Closeness metrics for system-level functional partitioning

Frank Vahid

Technical Report CS-94-04
September 15, 1994

Department of Computer Science
University of California
Riverside, CA 92521
vahid@cs.ucr.edu

Abstract

As methodologies and tools for chip-level design mature, design effort becomes focused on increasingly higher levels of abstraction. A crucial higher-level design task is the partitioning of system functionality for implementation on multiple system components, including partitions among hardware and software components. We present a set of closeness metrics to aid such partitioning. These metrics can be used by a designer or by automated algorithms, to cluster together functional objects that should be implemented on the same component. Our experiments demonstrate that the metrics can be used to reduce the time required to obtain high-quality partitions, as well as to guide designers during interactive partitioning.

Contents

List of Figures

.

1 Introduction

A system's functionality is usually implemented with a set of system components. Typical system components include standard processors, microcontrollers, custom processors, multi-chip modules, ASICs, blocks on an ASIC, memories, and buses. For example, consider the design of an ATM switching system, whose functionality is that of capturing incoming packets of data, and transmitting that data to other destinations. Such a system may consist of multiple ASICs, where a single ASIC may itself contain an embedded processor core along with numerous memories and custom processor blocks. Time-critical behaviors, such as data capture, may be implemented using custom hardware blocks, perhaps being divided among multiple blocks to satisfy block-size constraints or to permit concurrent design. On the other hand, less critical behaviors, such as that of framing data for transmission, or behaviors likely to change in the future, may be implemented as software on the processor core.

To achieve such an implementation, a system designer must solve the difficult problems of selecting the system components, and of partitioning the functionality for implementation on those components, such that design constraints are satisfied. The difficulty of these problems stems from the enormous number of possible solutions, and from the lack of sophisticated tools to aid the designer when exploring various options.

The current approach to solving these problems forms a gray area in current design methodology. Presently, designers start with an informal functional specification (usually written in a natural language like English), and then apply manual estimations and previous experience to achieve a system-level design. Such an approach usually leads to a suboptimal design and to long component integration and testing times. For example, when designing the above ATM switch, the inability or excessive time required to obtain accurate software-performance estimations often causes over-use of hardware to ensure that performance is satisfactory. Furthermore, integrating the hardware and software components often proves extremely time-consuming, due to ambiguous specifications and to late correction of functional errors.

With the maturation of the design tasks that follow system design, such as real-time software design and custom processor design (using behavioral synthesis), many researchers have begun to focus their efforts on developing approaches that overcome the shortcomings of current system design practice. Such approaches start with a formal, simulatable specification of functionality. The pieces that make up that specification are then partitioned, with the assistance of automation tools, for implementation among system components. By using a simulatable specification, these approaches enable early detection of functional errors, which in turn greatly reduces the time to correct such errors, as compared to the current practice of detecting and correcting such errors only after implementation. These approaches can also use automated estimators and partitioning algorithms to explore many more solutions than previously possible.

In this paper, we discuss closeness metrics for automated functional partitioning of a

specification among hardware or software components. A **closeness metric** measures the likelihood that two pieces of the specification should be implemented in the same system component. For example, if two system functions use the same data, execute sequentially, and have the same hardware requirements, then implementing them on the same system component would likely lead to a good design. *Closeness metrics can be especially useful* for guiding partitioning decisions when global metric values, such as overall performance, size, and pins, are not yet available. Global metric values are unavailable when we are in the process of forming an initial partition, since such values are based on a complete partition, or when estimators have not yet been implemented for those metrics. Closeness metrics can also useful for suggesting modifications of an existing partition that might lead to improved global metric values.

Our experiments show that these closeness metrics lead to high-quality partitions and reduced partitioning time when used in conjunction with well-known clustering algorithms. The metrics can also be used to suggest relocations of functions to other components to a designer during interactive partitioning. As we shall see, the features that distinguish these metrics from similar research efforts are the high level of abstraction at which partitioning is done, and the thorough covering of the system design problem. The paper is organized as follows. In Section 2, we provide a problem definition. In Section 3, we define several useful closeness metrics between coarse-level pieces of a functional specification. In Section 4, we provide the results of several experiments on practical examples. In Section 5, we highlight related research. In Section 6, we provide conclusions and future work.

2 Problem definition

In this section, we will first highlight our approach to system design, describing the central role of the partitioning task. We will then provide a formal definition of the problem of partitioning functional objects for implementation among system components.

2.1 System design

We define system design as the problem of mapping a system's functionality, as captured with an executable specification, to some set of system components such that design constraints on parameters such as monetary cost, performance, and power are satisfied. Our approach to system design consists of three well-defined tasks on three classes of functional objects, as summarized in Figure 1. The three classes of **functional objects** that comprise any executable specification are variables, behaviors, and channels. **Variables** store data, **behaviors** transform data, and **channels** transfer data between behaviors. In our terminology, a behavior is a computation that is coarser grained than an arithmetic operation. Instead, it corresponds to a block of statements in the specification such as a loop body, procedure, or process. The abstraction level of a behavior is sometimes called the "algorithmic" level in other work. A behavior is similar to a "task" described in [1].

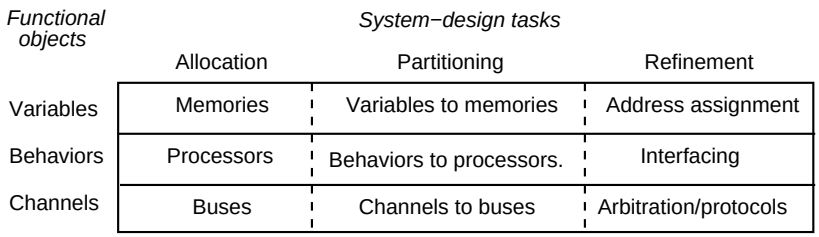

Figure 1: System-design tasks

For each of these functional objects, there are three tasks to be performed: allocation, partitioning, and refinement. **Allocation** adds system components to the design. One class of system component consists of memories, such as RAMs, ROMs, register-files, and registers. Memories are used to store scalar and array variables. Another class of component consists of standard processors and microcontrollers as well as custom ASIC "processors". These standard/custom processors are used to implement behaviors and variables. A third class of "component" consists of physical buses. Buses are used to implement communication channels. **Partitioning** maps each class of functional objects to allocated components. Behaviors are mapped to standard/custom processors, variables are mapped to memories or processors, and channels are mapped to buses. Each mapping is many to one. **Refinement** adds new behaviors to maintain correct functionality for a given allocation and partition. Variables partitioned among memories require memory address translation. Behaviors separated among components must be modified to maintain correct communication. Channels mapped to buses require interface synthesis to determine communication protocols, and arbiter synthesis to resolve any simultaneous bus requests. After allocation, partitioning, and refinement, a new specification can be generated consisting of a set of interconnected system-components, each functionally specified.

Given the above definition of system design, we can now proceed to formally define the problem of partitioning a specification among a set of system components. We first define our internal format of the specification, then the desired output, and finally the procedures we assume are available to us.

2.2 Specification-level access graph

We read the input specification into an intermediate form that is intended to represent the access relationships between the coarse-level computations of the specification. This intention is in contrast to that of most high-level synthesis internal forms, which seek to expose arithmetic-level data dependencies [2, 3, 4, 5]. For this reason, we refer to the form as a specification-level access graph, or AG. The AG is a directed graph. Each node of the graph represents a behavior or a variable from the specification. A directed edge, or **channel**, represents an access by the source behavior to another behavior (subroutine call)

or to a variable or external port (data read or write). It may also represent data being transferred from the source behavior to another behavior as specified by a message passing construct in the input specification.

An edge has two weights. The *accfreq*, or access-frequency, weight indicates the number of times the access occurs during an average start-to-finish execution of the source behavior, as determined from a branch probability file. (Minumum and maximum accfreq weights can also be associated with each edge in a similar manner). The branch probability file may be obtained manually or through profiling. The second type of weight is the *bits* weight, which indicates the number of bits transferred during an access. For access to a scalar, this is the number of bits into which the scalar would be encoded. For access to an array of scalar elements, this is the number of bits to encode an array element, plus the number of address bits needed to specify an element's address. For a more complex data item, such as a three-dimensional array, the data item is first transformed to an array of scalars. For access to another behavior, the bits weight is the number of bits needed to transfer all (if any) parameters, where the number of bits for each parameter is computed as in the case of variables. For a message pass, this is the number of bits into which the message would be encoded.

The AG can be defined formally as a three-tuple $< BV_{all}, IO_{all}, C_{all} >$. BV_{all} is a set of all behaviors and variables derived from the specification, i.e., $BV_{all} = B_{all} \bigcup V_{all}$, where B_{all} is the set of behaviors $\{b_1, b_2, ...\}$, and V_{all} is the set of variables $\{v_1, v_2, ...\}$. IO_{all} is the set of all input and output ports of the specification, i.e., $IO_{all} = \{io_1, io_2, ...io_p\}$. Finally, C_{all} is the set of all channels derived from the specification, i.e., $C_{all} = \{c_1, c_2, ...\}$. Each channel c_i is a four-tuple $< src, dst, accfreq, bits >$, where $src \in B_{all}$, $dst \in BV_{all} \bigcup IO_{all}$, $accfreq \in Positive$, and $bits \in Positive$. src is the accessor behavior. dst is the behavior, variable or port accessed by src. $accfreq$ is the average number of accesses to the channel. $bits$ is the number of bits transferred per access.

As a simple example, consider the partial specification of a fuzzy-logic controller shown in Figure 2. Two inputs, *in1* and *in2*, must be converted to a single output, *out1*, using the rules of fuzzy logic. Such controllers are common in numerous consumer applications, such as video camera focus control, thermostat control, and automobile cruise control, in which smooth transitions are needed from one output value to another. The main process *FuzzyMain* first samples the values on the inputs *in1* and *in2* by writing them into variables *in1val* and *in2val*. It then calls a procedure *EvaluateRule* twice, once for each input, and that procedure fills the values of an array (*tmr1* or *tmr2*) based on the input value and on the values contained in another predefined array (*mr1* or *mr2*). After other tasks are performed on the new *tmr* arrays (ommitted from the figure for brevity), including convolution, a centroid value is computed, and that value is then output. The process then repeats after a specified time interval.

The basic AG representation of this specification is shown in Figure 3. Each process, procedure and variable has its own node. Each variable or procedure access has its own edge. For example, the writing of *in1val* in *FuzzyMain* translates to an edge between those

```
entity FuzzyControllerE is
  port ( in1, in2 : in integer;  out1:  out integer );
end;
...
FuzzyMain: process
      variable in1val, in2val : integer;
      type mr_array is array (1 to 384) of integer;
      variable mr1, mr2: mr_array; -- membership rules
      type tmr_array is array (1 to 128) of integer;
      variable tmr1, tmr2: tmr_array; -- truncated memb. rules
      ...
begin
      in1val := in1;  in2val := in2;
      EvaluateRule(1);
      EvaluateRule(2);
      Convolve;
      out1 <= ComputeCentroid;
      wait until ...
end process;


procedure EvaluateRule(num : in integer) is
   variable trunc : integer;  -- truncated value
begin
   if (num = 1) then
      trunc := Min(mr1(in1val), mr1(128+in1val));
   elsif (num = 2) then
      trunc := Min(mr2(in2val), mr2(128+in2val));
   end if;

   for i in 1 to 128 loop
      if (num = 1) then
         tmr1(i) := Min(trunc, mr1(256+i));
      elsif (num = 2) then
         tmr2(i) := Min(trunc, mr2(256+i));
      end if;
   end loop;
end;
```

Figure 2: Partial VHDL specification for a fuzzy-logic controller system

two nodes. *accfreq* is 1 since the access only occurs once, while *bits* is 8 since the data being accessed is 8 bits (assuming that an integer is encoded with 8 bits). As another example, the two calls of *EvaluateRule* by *FuzzyMain* also translate to a single edge between those two nodes. In this case, *accfreq* is 2 because there are 2 calls, while *bits* is 8 because an integer is transferred during the call. Finally, the accesses of *EvalutateRule* to *mr1* result in an edge with an *accfreq* of 65, and a *bits* value of 17 (8 data bits plus 9 address bits). The *accfreq* is 65, rather than 130, because the first time the procedure is called, *mr1* is accessed 130 times, while the second time, *mr2* is called 130 times, meaning that the average accesses to each variable per call is *65*. (Note that minimum and maximum values, though not shown, would be 0 and 130 respectively). Note that the direction of an edge indicates the direction of an access (i.e., it indicates the accessor and accessee), *not* the direction of the flow of data.

Performance constraints (minimum, maximum, or average) can be specified on the ex-

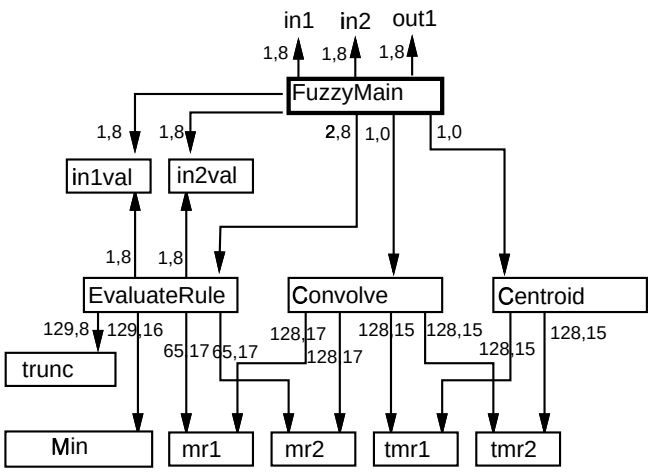

Figure 3: Basic AG for the example system (some nodes and edges omitted for brevity)

ecution time of any behavior b ($Tcons$), or on the bitrate of any port io ($Brcons$).

Given the AG and the constraint inputs, we wish to map the behaviors B_{all}, the variables V_{all}, and the channels C_{all} to a set of system components such that user-defined and system-capacity constraints are satisfied. A partition among system components is defined as a three-tuple $< I_{all}, M_{all}, P_{all} >$. I_{all} is a set of buses to which channels can be assigned, i.e., $I_{all} = \{i_1, i_2, ...\}$, where $i_k =< C, wires >$, $C \subset C_{all}$, $i_1.C \cup i_2.C \cup ... = C_{all}$, $i_j.C \cap i_k.C = \emptyset$ for all $j, k, j \neq k$, and where $wires \in Positive$. In other words, a bus i_k consists of a number of bus wires ($wires$), and of a set of channels (C), where every channel must be mapped to exactly one bus, (though each bus may have more than one channel). M_{all} is a set of memories to which variables can be assigned, i.e., $M_{all} = \{m_1, m_2, ...\}$, where $m_k =< V >$, $V \subset V_{all}$, $m_j.V \cap m_k.V = \emptyset$ for all $j, k, j \neq k$. P_{all} is a set of all standard or custom processors to which behaviors and variables can be assigned, i.e., $P_{all} = \{p_1, p_2, ...\}$, where $p_k =< BV >$, $BV \subset BV_{all}$, $p_1.BV \cup ...p_n.BV \cup m_1.V \cup ...m_o.V = BV_{all}$, $p_j.BV \cap p_k.BV = \emptyset$ for all $j, k, j \neq k$. In other words, a memory m_k consists of a set of variables (V), and a processor p_k consists of a set of behaviors and variables (BV). Every variable in V_{all} must be mapped to exactly one memory or a processor, and every behavior in B_{all} must be mapped to exactly one processor.

For example, Figure 4 shows a partition of several of the previous example's nodes among two memories, an ASIC, a processor and a bus. Note that two communication channels have been partitioned onto one 17-bit bus.

The AG and system component representations are actually part of a representation called the Specification-Level Intermediate Format (SLIF); for the complete definition of SLIF, the reader is referred to [6].

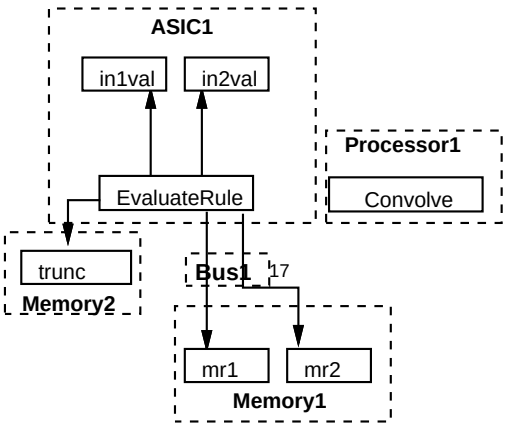

Figure 4: Partitioning AG nodes among system components

A partition should minimize the value of a given objective function:

$$Objfct(AG, Tcons, Brcons, P_{all}, M_{all}, I_{all})$$

This function returns 0 if all constraints are satisfied, otherwise it returns a number whose magnitude increases as violations occur; the number is referred to as the **cost** of a partition.

With the above definition of the functional partitioning problem, we can proceed to describe the closeness metrics that will help us create a minimal-cost partition for a given system.

3 Closeness metrics for the algorithmic level

In this section, we will define several closeness metrics between algorithmic-level functional objects. As we shall see in Section 4, these metrics can be used either to reduce the number of functional objects, to construct an initial partition, or to determine which objects to move in order to improve a given partition. However, we will first discuss an important related issue: normalization.

3.1 Normalization

Normalization refers to the mapping of a closeness metric value to a number between 0 and 1. There are three reasons why we wish to normalize metric values: (1) to combine values of different units, (2) to find natural clusters, and (3) to determine overall design significance.

To understand the problem of combining values of different units, consider defining closeness between behaviors as the sum of the number of shared wires and of shared transistors. Since there may be an order of magnitude more transistors in a design than

wires, a small change in the percentage of transistors will (undesirably) dominate the closeness value. For example, assume that two behaviors A and B share 10 wires and 10,000 transistors, and another two behaviors C and D share 100 wires and 9,900 transistors. If we simply add wire and transistor values without concern for the units, then $Closeness(A, B) = 10 + 10,000 = 10,010$ and $Closeness(C, D) = 100 + 9,900 = 10,000$. However, if 1 wire is much more important than 1 transistor, then we would prefer that the closeness of C and D be much higher than the closeness of A and B, since 90 wires would be more important than 100 transistors.

The second reason for normalization is to find the "natural" clusters of functional objects, where a natural cluster is a set of functional objects that share many wires, transistors, etc., with each other, but not with other objects. For example, if two behaviors A and B share 10,000 transistors, and their total size is 10,000 transistors (i.e., they share all their transistors), they should be considered extremely close, whereas if their total size were 100,000 transistors (i.e., they share only 10% of their transistors), then they shouldn't be treated nearly as close.

The third reason for normalization is to determine overall significance to the design. For example, suppose a designer has a two-way partition that he wishes to improve. He may ask for a list of closenesses between all pairs of objects that are in opposite groups, where closeness is based solely on shared wires. Suppose the pair A, B appears at the top of the list with a closeness of 10 wires. To decide if it would be worthwile to move those two objects into the same group, the designer must know how significant 10 wires is to the entire design. If there are only 20 wires in the entire design, then 10 is very significant; on the other hand, if there are 10,000 wires, then the designer might instead wish to concentrate on another metric.

We provide two normalization techniques. In the first technique, called **global normalization**, we divide each metric value by a number that computes that metric over the entire design. For example, we can divide the wires value by the total number of wires in the design, and divide the transistors value by the total number of transistors in the design. Suppose the total number of wires was 200, and the total number of transistors were 100,000. Dividing the metric values in the previous example would yield $Closeness(A, B) = \frac{10}{200} + \frac{10,000}{100,000} = .05 + .1 = .15$, and $Closeness(C, D) = \frac{100}{200} + \frac{9,900}{100,000} = .5 + .099 = .599$. C and D are much closer than without any normalization, as desired.

In the second normalization technique, called **local normalization**, we divide each metric value by a number that computes that metric over both objects involved. For example, rather than computing the number of wires in the entire design, as was done for global normalization, we can instead compute the number of wires that either A or B accesses, regardless of whether those wires are shared between A and B. Likewise, we can compute the total number of transistors to implement both A and B. In the previous example, suppose that for A and B the total wires were 12, and the total transistors were 20,000. Dividing metric values would yield $Closeness(A, B) = \frac{10}{12} + \frac{10,000}{20,000} = .83 + .5 = 1.33$, We see right away that, since almost all wires used by A and B are shared, the wire term

has a much larger effect on the overall closeness than when using global normalization.

In either type of normalization, we can divide the result by the number of metrics involved, in order to achieve a final closeness value between 0 and 1. A function that combines several closeness metric values into a single, normalized number is called a **closeness function**.

We now describe our closeness metrics, including local and global normalization factors for each.

3.2 Behavior closeness metrics

There are six closeness metrics that we have defined between any two sets of behaviors. **Connectivity** is based on the number of wires shared between the sets of behaviors. Grouping behaviors that share wires should result in fewer pins. **Communication** is based on the number of bits of data transferred between the sets of behaviors, independent of the number of wires used to transfer the data. Grouping heavily communicating behaviors should result in better performance, due to decreased communication time. **Hardware sharing** is based on the estimated percentage of hardware that can be shared between the two sets of behaviors. Grouping behaviors that can share hardware should result in a smaller overall hardware size. **Common accessors** is based on the number of behaviors that access both sets of behaviors. Grouping such behaviors should result in fewer overall wires. **Sequential execution** is based on the ability to execute behaviors sequentially without loss in performance. **Constrained communication** is based on the amount of communication between the sets of behaviors that contributes to each performance constraint. Grouping such behaviors should help ensure that performance constraints are met. **Balanced size** is based on the size of the sets of behaviors. Grouping smaller behaviors should eventually lead to groups of balanced size.

In the rest of this section, we will define each of the above closeness metrics between two sets of behaviors BV_i and BV_j. (Note that in this section, a variable is treated as a type of behavior, so when we say behaviors, we mean behaviors and variables). For each metric definition, we include a definition of local and global normalization values to normalize the metric value to a number between 0 and 1, as discussed in Section 3.1.

3.2.1 Connectivity

We start with the definition of the connectivity metric. This metric measures the estimated number of wires shared between two sets of behaviors. Wires appear between behaviors in the case when the behaviors' channels have been mapped to buses. However, we often wish to compute behavior closeness before having mapped channels to buses. In this case, we map each channel to its own bus by default, with a wire width equal to the channel's bits weight. Since we consider two possible cases, one in which channel to bus partitioning has been done, and the other when it hasn't, our approach is more general than previous approaches [7, 8] that consider only the second case.

We assume the existence of a procedure *AccessedBuses(bv)* that returns the set of buses connected with the given behavior or variable. Specifically, it returns the set of buses I for which $i.C$ (the channels mapped to the bus) contains at least one channel c in which $c.src = bv$ or $c.dst = bv$. The connectivity metric is then defined as follows:

$$ConnectivityMetric(BV_j, BV_k) = width\_common_{j,k}/norm\_factor \quad (1)$$

$$\begin{aligned}
width_common_{j,k} &= \textstyle\sum_{i \in I_j \cap I_k} i.wires, \\
I_j &= \textstyle\bigcup_{bv \in BV_j} \text{AccessedBuses}(bv), \\
I_k &= \textstyle\bigcup_{bv \in BV_k} \text{AccessedBuses}(bv). \\
norm_factor &= width_both_{j,k} \text{ for local normalization}, \\
&= width_all \text{ for global normalization}, \\
width_both_{j,k} &= \textstyle\sum_{i \in I_j \cup I_k} i.wires, \\
width_all &= \textstyle\sum_{i \in I} i.wires
\end{aligned}$$

In other words, I_j and I_k represent the set of buses connected to behaviors in BV_j and BV_k, respectively. $width\_common_{j,k}$ represents the total width of all buses connected to both BV_j and BV_k. $width\_both_{j,k}$ represents the total width of all buses connected to either BV_j or BV_k. $width\_all$ represents the total width of all buses in the system.

For example, consider computing the connectivity metric between the *EvaluateRule* and *trunc* nodes of Figure 3, assuming each channel is mapped to its own bus with a bitwidth equal to that channel's *bits* values. Then $width\_common$ would be 8, the width of the only bus between those two nodes. $width\_both$ would be $8 + 8 + 8 + 8 + 16 + 17 + 17 = 82$. $width\_all$ would be the sum of all the widths of all the edges, or 216. Therefore, the locally normalized metric value of would be $8/82 = .098$, while the globally normalized value would be $8/216 = .037$. Compare this with the closeness values between *EvaluateRule* and *FuzzyMain* of $8/122 = .066$ locally and $8/216 = .037$ globally. Note that because *trunc* is only connected with *EvaluateRule*, it leads to a much higher local closeness value than between *EvaluateRule* and *FuzzyMain*, as *FuzzyMain* is connected with many other nodes also. On the other hand, because both pairs are connected by the same number of bits, they result in the same global closeness value.

3.2.2 Communication

We now consider the communication closeness metric. This metric differs from the connectivity metric in that it measures the amount of data transferred between sets of behaviors, rather than the number of wires used to transfer that data. For example, if two behaviors communicate 16 bits of data 10 times over an 8 bit bus, then the communication metric would consider $10 \times 16 = 160$ bits, whereas the connectivity metric would consider only the 8 wires of the bus.

We assume the existence of a procedure *AccessedChannels(bv)* that returns the set of channels in which the given behavior or variable is the accessor or the accessee. Specifically, it returns the set of channels C such that c has $c.src = bv$ or $c.dst = bv$. The communication metric is then defined as follows:

$$CommunicationMetric(BV_j, BV_k) = bits\_common_{j,k}/norm\_factor \quad (2)$$

$$\begin{aligned}
bits_common_{j,k} &= \textstyle\sum_{c \in C_j \cap C_k} c.accfreq * c.bits, \\
C_j &= \textstyle\bigcup_{b \in BV_j} \text{AccessedChannels}(bv), \\
C_k &= \textstyle\bigcup_{b \in BV_k} \text{AccessedChannels}(bv). \\
norm_factor &= bits_both_{j,k} \text{ for local normalization,} \\
&= bits_all \text{ for global normalization,} \\
bits_both_{j,k} &= \textstyle\sum_{c \in C_j \cup C_k} c.accfreq * c.bits, \\
bits_all &= \textstyle\sum_{c \in C} c.accfreq * c.bits
\end{aligned}$$

In other words, C_j and C_k represent the set of channels accessed by behaviors in BV_j and BV_k, respectively. $bits\_common_{j,k}$ represents the total number of bits transferred between BV_j and BV_k. $bits\_both_{j,k}$ represents the total number of bits transferred between BV_j and any other behavior, or between BV_k and any other behavior. *bits_all* represents the total number of bits transferred over channels throughout the entire specification. (If a behavior contains an infinite loop, the behavior is ignored when computing this metric, since it's accessed channels will have access frequencies of infinity).

For example, consider computing the communication metric between the *EvaluateRule* and *trunc* nodes of Figure 3. Then *bits_common* would be $129 * 8 = 1032$. *bits_both* would be $2 * 8 + 8 + 8 + 129 * 8 + 129 * 16 + 65 * 17 + 65 * 17 = 5338$. *bits_all* would be the total bits transferred over all edges, or 17402. Therefore, the locally normalized metric value would be $1032/5338 = .193$, while the globally normalized value would be $1032/17402 = .059$. Compare this with the closeness values between *EvaluateRule* and *FuzzyMain* of $16/5338 = .003$ locally and $16/17402 = .001$. Note that because communication between *EvaluateRule* and *trunc* occurs much more frequently than between *EvaluateRule* and *FuzzyMain*, the former pair has much higher closeness values. Also note that the communication closeness of *EvaluateRule* and *trunc* is much higher than the connectivity closeness, making it clear that the number of wires between behaviors is not sufficient for predicting the amount of communication between those behaviors.

3.2.3 Hardware sharing

The hardware sharing metric is a measure of the amount of hardware that two sets of behaviors could possibly share between them. For example, if two behaviors both perform multiplication, then they may be able to share a single multiplier, thus reducing total hardware size than if they were implemented in separate groups, which would require two

multipliers. We assume the existence of a procedure *Size(BV)* that returns the hardware size for the given functional objects on a particular type of ASIC. This size may be computed through synthesis, through summation of abstract hardware weights associated with each object, or through more sophisticated techniques that use incremental synthesis techniques. If a particular type of ASIC is not specified (along with a set of available functional units and a maximum number of such each unit that can be used), then a default type is assumed. The hardware sharing metric is defined as follows:

$$HardwareSharingMetric(BV_j, BV_k) = size\_shared_{j,k}/norm\_factor \quad (3)$$

$size\_shared_{j,k}$	$= size_j + size_k - size_{j,k}$,
$size_{j,k}$	=Size($BV_j \bigcup BV_k$),
$size_j$	=Size(BV_j),
$size_k$	=Size(BV_k).
$norm\_factor$	$= size\_both\_min$ for local normalization,
	$= size\_all$ for global normalization,
$size\_both\_min$	=Min($size_j, size_k$),
$size\_all$	=Size(BV_{all})

In other words, $size_j$ and $size_k$ represent the hardware size to implement BV_j and BV_k, respectively. $size_{j,k}$ represents the size for a single implementation of both sets of behaviors BV_j and BV_k. $size\_shared_{j,k}$ represents the size of the hardware that would be shared between the two sets. $size\_both\_min$ is the minimum of $size_j$ and $size_k$, which is the maximum amount that could be shared between the two sets of behaviors. $size\_all$ is the size for a single implementation of all of the behaviors in the specification.

For example, consider computing the hardware sharing metric between the *EvaluateRule* and *Convolve* nodes of Figure 3. Suppose the size of *EvaluateRule* if implemented alone ($size_j$) were 8,000 gates, the size of *Convolve* ($size_k$) were 5,000 gates, the size of a combined implementation of both ($size_{j,k}$) were 10,000 gates (which is less than the sum of 8,000 and 5,000 due to sharing of datapath components such as an ALU), and the size of the entire design ($size\_all$) were 100,000 gates. Then $size\_shared$ would be $8000+5000-10000 = 3000$. $size\_both\_min$ would be 5000. Therefore, the locally normalized metric value would be $3000/5000 = .6$, while the globally normalized value would be $3000/100000 = .03$.

An analogous metric could be defined for software sharing, where size would be measured as the number of instructions, and sharing would involve the use of common subroutines.

3.2.4 Common accessors

When two sets of behaviors (and variables) are accessed via subroutine calls or variable reads/writes by many of the same behaviors, grouping those sets will likely reduce the number of wires and improve the performance of the design. We assume the existence of

a procedure *Accessors(bv)* that returns the set of behaviors that access the given behavior or variable. Specifically, it returns the set of behaviors B such that there exists a channel c for which $c.src = b$ and $c.dst = bv$, where $b \in B$. We also consider a behavior as its own accessor, in order to encourage grouping a variable or behavior with the behaviors that access it. The common accessors metric is defined as follows:

$$CommonAccessorsMetric(BV_j, BV_k) = |accessors\_common_{j,k}|/norm\_factor \quad (4)$$

$$\begin{array}{ll}
accessors_common_{j,k} & = accessors_j \bigcap accessors_k, \\
accessors_j & = \bigcup_{bv \in BV_j} \text{Accessors}(bv), \\
accessors_k & = \bigcup_{bv \in BV_k} \text{Accessors}(bv), \\
norm_factor & = |accessors_both_{j,k}| \text{ for local normalization}, \\
 & = |accessors_all| \text{ for global normalization}, \\
accessors_both_{j,k} & = accessors_j \bigcup accessors_k, \\
accessors_all & = B_{all}
\end{array}$$

In other words, $accessors_j$ and $accessors_k$ represent the set of behaviors that access at least one behavior/variable in BV_j and BV_k, respectively. $accessors\_common_{j,k}$ represents the set of behaviors that access at least one behavior/variable in each of BV_j and BV_k. $accessors\_both_{j,k}$ represents the set of behaviors that access at least one behavior/variable in either B_j or B_k. *accessors_all* represents all possible accessor behaviors, which is simply all behaviors in the entire specification. (The notation $|accessors|$ represents the number of elements in the set *accessors*).

For example, consider computing the common accessors metric between the *in1val* and *in2val* nodes of Figure 3. Then the number of common accessors would be 2 (*FuzzyMain* and *EvaluateRule*). The number of accessors of both nodes would also be 2. The number of all possible accessors would be the total number of nodes, or 12. Therefore, the locally normalized metric value would be $2/2 = 1.0$, while the globally normalized value would be $2/12 = .167$. Compare this with the closeness values between *in1val* and *mr1* of $1/3 = .333$ locally and $1/12 = .083$ globally. Although *in1val* and *mr1* have a non-zero closeness because of their common access by *EvaluateRule*, *in1val* and *in2val* have a much higher closeness because they are accessed by exactly the same two nodes.

3.2.5 Sequential execution

If two behaviors are defined sequentially in the specification, they are less likely to reduce performance when mapped to a single processor than are two concurrent behaviors. We assume the existence of a function *SequentialBehs(b1,b2)* that returns 0 if behaviors $b1$ and $b2$ could execute concurrently, else it returns 1. The sequential execution metric is defined as follows:

$$SequentialExecutionMetric(B_j, B_k) = seq\_pairs_{j,k}/norm\_factor \quad (5)$$

$seq\_pairs_{j,k}$ $= \sum_{b1 \in B_j, b2 \in B_k}$ SequentialBehs$(b1, b2)$,
$norm\_factor$ $= |B_j| \times |B_k|$ for local normalization,
$= \frac{|B_{all}| \times (|B_{all}|-1)}{2}$ for global normalization

In other words, $seq\_pairs_{j,k}$ equals the number of pairs of behaviors that can execute sequentially, where a pair consists of one behavior from B_j and one from B_k. The local normalization factor is the total number of possible pairs between those two behavior sets, while the global normalization factor is the total number of possible pairs of behaviors from the entire specification.

In the example of Figure 3, all pairs of behaviors have a closeness value of 0 for this metric, since they must execute sequentially.

3.2.6 Constrained communication

When we are given a set of performance constraints on behaviors, it may be better to concentrate on the communications that affect those constrained behaviors, rather than on all communications as was done in the above communication metric. Grouping heavily communicating behaviors will reduce inter-group communication delay, and will thus make it more likely that we will meet performance constraints. Therefore, when deciding whether to group two sets of behaviors, we look not only at the amount of communication between those sets, but also at how much this communication affects the given performance constraints. We assume the existence of a procedure *AccessedChansRecur(b)* that returns all channels accessed by behavior b, and all channels accessed by any accessee of b, and so on. The constrained communication metric is then defined as follows:

$$ConstrainedCommunication(BV_j, BV_k) = bits\_common_{j,k}/norm\_factor \quad (6)$$

$bits\_common_{j,k}$ $= \sum_{t \in Tcons} \text{ConstrCommBehs}(t.b, BV_j, BV_k)$,

$\text{ConstrCommBehs}(t.b, BV_j, BV_k) = \sum_{c \in ContribChansRecur(t.b)} c.accfreq * c.bits$,

$\text{ContribChansRecur}(t.b) = \bigcup_{c \in AccessedChansRecur(t.b)} contrib\_chan_c$,

$contrib\_chan_c$ $= c$ if $c.src \in BV_j$ and $c.dst \in BV_k$,
$= c$ if $c.src \in BV_k$ and $c.dst \in BV_j$,
$= \emptyset$ otherwise,

$norm\_factor$ $= \sum_{t \in Tcons} \text{ConstrComm}(t.b, BV_j, BV_k)$
for local normalization,
$= \sum_{t \in Tcons} \text{ConstrComm}(t.b, BV_{all}, BV_{all})$
for global normalization,

$\text{ConstrComm}(t.b, BV_x, BV_y)$ $= \sum_{c \in AccessedChansRecur(t.b)} c.accfreq * c.bits$ if
$\text{ConstrCommBehs}(t.b, BV_x, BV_y) > 0$,
$= 0$ otherwise

In other words, $ContribChansRecur(t.b)$ is the set of all channels that involve a communication between a behavior in BV_j and one in BV_k, where that communication contributes to the performance of constrained behavior $t.b$. $ConstrCommBehs(t.b, BV_j, BV_k)$ is the total number of bits transferred over those channels during an execution of $t.b$, and $bits\_common_{j,k}$ is the sum of this number for each constrained behavior in $Tcons$. The local normalization factor is the number of bits communicated between any two behaviors during the execution of constrained behaviors in which at least one communication occurs between BV_j and BV_k. The global normalization factor is the number of bits communicated between any two behaviors during execution of all constrained behaviors.

For example, assume that a constraint t_c exists on the execution time of *Convolve*, and another constraint t_e exists on the execution time of *EvaluateRule*. Consider computing the constrained communication metric between *EvaluateRule* and *trunc*. Then $ContribChansRecur(EvaluateRule)$ would consist only of the channel between *EvaluateRule* and *trunc*. $ConstrCommBehs(t_e)$ would then be $129 * 8 = 1032$. On the other hand, $ContribChansRecur(Convolve)$ would be empty, because no channels between *EvaluateRule* and *trunc* affect the execution time of *Convolve*, so $ConstrCommBehs(t_c)$ would be 0. Continuing, $bits\_common_{j,k}$ would be $1032+0 = 1032$. The local normalization factor would be the number of bits transferred during execution of *EvaluateRule* (since the execution of *EvaluateRule* is affected by the channel between the two nodes for which we are computing the closeness), or $8+8+129*8+129*16+65*17+65*17 = 5322$. The bits transferred during execution of *Convolve* are not considered because no channel between *EvaluateRule* and *trunc* affects this execution (i.e., $ConstrCommBehs(t_c, EvaluateRule, trunc) = 0$). The global normalization factor would be the number of bits transferred during execution of *EvaluateRule*, or 5322, plus the number transferred during execution of *Convolve*, or 8192, totalling 13514. Therefore, the locally normalized metric value would be $1032/5322 = .194$, while globally normalized value would be $1032/13514 = .076$.

Note that if a constraint were imposed on *FuzzyMain*, then all channels would affect that constraint, even those at the bottom of Figure 3.

3.2.7 Balanced size

When clustering behaviors in order to obtain a partition among ASICs, we usually want a final partition that consists of groups that are roughly balanced in hardware size. If we don't make an effort to balance group size, the above metrics will end up clustering nearly all the objects in one group. One way to encourage balanced sizes is to favor merging smaller behaviors over larger ones, to prevent any one group from getting too large.

$$BalancedSizeMetric(BV_j, BV_k) = (size\_all - size_{j,k})/norm\_factor \quad (7)$$

$size_{j,k}$	=Size($BV_j \bigcup BV_k$),
$norm\_factor$	= $size\_all$ for both local and global normalization,
$size\_all$	=Size(BV)

$size_{j,k}$ represents the size for a single implementation of both sets of behaviors BV_j and BV_k. $size\_all$ is the size for a single implementation of all of the behaviors in the specification. Note that the metric numerator is $(size\_all - size_{j,k})$, since the smaller that $size_{j,k}$ is, the *larger* should be the metric value. Also note that there is no real way to locally normalize in this case, so we use the global normalization factor for both normalization types.

Since this metric is straightforward, we omit an example.

3.3 Variable closeness metrics

There are three closeness metrics that can be used to group variables for implementation in a memory. **Common accessors** is based on the number of behaviors that access both sets of variables. Grouping such variables should result in fewer overall wires. **Sequential access** is based on the occurrence of sequential, rather than concurrent, access of the variables by behaviors. Grouping sequentially-accessed variables into the same memory does not decrease performance, whereas grouping concurrently-accessed variables might decrease performance due to access conflicts. **Width similarity** is based on the similarity of the variables' bitwidths. Grouping variables with similar bitwidths should result in fewer wasted memory bits.

3.3.1 Common accessors

When two sets of variables are accessed by many of the same behaviors, grouping those variables into a memory will likely reduce the number of wires and memory access conflicts.

The definition of the function that computes this metric is identical to the common accessor metric in Section 3.2, with the exception that only variables are passed as parameters to the function. We therefore refer to that section for the definition of this metric.

3.3.2 Sequential access

When two sets of variables are accessed sequentially, grouping those variables into a memory is not likely to decrease performance, since there will not be contention for access to the memory. Thus we prefer to group sequentially accessed variables over concurrently accessed ones. We assume the existence of a procedure *SequentialAccessedVars(v1,v2)* that returns 0 if variables $v1$ and $v2$ could execute concurrently, else it returns 1. The definition of this metric is identical to that of the sequential execution metric in Section 3.2, except that only variables are passed as parameters, and the procedure *SequentialBehs(b1,b2)* is replaced by *SequentialAccessedVars(v1,v2)*. We therefore refer to that section for the definition of this metric.

3.3.3 Width similarity

When two sets of variables would be encoded into the same bitwidth, then we would expect that grouping those variables into the same memory would yield better use of the available memory bits. For example, 10 16-bit variables can use all the bits of a 10x16 memory, whereas 5 16-bit variables and 5 12-bit variables would result in 20 wasted bits in the same memory. We choose to define this metric in terms of the number of pairs of variables between the two sets that have similar widths. We assume the existence of a procedure *VariableWidth(v)*, which computes the number of bits for a scalar variable v, or for a scalar element of an array variable v, in the same manner as described in Section 2 for channels. The width similarity metric is defined as follows:

$$WidthSimilarityMetric(V_j, V_k) = pairs\_sim_{j,k} / norm\_factor \tag{8}$$

$$\begin{aligned}
pairs_sim_{j,k} &= \textstyle\sum_{u \in V_j, v \in V_k} pair_sim_{u,v}, \\
pair_sim_{u,v} &= pair_sim_frac_{u,v} \text{ if } \leq 1, \\
&= 1 \div pair_sim_frac_{u,v} \text{ otherwise,} \\
pair_sim_frac_{u,v} &\text{ VariableWidth}(u) \div \text{VariableWidth}(v), \\
norm_factor &= |V_j| \times |V_k| \text{ for local normalization,} \\
&= \tfrac{|V_{all}| \times (|V_{all}| - 1)}{2} \text{ for global normalization}
\end{aligned}$$

In other words, $pair\_sim\_frac_{u,v}$ represents the ratio of the widths of variables u and v. $pair\_sim_{u,v}$ is this ratio or its inverse, depending on whichever makes the value less than or equal to 1; this number thus indicates the similarity in widths of the two variables. $pair\_sim_{u,v}$ will therefore equal 1 when the pair has identical widths; otherwise,

the more disimilar the widths, the closer will $pair\_sim_{u,v}$ be to 0. $pairs\_sim_{j,k}$ is the sum of the $pair\_sim_{u,v}$ values for all possible pairs of variables, where a pair consists of one variable from set V_j and another from V_k. The local normalization factor is the maximum value of $pairs\_sim_{j,k}$ for the case where all variables have the same width (i.e., each pair has a $pair\_sim_{u,v}$ value of 1), which is simply the number of possible pairs. The global normalization factor is the number of possible pairs of all variables in the specification.

Because the example of Figure 3 has variables of identical width, it does make an interesting example for this metric. Thus, let us consider a different example consisting of 10 variables. Consider computing the width similarity metric between a pair of variables a (7 bits), b (8 bits), and another pair of variables c (8 bits), d (4 bits). Then $pair\_sim_{a,c} = 7 \div 8 = .875$, $pair\_sim_{a,d} = 1 \div (7 \div 4) = .571$, $pair\_sim_{b,c} = 8 \div 8 = 1.0$, and $pair\_sim_{b,d} = 1 \div (8 \div 4) = .5$. $pairs\_sim$ would equal the sum of those values, or 2.946. The local normalization factor would be $2 * 2 = 4$, while the global factor would be $\frac{10*9}{2} = 45$. Thus the local closeness value would be $2.946/4 = .737$, while the global closeness value would be $2.946/45 = .065$. Note that if d were nearer to 8 bits, the closeness would be much higher. Also note that if all variables were 8 bits, the local closeness would be 1.0.

3.4 Channel closeness metrics

There are three closeness metrics that can be used to group channels for implementation on a bus. **Common accessors** is based on the number of behaviors that access both sets of channels. Grouping such channels results in fewer overall wires. **Sequential access** is based on the occurrence of sequential, rather than concurrent, access of the channels by behaviors. Grouping sequentially-accessed channels into the same bus does not decrease performance, whereas grouping concurrently-accessed channels might decrease performance due to access conflicts. The merging of sequentially-accessed channels is also discussed in [9]. **Width similarity** is based on the similarity in bitwidths of the channels. Grouping channels with similar bitwidths results in fewer wasted bus wires during transfers.

Since these metrics are defined almost identically to the variable closeness metrics, we omit their detailed definition here.

4 Experiments

The above closeness metrics may be used during specification partitioning. In this section, we describe how they were used in several experiments, and highlight their impact.

4.1 Merging objects to reduce complexity

The previously-defined closeness metrics can be used to merge several close functional objects into a single new object. Such merging reduces the number of functional objects that must be partitioned for implementation among system components, therefore reducing the

computation time and memory requirements of automated partitioning techniques (such as group migration, simulated annealing, and integer linear programming). In other words, objects that should never be separated are merged before partitioning, therefore "pruning away" a large number of inferior solutions.

To demonstrate the usefulness of our closeness metrics in merging functional objects, we experimented with three examples: an ethernet controller (ether) [10], and an interactive TV processor system (itv) [11], and a microwave-transmitter controller (mwt). The ether example contained 1021 lines of VHDL behavior consisting of 125 functional objects, while the itv example contained 611 lines with 85 functional objects, and the mwt example contained 720 lines with 25 functional objects.

For each example, we reduced the number of functional objects by performing an initial merging of objects. The algorithm used for merging was hierarchical clustering [7], using a closeness function that was a locally-normalized weighted-sum of the *Connectivity*, *Constrained communication*, *Hardware sharing* and *Sequential execution* closeness metrics. After reducing the number of objects, we applied the group migration [12] algorithm (with a random initial two-way partition of the objects) to partition the objects for implementation on two VTI ASIC components, using a cost function that sought to minimize violation of ASIC size and pin constraints and of average execution-time constraints on numerous behaviors.

Figure 5 displays results for the ether example. The *number of objects* axis represents the number of objects after clustering. The *cost* axis represents the magnitude of constraint violations of the final partition after group migration was applied to the merged functional objects, where the cost was determined using the cost function described earlier. The *time* axis represents the computation time required by the group migration algorithm to find the best partition given the merged objects. Note that in general, the fewer the number of objects input to the algorithm, the faster the computation time (the fluctuations in computation time are due to the nature of the group migration algorithm, which may iterate anywhere between 1 to 4 times for a given example; we found similar fluctuations using simulated annealing). For this example, when we reduced the number of objects from 125 down to 85, we not only reduced the computation time by 30%, but also obtained a lower-cost final partition. A reduction down to 55 objects reduced the computation time by nearly 55%, at the expense of a slightly higher-cost partition. Any further reduction results in a much higher-cost partition, because the clustering process is forced to merge objects that are not very close. (Note that we intentionally formulated the constraints such that no partition would satisfy all constraints, so that we could compare partitions; otherwise, partitions satisfying all constraints would all have zero cost, so their quality could not be compared using our cost function).

The initial time spent merging the objects ranged from 650 seconds when reducing the number of objects down to 120, to 900 seconds when reducing the number of objects down to 5 objects. The initial merging time is neglible when we consider that after performing the initial merging once, a designer may go on to examine tens or hundreds of possible

combinations of system components. For example, he may try a two ASIC implementation, then a three ASIC implementation, then a one ASIC and one microcontroller implementation, and so on. For each combination of system components, he might reapply the group migration algorithm, but need not reapply the initial merging. Therefore, the time spent performing the initial clustering becomes negligible, and the time saved in group migration becomes very substantial.

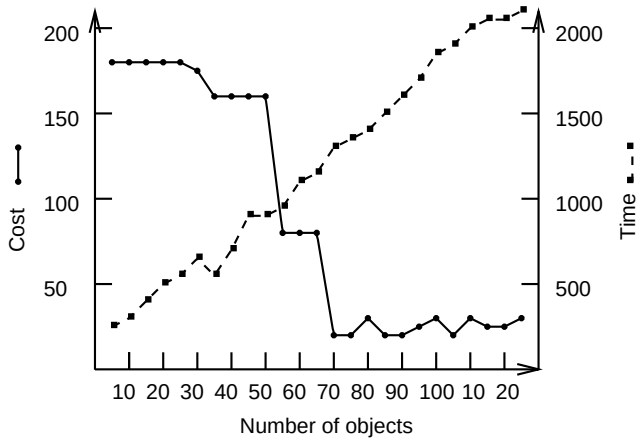

Figure 5: Cost vs. time tradeoffs achieved by merging objects before partitioning for the ether example.

Figure 6 summarizes results for the ether, itv and mwt examples. Each table entry is of the form *cost, CPU time*. For all the three examples, it appears that we can reduce the number of objects by 25% to achieve a roughly 25% reduction in CPU time without higher partition cost, and that we can reduce the number of objects by just less than 50% to achieve a 50% reduction in CPU time with a small increase in cost. Such results are intuitive because specifications are usually written in a modular manner, and modularity implies that there are clusters of interacting functional objects, as opposed to the case where every functional object interacts with every other object. The initial merging groups these interacting objects, thus freeing the subsequent group migration algorithm from the job of determining that they belong on the same component. We hope in future work to perform similar experiments to determine if these results can be generalized to a large class of examples.

We omit a detailed comparison of the results obtained using different combinations of closeness metrics, since our purpose here is to demonstrate that the metrics can be useful, rather than describing which combination of metrics are useful for which classes of examples. We plan to describe results of an extensive comparison of metrics, including a comparison of normalization techniques, in a future publication. However, we will note that our experiments so far indicate that the constrained communication metric is superior to the communication metric when execution-time constraints are given. Also, we have found

Number of objects	Example		
	ether	itv	mwt
5	180, 230	185, 120	135, 24
10	180, 310	185, 160	110, 28
15	180, 400	185, 160	15, 32
20	180, 475	175, 260	10, 35
25	180, 555	175, 305	10, 64
30	175, 640	175, 360	
35	160, 550	85, 595	
40	160, 700	85, 445	
45	160, 870	85, 960	
50	160, 900	85, 580	
55	80, 965	55, 620	
60	80, 1110	60, 1050	
65	80, 1165	60, 1090	
70	50, 1280	55, 870	
75	30, 1350	50, 1700	
80	30, 1420	60, 980	
85	20, 1485		
90	20, 1615		
95	25, 1700		
100	30, 1840		
105	20, 1880		
110	30, 1970		
115	25, 2030		
120	25, 2030		
125	30, 2100		

Figure 6: Results of merging objects before partitioning for three examples

that when branch probabilities are known (meaning that the communication metrics will be very useful), then the common accessors metric does not contribute to a better partition. Instead, it is more useful when branch metrics are not known, since it provides an indirect measure of communication. Finally, we have found that the constrained communication metric is very useful for hardware/software partitioning, since the metric can be used to group unconstrained (or loosely constrained) behaviors for implementation in the software component, and heavily constrained behaviors for implementation in the hardware component.

4.2 Partitioning among system components

In the previous section, we indicated that methods such as group migration, simulated annealing and ILP could be used to obtain a final partition of functional objects among system components. In that case, clustering was simply used as a "preprocessing" step to reduce the number of objects. In this section, we use clustering itself as the method of obtaining a final partition. In this method, the objects are merged until the number of groups equals the number of system components. As in the previous section, closeness

metrics are combined in a closeness function used by a hierarchical clustering algorithm, but here care must be taken to maintain balanced groups so that component size constraints are not violated.

We created two-ASIC partitions for each of the above three examples, using the globally normalized closeness metrics of *Connectivity*, *Constrained communication*, *Sequential execution* and *Balanced size* (without the balanced size metric, the clustering placed all objects but one in the same group). We used a reduced set of objects for the ether and itv examples. Figure 7 summarizes the results. For each example, we indicate the size and pins of each ASIC after implementing the partitioned functional objects. We also show in the *perf* column the execution times of constrained behaviors for each example. The *time* and *cost* columns represent the CPU time for the algorithm on a Sparc 2, and the final cost (representing the magnitude of constraint violations as discussed in Section 2). Our initial results show that clustering using these closeness metrics yields high-quality final partitions in reasonable time. For comparison purpose, we also generated two-ASIC partitions using group migration. Final costs obtained using group migration are also shown in the figure. While these costs are slightly better than obtained using clustering, the CPU time varied greatly for group migration. Group migration executed 3 times faster than clustering for the first example, 15 times slower for the second example, and about the same for the third example. We can see that no one approach to partitioning gives the best cost or CPU time for every example. We conclude from our experiments that clustering using our closeness metrics is one of several viable approaches to partitioning functional objects among system components.

Example	Alg	asic1	asic2	perf	time	cost
ether	clust	31200,126	16800,84	22	235	30
	gm	22100,103	24200,69	24	75	3
itv	clust	132800,141	98300,28	165,55,16	62	65
	gm	130300,135	101400,74	165,55,16	980	62
mwt	clust	9400,74	4600,38	372/49	60	84
	gm	6405,57	6700,74	276,57	65	10

Figure 7: Results of two-ASIC partitioning by clustering.

4.3 Guiding manual partitioning

A third common use of closeness metrics is to guide a designer during manual partitioning of functional objects among system components. Given a current partition, the designer may request information to guide her in deciding which objects to move from one component to another. To obtain such information, she can indicate a "seed" object or group of objects along with one or more closeness metrics; the closenesses of all other objects to that seed can then be generated automatically. Based on these closenesses, he might then decide

to move the closest object to the same component as the seed. For example, Figure 8 provides a simple example in which functional objects *A*-*H* are currently partitioned among a processor and ASIC. The designer wishes to improve the system's performance by moving an object from software to hardware. The designer can request closenesses of all objects in the processor to the set of objects in the ASIC, any he may then decide to move the closest object *B* to the ASIC. There are obviously many variations of the above closeness request example.

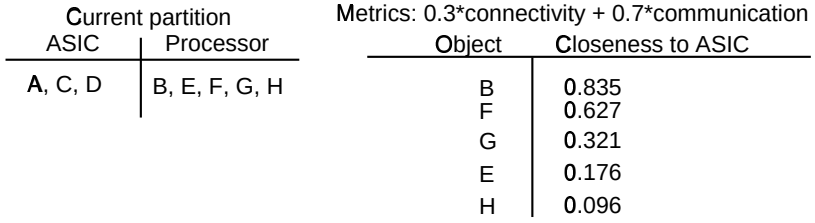

Figure 8: Using closenesses during manual partitioning.

5 Related research

Although this is the first work that we know of that defines closeness metrics among algorithmic-level functional objects, there are several related research efforts on lower levels of granularity. In [13], closeness metrics are defined to cluster logic operations among blocks of logic. In [7], closeness metrics are defined to cluster arithmetic operations among datapath modules, with several extensions described in [8]. Our metrics of hardware sharability, sequential execution, and connnectivity are similar in purpose to those defined in those other efforts, with our metrics being defined for much higher-level functional objects. In [14], closeness metrics are defined between assignment statements to cluster those statements into hardware and software groups.

Several other system-level partitioning efforts address issues different from that of closeness metric definition. In [5] and [15], arithmetic-level operations are partitioned among ASICs using iterative-improvement algorithms. In [16] and [17], specification pieces are partitioned among a simple hardware/software architecture, again using iterative-improvement algorithms. In [18] and [19], processes and procedures of a specification are partitioned among ASICs using iterative improvement or genetic algorithms. In [20] and [21], ILP formulations are presented for simultaneously scheduling arithmetic operations and partitioning them among ASICs. In [22], an ILP formulation is presented that selects a set of processors on which to implement a set of performance-constrained processes, and to partition the processes among those processors.

Some other related works include that in [23], which provides techniques for rapid size and time estimation for a partition of arithmetic operations among ASICs. Research pre-

sented in [1] and [24] overviews the hardware/software partitioning problem, including discussions on granularity, estimation, and simulation of interacting hardware and software components. A survey of a variety of research efforts in the field of hardware/software codesign appears in [25].

The most important difference between our work and previous work is the higher-level of granularity, which we have found leads to an order of magnitude reduction of functional objects, and hence to tractable computation and to designer comprehension. In addition, our work differs from previous efforts by covering the partitioning of all *three* aspects of functionality, namely behaviors, variables, and communications, rather than focusing on behaviors (e.g., arithmetic-level operations) alone as in most previous work on closeness metrics. One last difference is that we discuss both global and local normalization, which is not covered in previous works.

6 Conclusion

We have defined several new closeness metrics between algorithmic-level functional objects. Partitioning at this level of granularity enables us to partition much larger systems than possible with previous efforts. More importantly, it encourages designer interaction, especially when we consider that current manual partitioning is performed at that level of abstraction, meaning that our approach fits well with the current design methodology. Supporting and encouraging designer interaction is crucial for the acceptance and practical use of any system-level design tool, and partitioning at the algorithmic level achieves this goal.

We have demonstrated that these closeness metrics can be used by clustering algorithms to substantially reduce the computation time of iterative-improvement partitioning algorithms without a loss in partition quality. Such a reduction of computation time greatly increases the usefulness of automated algorithms during partitioning. We have also shown that the metrics can be used to create high-quality final partitions, offering an alternative or complement to iterative-improvement algorithms. In addition, we have described how the closeness metrics can be used to guide manual partitioning. Such guidance can greatly simplify the system designer's task and substantially reduce the amount of hand-calculation necessary. In summary, the closeness metrics can be used to greatly enhance the rapidly growing number of functional-level system-design tools.

References

[1] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[2] A. Orailoglu and D. Gajski, "Flow graph representation," in *Proceedings of the Design Automation Conference*, pp. 503–509, 1986.

[3] G. DeMicheli and D. Ku, "HERCULES - a system for high-level synthesis," in *Proceedings of the Design Automation Conference*, 1988.

[4] D. Thomas, E. Langese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, 1990.

[5] R. Gupta and G. DeMicheli, "Partitioning of functional models of synchronous digital systems," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 216–219, 1990.

[6] F. Vahid, "SLIF: A specification-level intermediate format for system design." UC Riverside, CS Dept., Technical Report 94-5,1994.

[7] M. McFarland and T. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on Computer-Aided Design*, September 1990.

[8] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, July 1991.

[9] D. Filo, D. Ku, C. Coelho, and G. DeMicheli, "Interface optimization for concurrent systems under timing constraints," in *IEEE Transactions on Very Large Scale Integration Systems*, pp. 268–281, September 1993.

[10] R. Gupta and G. DeMicheli, "System-level synthesis using re-programmable components," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 2–7, 1992.

[11] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems.* New Jersey: Prentice Hall, 1994.

[12] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, February 1970.

[13] R. Camposano and R. Brayton, "Partitioning before logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[14] X. Xiong, E. Barros, and W. Rosentiel, "A method for partitioning UNITY language in hardware and software," in *Proceedings of the European Design Automation Conference (EuroDAC)*, 1994.

[15] Z. Peng and K. Kuchcinski, "An algorithm for partitioning of application specific systems," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 316–321, 1993.

[16] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test of Computers*, pp. 29–41, October 1993.

[17] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test of Computers*, pp. 64–75, December 1994.

[18] F. Vahid and D. Gajski, "Specification partitioning for system design," in *Proceedings of the Design Automation Conference*, 1992.

[19] N. Kumar, R. Vemuri, and R. Vemuri, "Partitioning for multicomponent synthesis from VHDL specifications," in *VHDL International Users' Forum*, pp. 19–28, 1993.

[20] C. Gebotys, "An optimization approach to the synthesis of multichip architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, no. 1, pp. 11–20, 1994.

[21] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," in *IEEE Transactions on Very Large Scale Integration Systems*, pp. 21–32, 1994.

[22] S.Prakash and A. Parker, "Synthesis of application-specific multiprocessor architectures," in *Proceedings of the Design Automation Conference*, pp. 8–13, 1991.

[23] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proceedings of the Design Automation Conference*, 1991.

[24] A. Kalavade and E. Lee, "A hardware/software codesign methodology for DSP applications," in *IEEE Design & Test of Computers*, 1993.

[25] W. Wolf, "Hardware-software co-design of embedded systems," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, 1994.