). This experiment uses a workload where the optimal algorithm with partitioning uses more bins than the fluid-flow optimal. To do this, we create an optimal load with small gaps in each processor. Then, the number of processors for the optimal load is chosen to be 200 to aggregate the gaps into one full processor ($200 * \frac{1}{200}$). The $EEPBFD$ algorithm uses a guard that avoids partitioning modules that are $> \frac{1}{3}$, while acting as CEP otherwise.

Figures 8 and 9 depict the fraction of unused processors and the number of processors allocated respectively. As can be seen, when the excess-bin guard is used, then we are able to save one processor with respect to using CEP. It must be noted that without partitioning, an additional 43 processors would be required. However, as depicted in Figure 10, $EEPBFD$ can slightly increase the network load of the partition.

VI. Conclusions

We have proposed a family of *partitioning bin-packing* algorithms that can potentially partition objects to be packed

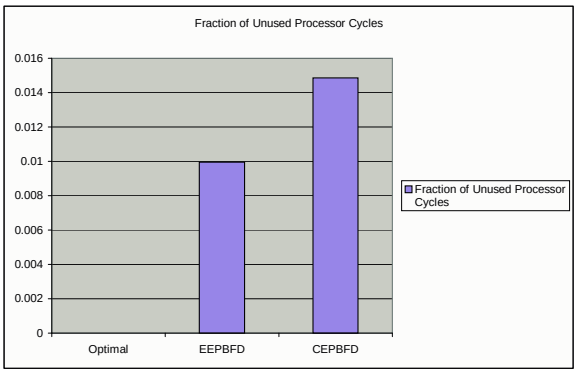

Fig. 8. Excess-bin Partitioning - Unused Processor Cycles

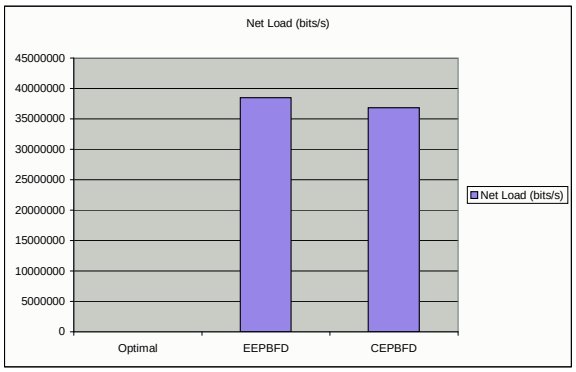

Fig. 10. Excess Partitioning - Network Load

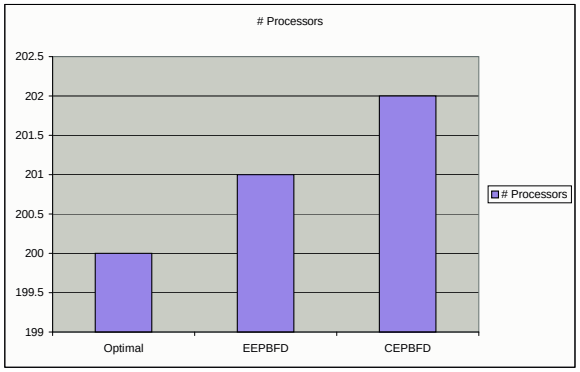

Fig. 9. Excess-bin Partitioning - Number of Processors Allocated As **Compared to 244 without Partitioning**

and allocate them as smaller objects. These partitioning algorithms are polynomial in complexity, but in some cases can perform better than even optimal schemes without partitioning that have exponential complexity. We extend the analysis of the worst-case performance of bin-packing to prove that partitioning components leads to a reduction in the worst-case bounds on the number of processors required. We also conjecture that in the worst case, one processor every 10 processors can be saved and pose its proof as an open problem.

Our analysis also leads to a new algorithm called *Excess-bin Early Partitioning BFD* ($EEPBFD$) that avoids partitioning objects unnecessarily. This algorithm is extended to concurrently minimize both the number of processors needed for the components and the number of links needed for the messages across processors. Three other schemes are developed: Cycle-based Early Partitioning (CEP), Cycle-based Late Partitioning (CLP) and Bandwidth-compression-based Late Partitioning (BLP). These algorithms model the software system as a collection of composite components that encapsulate components that communicate with each other. The algorithms then try to deploy each of these composites into a single processor and upon failure, they decompose a component into parts, and attempt to minimize the communication among the parts. Such parts are then evaluated again for deployment. CEP assumes that keeping the largest components together automatically minimizes bandwidth and hence tries to avoid the partitioning of large objects. However, if a composite cannot be packed, it is immediately partitioned. Both CLP and BLP defer the partitioning until the late stages, performing a full pass on all components that can be deployed without partitioning. Once there are no more components that fit available bins, the deferred components are partitioned and deployed. Two choices are used then to partition the objects: one based on the CPU cycles needed by the object (CLP), and another based on the calculated amount of bandwidth that the internal messages require (called *bandwidth compression factor*).

Finally, experiments show that, in the average case, the three algorithms behave relatively the same. However, the experiments for the worst-case region show that in general CEP behaves well and uses the least number of processors. On the other hand, BLP provides the minimum number of links required by the system while utilizing about 4% more processors than CEP. These schemes allow us to tradeoff processors against links depending on the specific needs of the system.

Future work can prove (or disprove) our conjecture. Furthermore, the cost of partitioning in terms of processor cycles must be accounted for. Finally, more complex hardware topologies must be considered.

REFERENCES

[1] Take F. Abdelzaher and Kang G. Shin. Period-Based Load Partitioning and Assignment for Large Real-Time Applications. *IEEE Transactions on Computers*, 49, 2000.

[2] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceesings 8th IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[3] Brenda S. Baker. A new proof for the first-fit decreasing bin-packing algorithm. *Journal of Algorithms*, 6:49–70, 1985.

[4] Coffman, E.G., Jr., M.R. Garey, and D.S. Johnson. Bin Packing with Divisible Item Sizes. *Journal of Complexity*, 3(4):406–428, December 1987.

[5] Dionisio de Niz and Raj Rajkumar. Glue-code Generation: Closing the Loophole in Model-based Development. In *IEEE Real-time Systems and Application Simposium (RTAS) 2004 - MoDES*, Toronto, CANADA, June 2004.

[6] Dionisio de Niz and Raj Rajkumar. Time Weaver: A Software-Through-Models Framework for Embedded Real-Time Systems. In *Proceedigs of ACM Language Compilers and Tools for Embedded Systems Symposium 2003*, San Diego, CA, June 2004.

[7] D.S. Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[8] D.S. Johnson, A. Demers, D. Ullman, M.R. Garey, and R.L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal of Computing*, 3:299–325, 1974.
[9] Mark H. Klein, Thomas Ralya, Bill Pollak, and Ray Obenza. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.
[10] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.
[11] Matt W. Mutka and Jong-Pyng Li. A Tool for Allocation Periodic Real-Time Tasks to a Set of Processors. *Systems Software*, 29:135–148, 1995.
[12] Nir Naaman and Raphael Rom. Packet Scheduling with Fragmentation. In *Proceedings of the IEEE INFOCOM Symposium 2002*, New York, NY, June 2002.
[13] Ken Tindell, Alan Burns, and Andy Wellings. Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy. *Real-Time Systems*, 4(2):145–65, 1992.

APPENDIX

`Time Weaver` is a model-based development framework for embedded real-time systems with two key contributions: (a) the reduction of complexity in the modeling of embedded real-time system through novel decomposition structures and (b) the integration of analysis and synthesis algorithms to the framework to automate design choices that together with a code generation framework is able to produce the final runtime system.

`Time Weaver` provides two key modeling abstractions to construct the decomposition structures of an embedded system. These abstractions are *couplers* and *semantic dimensions*. A coupler describes a relationship between software modules. Multiple couplers can be added among the same group of modules (two or more) to describe the multiple relationships that exist among them. For instance, two modules A and B can be related because A sends data to B. In addition, module B can also be related to another module C in a mutual-exclusion relationship, i.e., only one of the two can be running at any given time. Both of these sample relationships are encoded with couplers in our framework enabling a module to be involved in multiple relationships (*couplings*) at the same time. Such couplings are classified into types that can be extended as needed. Before discussing these types of couplers, we introduce the structure of the `Time Weaver` components.

A. Components

Software modules in `Time Weaver` are known as *components*. These components are modules that communicate to other components through ports. Ports communicate events that represent occurrences in a component that need to be communicated. These events can contain data and are typed. As a consequence, ports are considered to be *typed* and they are restricted to communicate a single type of event. Such typed ports can only be connected to ports of the same type.

Components encapsulate a piece of code in an element called *Application Agent*. The reaction of the application agent to the events received by the component is modeled in a *transition table* that encodes a transition for each type of event it can receive. This transition captures the worst-case execution time needed to process the event and the set of output events that it generates. Inside the ports two additional types of pieces of code are present: *protocol agents* and *state managers*. Protocol agents encapsulate the communication protocol used to send events to the ports of other components connected to this port. Protocol agents define also a worst-case execution time needed to run the communication protocol. Protocol agents can be active, effectively encapsulating a thread inside. When a receiving protocol agent (in an input port) is active, it receives the event from the sending protocol agent and continues the processing of the event with its own thread.

State managers, on the other hand, encapsulate a synchronization protocol that behaves like an event valve, retaining or allowing the pass of events from the protocol agent to the application agent in input ports and the opposite for output ports. The state managers that participate in this synchronization protocol are from different ports from different or the same component. For instance, to implement a mutually-exclusive execution of two components A and B that receive an event and generate another as a response, their input and output ports are added state managers that lock a mutex (common to all four state managers) upon arrival of an event to the input ports and unlocks it when the output event is generated. The synchronization protocol of the state managers is modeled in a transition table, similar to the one used by the application agent, that encodes the transitions that occur in the protocol.

Components have two special ports: the *Activation* port and the *Completion port*. The activation port can generate a special *Tick* event periodically when it is set as active. The period of the tick can be set as a port property. In addition, activation ports can have a deadline specified to define the time when the computation initiated by this timer is expected to complete.

The completion port receives a *Completion* event when the computation triggered by an input event finishes. This port is intended to be used to host state managers that need to synchronize at the completion of the components computation.

B. Coupler Types

The different types of relationships among components are modeled with different types of couplers. The basic coupler types are as follows:

- **Event** Couplers. This type of couplers encodes communication relationships among ports.
- **Synchronization** Couplers. This type of couplers defines both a group of ports to be involved in a synchronization protocol and a state manager that implements and models such a protocol. When ports are associated to a synchronization coupler, copies of the state manager specified by the coupler are inserted into such ports and the synchronization group is set in each of these state managers.
- **Property** Couplers. This type of couplers defines relationships among properties of the ports. For instance one such a coupler can defined a common period to be used by the activation port of two components.
- **Constraint** Couplers. Define restriction to be honor by synthesis algorithms. For instance, a *disjoint deployment* constraint can be associated to the replicas of a replicated component to ensure they are deployed on different processors enabling failure independence.
- **Hardware** Couplers. Describe the hardware upon which components can be deployed.

These basic types of couplers can be used recursively to create composition/decomposition structure that forms the basis of the complexity reduction in `Time Weaver`.

C. Composition Structures

Components can be associated into composite components with composite ports. To these composite ports multiple component ports and other composite ports can be associated. Composite ports can be coupled with event couplers. This coupling has the same effect of connecting all the composing ports individually. However, only ports of the same type and opposite direction (input or output) are effectively connected.

The relationships modeled by couplers can be composed coupling couplers with another one. Composite couplers can define *coupling ports* that allows couplers to be defined over such ports that can later be associated to the final component ports. For instance, in a replication composite coupling, two coupling ports can be defined (one for each replica). Over these ports a synchronization coupler can be defined with a state manager that ensures that the input events are received in all the replicas (two in this case) before passing it to the application agent. In addition, a *disjoint deployment* coupler can also be defined over the coupling ports to ensure that the components connected to such ports are not deployed on the same processor. This composite coupler can be readily reused in other systems as needed.

Property couplers are composed defining relationships among components. For instance, a root period can be defined by a coupler C_0 that couples two property couplers C_1 and C_2. C_1 can then defined a period to be twice the period of the root coupler and C_2 can define its period to be half of the roots period.

Finally, hardware couplers are composed with two purposes. First, these couplers are used to describe the communication hardware such as a network connecting two processors. Secondly, the hardware couplers are also used to describe communication scopes. Communication scopes identify groups of connected ports (with event couplers) such that all the members of the group are connected only to other ports also associated to this scope. For instance, consider two processor couplers P_1 and P_2 that are coupled by a network coupler N_1 and two components C_1 and C_2 associated to (coupled by) P_1 while a third one C_3 is associated to P_2. If the port $C_1.out_1$ is connected to $C_2.in_1$ and $C_1.out_2$ is connected to $C_3.in_1$ then port group consisting of ports $C_1.out_1$ and $C_2.in_1$ belongs to the communication scope defined by P_1 while the port group consisting of ports $C_1.out_2$ and $C_3.in_1$ belongs to the communication scope defined by N_1. Protocol agents are added to hardware couplers to specify the communication protocol to be used by the ports in their communication scopes. For instance a TCP protocol agent associated to N_1 would change the protocol agent in ports $C_1.out_2$ and $C_3.in_1$ for TCP protocol agents.

D. Semantic Dimensions

The multiple types of relationships couplers can model can belong to different aspects (functionality, fault-tolerance, timing, etc.) of the system. For this reason, our framework separates these types of couplers into different views each of them focusing on a single aspect. These views can be modified independently from the user point of view. The realizations of these views, however, do interact on the target platform since they consume common resources and impose constraints. These interactions can also be captured during model construction and resource demands mediated during platform deployment. Given the automatic resolution of conflicts these views have stronger semantics and are called *semantic dimensions*. Our framework defines a set of basic semantic dimensions for embedded real-time systems and enables de addition of new dimensions.

E. Glue Code Generation

`Time Weaver` has a glue code generation framework that links the executable code inside the executable elements of the components (application agents, protocol agents, and state managers) to build the final running system. In this framework it is possible to define one target programming language per process, assuming that an implementation in this programming language is available for all the executable elements of the components attached to this process. Optimizations to this code generation has been implemented, the most effective been based on graph reinterpretation when components are assumed to be data flow components [5].

F. Analysis / Synthesis

Analysis and synthesis are two key capabilities of `Time Weaver`. It is able to generate a timing model that can be exported to timing analysis tools such as TimeWiz®. Synthesis algorithms are a central part of our framework to automate design decisions in an optimal or near-optimal way. In this paper we present our extensions to bin-packing algorithms to synthesize the mapping of components to hardware.