# Closeness metrics for system-level functional partitioning

Frank Vahid
Department of Computer Science
University of California, Riverside, CA 92521

Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA 92717

## Abstract

*An important system design task is the partitioning of system functionality for implementation among 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. We summarize experiments for determining the best combination of metrics for particular uses, and we demonstrate the advantages of clustering with the closeness metrics before applying hardware or hardware/software partitioning.*

## 1 Introduction

A system's functionality must be implemented on system components, such as standard or custom processors, 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 an ASIC may contain a processor core and numerous memories and custom processor blocks. Time-critical behaviors such as data capture may be implemented using custom hardware blocks, while less critical behaviors, such as 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 partition the system's functionality for implementation among the various system components such that design constraints are satisfied. Although there are an enormous number of possible partitions to choose from, only a small number of possibilities are explored in current practice. In current practice, the partitioning task is vaguely defined, decisions are based on mental or hand-calculated estimations, and documentation of decisions is scarce. These shortcomings have led researchers to propose starting from a simulatable functional specification, whose pieces are then partitioned among system components with the aid of tools.

In this paper, we define closeness metrics for functionally partitioning a specification among hardware or software components. A **closeness metric** measures the likelihood that two pieces of the specification should be implemented on 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 used for two purposes: pre-assignment clustering or N-way clustering. In **pre-assignment clustering**, we merge together a few very close pieces of the specification. Then, when we subsequently apply partitioning heuristics to assign those pieces to system components, the heuristics deal with fewer objects, leading to reduced runtimes and possibly to better results. In **N-way clustering**, we group close objects until there are only N groups remaining, where each group is then assigned to its own system component. N-way clustering is a viable alternative to other N-way partitioning heuristics, and is especially useful 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 for those metrics don't exist.

Previous research into closeness metrics has been focused on fine-granularity objects: logic operations in [1], arithmetic operations in [2], and statements in [3]. To our knowledge, our metrics are the first to address objects at the procedural level. Such coarse granularity supports manageable complexity and designer interaction. The case for procedural-level granularity during system partitioning has been presented in [4, 5, 6, 7], although those efforts did not address the issue of closeness metrics. Other functional partitioning efforts include operation-level approaches in [8, 9, 10, 11], statement-sequence level approaches in [12, 13], and state-level approaches in [14, 15]. Our closeness metrics can be used in conjunction with the procedural, statement-sequence, or state-level approaches.

The paper is organized as follows. In Section 2, we provide a problem definition. In Section 3, we discuss normalization techniques. In Section 4, we define closeness metrics among behaviors. In Section 5, we describe results of experiments to determine the best combination of metrics, and to demonstrate the advantages of using the closeness metrics. In Section 6, we provide conclusions.

## 2 Problem definition

A major system design problem is the partitioning of functional objects among system components. There are three types of **functional objects** in a specification: **Variables** store data, **behaviors** transform data, and **channels** transfer data between behaviors. We consider behaviors at a process/procedural level of granularity, as opposed to the finer-grained statement or arithmetic-operation level, since coarse granularity is needed for manageable complexity and designer interaction when dealing with large systems. There are also three types of system components: **memories**, such as RAMs, ROMs, register-files, and registers, store scalar and array variables; standard and custom **processors** implement behaviors and variables; **buses** implement communication channels. Hence, there are three partitioning problems to be solved: behaviors/variables to processors, variables to memories, and channels to buses. This paper focuses on the first problem.

We read a functional specification into a directed-graph intermediate format called an **access graph**, or AG. The AG represents the accesses, rather than dependencies, be-

tween behaviors and variables. An access is a procedure call, a variable/port read or write, or a message pass. The AG's orientation around accesses, its coarse granularity, and its estimation annotations make it particularly well-suited for system-level partitioning. For details on the AG and the more general specification-level intermediate format SLIF, we refer the reader to [16]. We include a partial AG definition in Figure 1 for use in our closeness metric equations.

| Item | Definition |
|---|---|
| AG | $< IO_{all}, BV_{all}, C_{all} >$ (Functional objects) |
| $IO_{all}$ | $\{io_1, io_2, ...\}$ (Input/output ports) |
| $BV_{all}$ | $B_{all} \cup V_{all}$ |
| $B_{all}$ | $\{b_1, b_2, ...\}$ (**Behaviors**) |
| $V_{all}$ | $\{v_1, v_2, ...\}$ (**Variables**) |
| $C_{all}$ | $\{c_1, c_2, ...\}$ (**Channels**) |
| $c_i$ | $< src, dst, accfreq, bits >$ |
| $src$ | $\in B_{all}$ (Accessor behavior) |
| $dst$ | $\in BV_{all} \cup IO_{all}$ (Accessee object) |

**Fig. 1**: Access-graph definition

Each communication channel $c_i$ 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. (Minimum and maximum accfreq weights can also be associated with each edge). 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 any parameters. For a message pass, this is the number of bits into which the message would be encoded.

# 3 Normalization

Before defining the closeness metrics, we first discuss normalization of metric values. Normalization refers to the scaling of metric values to a number between 0 and 1. There are two reasons why we wish to normalize metric values. First, normalized values provide an absolute measure of the closeness of objects, e.g., we do not know if 10 shared wires between two objects represents a high closeness, whereas we know that a normalized value of .9 is very high. Second, normalization provides a means for combining values of different units, such as transistors and seconds, into a single value.

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 a shared-wires value between two objects by the total number of wires in the design. In the second technique, called **local normalization**, we divide each metric value by a number that computes that metric only over both objects involved, rather than over the entire design. For example, we can divide the shared-wires value by the number of wires accessed by either of the involved objects.

In either type of normalization, we add the normalized values of all metrics, and then divide the sum 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 techniques for each.

# 4 Behavior closeness metrics

We have defined seven closeness metric between two sets of procedural-level behaviors and variables, $BV_i$ and $BV_j$. **Connectivity**, **shared hardware** and **sequential execution** are common metrics also used for finer-grained behaviors [1, 17, 2]; our definitions differ from previous ones since they are for procedural-level behaviors, and since they include the two types of normalization. The **communication** metric considers the amount of data transferred during execution, which hasn't been addressed directly by previous metrics. **Constrained communication** is an even more powerful metric that considers both the communication and the provided performance constraints. In case communication metrics can't be computed due to a lack of appropriate estimators, we've defined the **common accessors** metric as an indirect measure. Finally, our **balanced size** metric ensures that no single cluster grows too large.

## 4.1 Connectivity

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. If channels have not yet been mapped to buses, we map each channel to its own bus, with a wire width equal to the channel's bits weight.

We assume a procedure *AccessedBuses(bv)* exists that returns the set of buses connected with the given behavior or variable. Specifically, it returns the set of buses $I = \{i_1, i_2, ...\}$ for which $i.C$ (the set of 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:

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

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

In other words, $I_j$ and $I_k$ represent the set of buses connected to any behaviors in $BV_j$ and $BV_k$, respectively. $width\_cmn_{j,k}$ 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.

### 4.2 Communication

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 $16 \times 10 = 160$ bits, whereas connectivity would consider only the 8 wires of the bus.

We assume a procedure *AccessedChannels(bv)* exists 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:

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

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

In other words, $C_j$ and $C_k$ represent the set of channels accessed by behaviors in $BV_j$ and $BV_k$, respectively. $bits\_cmn_{j,k}$ 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 its accessed channels will have access frequencies of infinity).

### 4.3 Hardware sharing

The hardware sharing metric measures the amount of hardware that two sets of behaviors could share. For example, if two behaviors both perform multiplication, then they may share a single multiplier. We assume a procedure *Size(BV)* exists that returns the hardware size for the given functional objects on a particular ASIC type. This size may be computed through synthesis, through summation of abstract hardware weights associated with each object, or through incremental synthesis techniques. If a particular ASIC type is not specified (along with a set of available functional units and a maximum number of each unit that can be used), then a default type is assumed. The hardware sharing metric is defined as:

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

$$\begin{aligned}
size_shared_{j,k} &= size_j + size_k - size_{j,k}, \\
size_{j,k} &= \text{Size}(BV_j \textstyle\bigcup BV_k), \\
size_j &= \text{Size}(BV_j), \\
size_k &= \text{Size}(BV_k), \\
norm &= size_both_min \text{ for local norm.,} \\
&= size_all \text{ for global norm.,} \\
size_both_min &= \text{Min}(size_j, size_k), \\
size_all &= \text{Size}(BV_{all})
\end{aligned}$$

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.

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.

### 4.4 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 improve performance by reducing inter-component communication. We assume a procedure *Accessors(bv)* exists 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 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:

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

$$\begin{aligned}
accessors_cmn_{j,k} &= accessors_j \textstyle\bigcap accessors_k, \\
accessors_j &= \textstyle\bigcup_{bv \in BV_j} \text{Accessors}(bv), \\
accessors_k &= \textstyle\bigcup_{bv \in BV_k} \text{Accessors}(bv), \\
norm &= |accessors_both_{j,k}| \text{ for local norm.,} \\
&= |accessors_all| \text{ for global norm.,} \\
accessors_both_{j,k} &= accessors_j \textstyle\bigcup accessors_k, \\
accessors_all &= B_{all}
\end{aligned}$$

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. $accessors\_cmn_{j,k}$ 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 $accessors$).

### 4.5 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 a procedure *SequentialBehs(b1,b2)* exists that returns 0 if behaviors $b1$ and $b2$ could execute concurrently, else it returns 1. This metric is defined as:

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

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

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.

### 4.6 Constrained communication

This metric is significantly unique from metrics defined in previous works. When we are given a set of performance constraints on behaviors, it is probably 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 a procedure *AccessedChansRecur(b)* exists that returns all channels accessed by behavior $b$, and all channels accessed by any accessee of $b$, etc. This metric is then defined as:

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

$bits\_cmn_{j,k}$
$= \sum_{t \in Tcons}$ConstrCommBehs($t.b, BV_j, BV_k$),
ConstrCommBehs($t.b, BV_j, BV_k$)
$= \sum_{c \in ContribChansRecur(t.b)} c.accfreq * c.bits$,
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,
$norm$
$= \sum_{t \in Tcons}$ ConstrComm($t.b, BV_j, BV_k$)
for local norm.,
$= \sum_{t \in Tcons}$ ConstrComm($t.b, BV_{all}, BV_{all}$)
for global norm.,
ConstrComm($t.b, BV_x, BV_y$)
$= \sum_{c \in AccessedChansRecur(t.b)} c.accfreq * c.bits$ if
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 $bits\_cmn_{j,k}$ 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.

This metric is particularly 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 tightly constrained behaviors for implementation in the hardware component.

Early investigations of this metric showed that proper normalization is crucial to effective metric use. In particular, the normalization denominator described above is usually quite large compared to the numerator, meaning that this metric's value is quite small compared to other metric values and thus doesn't play a significant role in partitioning unless weighed very heavily. We are looking into a better normalization method; in the meantime, we use a variation of this metric in which we compute just as the communication metric, but set the value to zero if the communication does not contribute to a constraint.

### 4.7 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.

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

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

## 5 Results

In this section, we describe how we determined the best combination of metrics for pre-assignment clustering and for N-way clustering. We selected six examples: a microwave-transmitter controller *mwt*, a fuzzy-logic controller *fuzzy*, a telephone answering machine *ans*, a volume-measuring medical instrument *vol*, an interactive-TV processor *itv*, and an Ethernet coprocessor *ether*. The first three examples represent control-dominated systems with some data computation, while the last three represent larger examples with a roughly-even mix of control and data computation. The examples averaged 630 lines of algorithmic-level VHDL.

### 5.1 N-way clustering

We developed a $2^k$ experiment to determine the effect of $k$ metrics during N-way clustering. (For information on $2^k$ experimental designs, see [18]). We examined the metrics of *Connectivity*, *Communication*, *Constrained communication*, *Common accessors*, and *Balanced size*; for conciseness, we shall refer to these below as *Cn, Cm, Cc, Ca* and *Bs*. We did not include *Hardware sharing*, because early experiments indicated that it was not useful for the given examples. Apparently, the hardware size estimator used a PLA model for control in which the control size for two behaviors was often larger than the sum of the control sizes for each behavior alone, which in turn meant that the hardware sharing metric was often negative. We also did not include *Sequential execution*, since only two of the examples involved concurrent processes.

For each example, we applied 2-way clustering 32 times, once for each of the $2^5$ possible combinations of the five closeness metrics. We experimented with both global and local normalization; global yielded slightly better results in this experiment. Each final partition was evaluated using a cost function that measured the magnitude of size, pin, and performance constraint violations; a cost of 0 meant no violation. We intentionally formulated constraints such that there would always be a constraint violation, so we could compare how close each partition was to 0.

Results are summarized in Table 2. Each row represents one of the five examples, with the last row representing the average over all examples. Each column represents a particular combination of metrics. Column 1 represents clustering without any metrics, which is essentially random clustering. Columns 2 through 6 represent clustering with the single metric of *Cn, Cm, Cc, Ca,* or *Bs*, respectively. Columns 7 through 16, 17 through 26, and 27 through 31 represent combinations of two, three, and four metrics, respectively. Column 32 represents all five metrics. The lowest averages occur in column 30 and 28, representing *Cn-Cc-Ca-Bs* and *Cn-Cm-Cc-Bs*, respectively. The next lowest average occurs in column 32, where all five metrics are used.

We have found that N-way clustering should be followed by fast greedy improvement for best results [19]. Table 1 compares such clustering (*Clust+grdy*) with greedy improvement (*greedy*), group migration (*Groupmig*) and simulated annealing (*Simann*) [20], all of which improve on a random initial partition. Results are shown for 3-way and 4-way hardware partitioning, and for hardware/software partitioning. Note that *Clust+grdy* compares favorably with the other heuristics, and in the last example actually finds the best solution in two cases.

### 5.2 Pre-assignment clustering

We performed a similar experiment for pre-assignment clustering. All combinations of metrics were applied for the examples to reduce the number of initial objects. In pre-assignment clustering, there are many possible methods for deciding when to terminate clustering (unlike N-way clustering, where we simply stop when there are only N groups remaining). We experimented with all possible closeness thresholds, and found 0.3 to yield the best results for our examples. In other words, terminating clustering when no objects had a closeness greater than 0.3 yielded good

| Ex | P | Greedy | | Groupmig | | Simann | | Clust+grdy | |
|---|---|---|---|---|---|---|---|---|---|
| | | C | T | C | T | C | T | C | T |
| vol | 3 | 50 | 3 | 0 | 20 | 22 | 220 | 96 | 17 |
| | 4 | 88 | 7 | 29 | 87 | 16 | 246 | 15 | 20 |
| | hs | 61 | 2 | 16 | 21 | 18 | 148 | 66 | 14 |
| ans | 3 | 25 | 8 | 7 | 198 | 0 | 174 | 16 | 34 |
| | 4 | 0 | 11 | 2 | 187 | 0 | 208 | 15 | 43 |
| | hs | 0 | 1 | 0 | 7 | 0 | 0 | 0 | 25 |
| itv | 3 | 115 | 32 | 71 | 953 | 63 | 609 | 142 | 175 |
| | 4 | 141 | 60 | 100 | 2953 | 94 | 625 | 137 | 200 |
| | hs | 83 | 12 | 20 | 296 | 20 | 379 | 67 | 151 |
| eth | 3 | 114 | 34 | 114 | 1064 | 97 | 520 | 5 | 422 |
| | 4 | 66 | 60 | 39 | 2036 | 72 | 693 | 37 | 443 |
| | hs | 102 | 20 | 23 | 598 | 0 | 268 | 76 | 378 |

Table 1: Existing N-way partitioning heuristics

time reductions without cost increase. Higher thresholds terminated clustering too early, resulting in only minor time reductions (but no cost increase), while lower thresholds merged too many objects, resulting in cost increases (though greater time reductions). Due to space limitations, we omit the detailed results of the experiment. The conclusion from the experiment was that the combination of the *Connectivity* and *Communication* metrics yielded the best results.

### 5.3 Effects of pre-assignment clustering

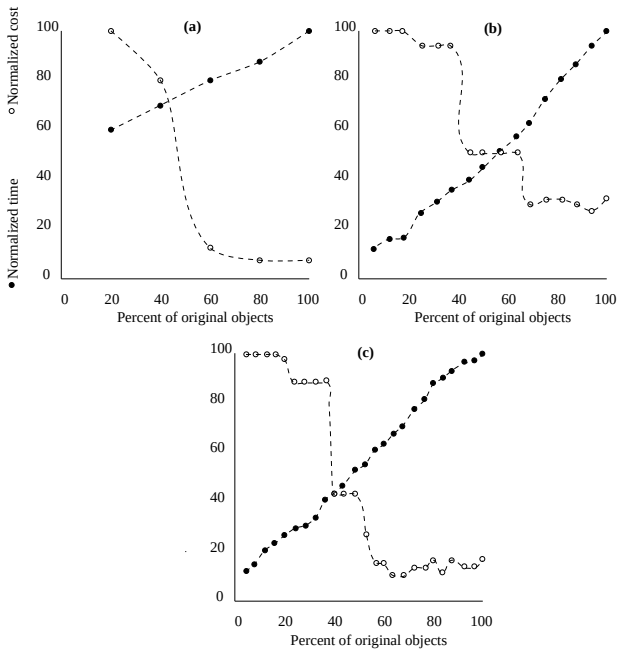


**Fig. 2**: Cost/time tradeoffs of pre-assignment clustering: (a) mwt, (b) itv, (c) ether.

To demonstrate the effect of pre-assignment clustering on improvement-heuristic runtime, we experimented further on *mwt*, *itv*, and *ether*. We followed pre-assignment clustering by random partitioning and then group migration. Results are shown in Figure 2. The x-axis is the percent of original objects remaining after pre-assignment

| Ex | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mwt | 131 | 65 | 131 | 145 | 131 | 121 | 131 | 151 | 131 | 65 | 131 | 83 | 131 | 121 | 131 | 111 | 138 | 65 | 138 | 15 | 137 | 121 | 137 | 106 | 138 | 15 | 138 | 83 | 137 | 94 | 137 | 94 |
| fuzzy | 180 | 77 | 260 | 77 | 180 | 77 | 180 | 77 | 180 | 216 | 180 | 242 | 180 | 72 | 180 | 72 | 181 | 77 | 181 | 77 | 181 | 77 | 181 | 77 | 181 | 163 | 181 | 66 | 181 | 63 | 181 | 98 |
| vol | 75 | 416 | 136 | 148 | 75 | 344 | 136 | 191 | 72 | 128 | 72 | 116 | 71 | 191 | 71 | 191 | 72 | 149 | 72 | 148 | 71 | 85 | 71 | 85 | 72 | 128 | 72 | 45 | 71 | 85 | 71 | 85 |
| ans | 244 | 208 | 244 | 240 | 244 | 258 | 244 | 258 | 249 | 158 | 249 | 246 | 249 | 258 | 249 | 185 | 248 | 134 | 248 | 134 | 248 | 134 | 248 | 134 | 248 | 134 | 248 | 134 | 248 | 141 | 248 | 141 |
| ether | 199 | 999 | 199 | 945 | 199 | 999 | 199 | 807 | 199 | 561 | 199 | 463 | 199 | 640 | 199 | 615 | 199 | 214 | 199 | 219 | 199 | 350 | 199 | 337 | 199 | 240 | 199 | 167 | 199 | 109 | 199 | 147 |
| Avg | 166 | 353 | 194 | 311 | 166 | 360 | 178 | 297 | 166 | 226 | 166 | 230 | 166 | 256 | 166 | 235 | 168 | 128 | 168 | 119 | 167 | 153 | 167 | 148 | 168 | 136 | 168 | 99 | 167 | 98 | 167 | 113 |

Table 2: 2-way clustering $2^k$ experiment to determine best closeness metric combination.

clustering. Thus, the right end of the axis represents no pre-assignment clustering, meaning all original objects are subsequently partitioned among parts; the center represents reduction to half of the original number. The y-axis represents the magnitude of constraint violations of the final partition after group migration, normalized to a number between 0 and 100. It also represents the runtime of group migration, again normalized. Results show that the fewer the number of objects input to group migration, the lower the runtime. In all three examples, we could reduce the objects by 25% (75% of the original number) for a 25% decrease in runtime without any increase in cost. We could the objects by 50% for a 50% decrease in runtime with only a small increase in cost. Such runtime reductions are significant since we may apply group migration hundreds of times, once for each possible combination of system components (whereas clustering is only done once).

## 6 Conclusion

We have defined several new closeness metrics between procedural-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 interaction is crucial for system-level tool acceptance, and partitioning at the procedural level achieves this goal.

We have demonstrated that these closeness metrics can be used by clustering algorithms to reduce the computation time of iterative-improvement partitioning algorithms without a loss in partition quality. Such a reduction of computation time 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, 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 enhance the rapidly growing number of system-design tools.

## References


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

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

[3] 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.

[4] F. Vahid and D.Gajski, "Specification partitioning for system design," in *Proceedings of the Design Automation Conference*, pp. 219–224, 1992.

[5] 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.

[6] P. Gupta, C. Chen, J. DeSouza-Batista, and A. Parker, "Experience with image compression chip design using unified system construction tools," in *Proceedings of the Design Automation Conference*, pp. 250–256, 1994.

[7] P. Eles, Z. Peng, and A. Doboli, "VHDL system-level specification and partitioning in a hardware/software cosynthesis environment," in *International Workshop on Hardware-Software Co-Design*, pp. 49–55, 1992.

[8] 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.

[9] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proceedings of the Design Automation Conference*, pp. 514–519, 1991.

[10] 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.

[11] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, pp. 21–32, March 1994.

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

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

[14] T. Ismail, K. O'Brien, and A. Jerraya, "Interactive system-level partitioning with Partif," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[15] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto, "A methodology for control-dominated systems codesign," in *International Workshop on Hardware-Software Co-Design*, pp. 2–9, 1994.

[16] F. Vahid and D. Gajski, "SLIF: A specification-level intermediate format for system design," in *Proceedings of the European Design and Test Conference (EDTC)*, 1995.

[17] M. McFarland and T. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on Computer-Aided Design*, pp. 938–950, September 1990.

[18] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling.* New York: John Wiley and Sons, Inc., 1991.

[19] F. Vahid and D. Gajski, "Clustering for improved system-level functional partitioning," in *International Symposium on System Synthesis*, 1995.

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