Procedure Cloning: A Transformation for Improved System-Level Functional Partitioning

Frank Vahid

Technical Report CS-96-6
August 5, 1996

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

Functional partitioning assigns the functions of a system's program-like specification among system components, such as standard-software and custom-digital processors. We introduce a new transformation, called procedure cloning, that significantly improves functional partitioning results. The transformation creates a clone of a procedure for sole use by a particular procedure caller, so the clone can be assigned to the caller's processor, which in turn improves performance through reduced communication and also reduces pins. We define several cloning heuristics that seek to clone the minimum number of procedures, a goal which must be met to obtain the best improvements. We highlight experiments demonstrating improved partition results with cloning, and comparing our cloning heuristics.

Contents

List of Figures

.

1 Introduction

Functional partitioning is becoming an increasingly important task for system design environments. In functional partitioning, a behavioral specification's functions are assigned to system components, which include standard processors, custom hardware processors, and memories. Such partitioning must satisfy packaging constraints, such as size and pin limits, while minimizing other metrics, such as execution time or power. Many research efforts have shown the benefits of hardware/software functional partitioning among standard and custom processors [1, 2, 3, 4, 5, 6], leading to reduced system costs and/or improved performance. Recent experiments have also shown dramatic benefits (i.e, fewer packages and better performance) of functional partitioning among hardware packages [7] as compared to the current approach of structural partitioning. To take advantage of these benefits, heuristics for rapid but high-quality functional partitioning must be developed.

Approaches to functional partitioning typically take as input a behavioral specification, which is a program-like description of desired system functionality. Developers of such specifications are faced with issues that have faced software developers for years, such as the importance of developing modular, readable, and reusable code. These issues lead to extensive use of procedures. Such procedures are often multiply-called from various places in the specification.

Most functional partitioning approaches convert the specification into an internal representation, and the objects that make up that representation are then partitioned among system components. The objects could be of various granularities, including statements, statement blocks, and procedures/processes. The question then arises: How does one handle multiply-called procedures? To our knowledge, two approaches have been considered: (1) Treat each procedure as a single computation, so a single instance of the procedure (or of its blocks or statements) are partitioned among components, or (2) Treat each procedure *call* as a computation, so multiple instances are partitioned.

The latter approach is required when we want to expose the largest possible solution space by using a dataflow graph internal representation, using a distinct graph node for each call to show the different data dependencies per call (similar to using distinct add nodes for each addition operation for behavioral synthesis [8]). However, this larger solution space, i.e., more nodes and hence more possible partitions, while possibly including better solutions than the first approach, provides a much harder problem to partitioning heuristics. Our experiments in this paper show that the number of nodes can increase by an order of magnitude (and thus the solution space by an even greater factor), leading to *grossly* inferior solutions, even from the very thorough partitioning heuristic of simulated annealing.

The former approach also has a drawback, but in this case our cloning transformation can eliminate this drawback. In particular, a single procedure instance may be called from procedures on other components, requiring inter-component communication, which reduces performance and may increase required pins. For example, in Figure 1(a), procedure *Main* calls procedures *Slow* 128 times and *Fast* 16 times, which in turn both call procedures *Util* 256 times and *Resource* 64 times. *Slow* is implemented in software, but *Util* and

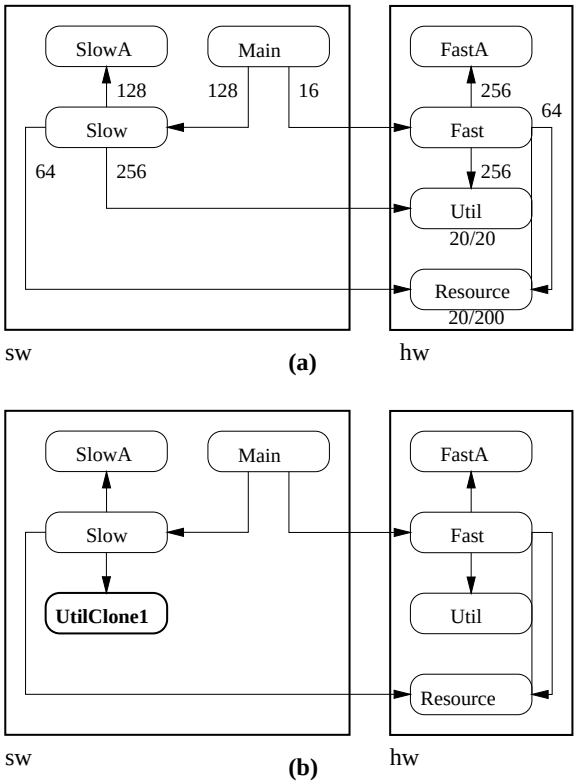


Fig. 1: Cloning example: (a) original partition, (b) reduced communication after cloning.

Resource are in hardware, leading to software/hardware communication. In some cases, such communication can't be avoided, as the called procedure is really a resource that must be implemented on a component separate from its callers, because of better performance on the separate component that far outweighs the extra communication, or because size limitations allow for only one procedure instance. For example, note that *Resource* requires many more cycles in software than in hardware (200 vs. 20). The time for communicating with and executing *Resource* is: $(128*64*(2+20))+(16*64*20) = 200,704$, assuming inter-component communication takes 2 cycles. If *Resource* were moved to software, then even if we ignored all communication, the time would would increase to: $(128*64*200)+(16*64*200) = 1,843,200$.

On the other hand, we can avoid the communication in the case of *Util*. Noting that *Util* requires 20 cycles in hardware or software, the time spent just communicating with and executing *Util* is: $(128 * 256 * (2 + 20)) + (16 * 256 * 20) = 802,816$ cycles. Moving *Util* to software would reduce the time to: $(128 * 256 * 20) + (16 * 256 * (2 + 20)) = 745,472$. However, we could gain an even better improvement if we create a unique copy of *Util* for use by *Slow* in software, as in Figure 1(b), i.e., we *clone Util* for *Slow*. The time drops even further to: $(128 * 256 * 20) + (16 * 256 * 20) = 737,280$. The *Util* procedure is really just a utility whose performance doesn't vary greatly among components and whose size is not very significant, and which therefore should usually be cloned to reduce communication.

We therefore see a need for an approach that clones procedures when beneficial to func-

tional partitioning. We have developed such a cloning approach. It uses an access graph representation, in which each procedure is initially represented as one node, coupled with a *procedure cloning transformation*. Cloning creates a copy of a procedure for exclusive use of one of its accessors. One might make an analogy with approaches that duplicate gates during circuit partitioning. Cloning heuristics are necessary to create just enough clones to improve partition results, without creating so many clones as to significantly increase the solution space and hence worsen partition results.

In this paper, we demonstrate the substantial partition improvements achieved when cloning is used, and we describe and compare several cloning heuristics that we have developed; in the process, we demonstrate clear inferiority of approaches that "maximally-clone" procedures in a dataflow graph representation to expose the largest solution space.

The paper is organized as follows. Section 2 defines the clone and unclone transformations as applied to an internal representation. Section 3 describes how estimation techniques must be modified to avoid computing extra size or I/O for same-part clones. Section 4 introduces several heuristics for performing cloning before, during and after functional partitioning. Section 5 highlights numerous experiments demonstrating cloning's benefits and comparing our heuristics. Section 5 provides conclusions.

2 Clone/unclone transforms

In this section, we describe our internal representation, define the clone transformation and provide an algorithm, define an unclone transformation which will be required by our heuristics, and contrast cloning with inlining.

2.1 Internal representation

We convert a specification into an Access-graph (AG) representation [9, 10], which we have found well-suited to solving system design problems, including partitioning and cloning. An AG node represents a behavior (procedure or process) or variable. An AG directed edge represents an access by one behavior to another behavior (i.e., a procedure call) or to a variable (i.e., a variable read or write). The edge direction indicates the accessor (the source) and the accessee (the sink), but does *not* indicate the direction of the flow of data; data could flow in either or both directions. A node can have fanin and fanout greater than 1, where **fanin** is the number of incoming edges, and fanout the number of outgoing edges. Each node is annotated with internal computation times (execution excluding any communication and accessed object times) and sizes for every possible type of implementation part (e.g., an Intel 8051 microcontroller or a Xilinx XC4000 FPGA). Each edge is annotated with its access frequency and the number of bits transferred per access. All annotations can be minimum, average, or maximum values. Equations have been developed to quickly compute size, I/O, and execution times (including communication) from these annotations for any given partition, but these are beyond the scope of this paper (see [9] for details).

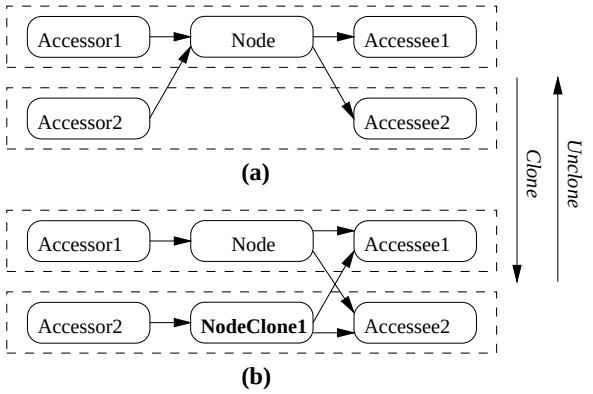


Fig. 2: Cloning an AG node.

2.2 Cloning

We first introduce the cloning transformation through a simple example. Consider the AG of Figure 2(a). *Node* has two accessors, *Accessor*1 and *Accessor*2. Cloning *Node* for *Accessor*2 results in the AG of Figure 2(b). *Accessor*2 now accesses its own copy *NodeClone*1, and no longer accesses *Node*. In addition, *NodeClone*1 accesses the same nodes (*Accessee*1 and *Accessee*2) that *Node* accesses. Also, because cloning is intended to allow an accessor to have a copy of the node on its own part, *NodeClone*1 has been created on *Accessor*2's part. Note that, in this example, we would probably need to further clone *Accessee*1 and *Accessee*2 for *NodeClone*1, in order to obtain a reduction in communication time and I/O.

More formally, the clone transformation, as applied to an AG, can be defined as follows:

- Input: (1) an AG, (2) a **base** n, which is the AG node to clone having $fanin > 1$, and (3) an **accessor** a, which is an AG node that accesses n via an edge e.
- Output: An AG with a new node n_{clone}, which is a copy of n including copies of outgoing edges, such that e now connects a to n_{clone}, $n_{clone}.fanin = 1$, and $n_{clone}.part$ is set to $a.part$.

Note that node cloning is only defined relative to a node's accessor; we must specify the *particular accessor* for which a node copy will be made. Also note that if n originally had a fanin of 1, cloning need not be performed because a is already the sole accessor of n.

Algorithm 2.1 provides an algorithm for cloning an AG node n for an accessor a:

Algorithm 2.1 Clone($slif$, n, a)

```
// Create unique node
n_clone = n.Copy()
n_clone.base = n
n_clone.id = clone.CreateUniqueCloneid
slif.AddNode(n_clone)
// Redirect edge to point to clone
e = slif.SeekEdge(n, a)
e.accessee = n_clone
// Copy node's outgoing edges for clone
for each oute ∈ n.outedges loop
    oute_copy = oute.Copy()
    oute_copy.accessor = n_clone
    slif.AddEdge(oute_copy)
end loop
// Put clone on accessor's part
if (n_clone.part! = a.part) then
    slif.UpdateForMove(n_clone, a.part)
end if
```

The algorithm first copies n and assigns the copy n_{clone} a unique name, while recording the clone's base. Recording the base is necessary for uncloning and for estimation reasons, as will be seen later. Next, the algorithm redirects a's edge to point to n_{clone} instead of n. It then copies n's outgoing edges and makes the copies originate from n_{clone}. Finally, it moves n_{clone} to a's part.

We currently only allow cloning of nodes representing procedures, not variables, even though both node types can have fanin > 1. Cloning of a variable node would prevent the variable accessors from communicating data through the variable. In the future, it might be interesting to determine if data is actually being communicated through the variable; if not, cloning of the node should be allowed.

2.3 Uncloning

Uncloning is the inverse transformation of cloning, but in some cases requires more work than undoing the changes of the clone transformation, as we shall see.

Consider the AG of Figure 2(b). $Node$ has a clone $NodeClone1$, which was created for $Accessor2$. Uncloning $NodeClone1$ to $Node$ results in the AG of Figure 2(a). $NodeClone1$ is gone along with its outgoing edges, and $Accessor2$ now accesses $Node$ instead.

More formally, the unclone transformation, as applied to an AG, can be defined as follows:

- Input: (1) an AG, (2) a clone n_{clone}, which is the AG node to be uncloned, and (3) an identical node n_{ident}.

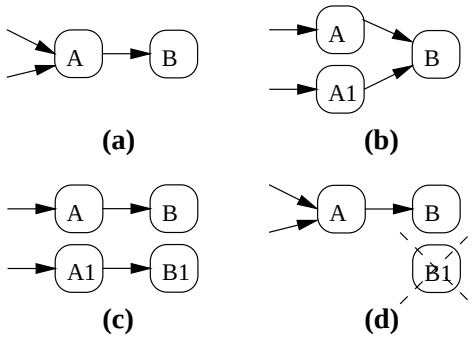


Fig. 3: Sequence leading to a stranded node.

- Output: An AG in which n_{clone}'s incoming edges now point to n_{ident}, and which does not include n_{clone}, any of n_{clone}'s outgoing edges, or stranded nodes.

A stranded node is a node that had incoming edges before the unclone, but none after the unclone. For example, if we had cloned $Accessee1$ and $Accessee2$ for $NodeClone1$ in Figure 2(b), resulting in $Accessee1Clone1$ and $Accessee2Clone1$, and we then deleted $NodeClone1$ while uncloning it to $Node$, $Accessee1Clone$ and $Accessee2Clone$ would not have any accessors, so they too should be deleted. A simpler example is given in Figure 3, showing a sequence of clone and unclone transforms that lead to a stranded node. Starting with Figure 3(a), we clone A for one of its accessors, resulting in $A1$ of Figure 3(b). This clone increases B's fanin to 2, making B a candidate for cloning. Cloning B for $A1$ yields $B1$ in Figure 3(c). Now, if we unclone $A1$ to A, as in Figure 3(d), $B1$ would be stranded. Thus, uncloning must delete such stranded nodes. Deleting a stranded node may yield further stranded nodes, which must also be deleted.

An identical node n_{ident} is defined as a node with the same base as n_{clone}. In most cases, n_{ident} is in fact the base n itself from which n_{clone} was created, but it could also be a fellow clone of n. We use the terminology of uncloning a node n_{cloned} *to* another node n_{ident}.

Note that uncloning is defined between two nodes. Initially, since cloning was defined for an accessor and a node which was then cloned, one might assume that uncloning should be defined for an accessor and the clone. There are two reasons why one doesn't specify the accessor. First, the accessor can be easily found by looking at the clone's incoming edges. Second, and more importantly, a sequence of clone and unclone transforms can lead to a clone having *more than one accessor*, even though a clone when it is first created always has exactly one accessor. For example, consider Figure 4. B of Figure 4(a) is cloned for A, yielding $B1$ in Figure 4(b). Then, A is cloned for one of its accessors, yielding $A1$ in Figure 4(c). $B1$ now has two accessors, $A1$ and A. Now that we see that we don't specify the accessor for uncloning, one might then assume that only the clone needs to be specified. However, because we don't necessarily have to unclone to the base, but instead to a fellow clone of the base, we must specify the identical node to which to unclone.

For brevity, we omit the details of the uncloning algorithm, and instead include a short description. The algorithm first redirects n_{clone}'s incoming edges to point to n_{ident}. Next, it

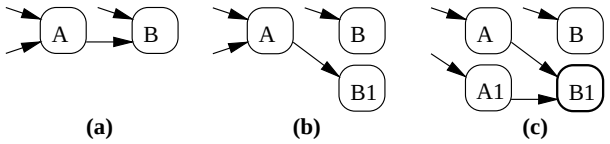


Fig. 4: Sequence leading to clone with multiple accessors.

passes n_{clone} to a function *Delete*. This function removes the given node and all outgoing edges, and then checks each accessee of those outgoing edges; if an accessee is now stranded, the *Delete* function recursively calls itself with the accessee.

2.4 Cloning is not inlining

Procedure cloning should not be confused with procedure inlining. In inlining, a procedure call is replaced by the procedure's contents. While inlining also has the effect of copying a procedure for sole use of the accessor, it also results in inlined implementation of the procedure, which is often not desired. In particular, the accessor can have multiple calls to the procedure; inlining replaces each call by the procedure contents, resulting in multiple instances of those contents. Inlining can result in explosive growth of a specification's size, and in turn of the implementation's size. In contrast, cloning only changes each procedure call's identifier to the clone's identifier, so there is just one instance of the clone procedure. Subsequent behavioral synthesis or software compilation can still tradeoff the different methods for procedure implementation, including as a control subroutine, a custom processor, a datapath functional unit, and as inlined contents. Also, consider the case of having multiple clones or inlined procedure contents on the same part. With multiple clones, we can easily modify size estimation to only consider each clone once, and we can easily unclone to a single procedure when partitioning is complete. With inlined procedure contents, size is greatly increased, and recombining multiple instances of contents is a non-trivial task.

3 Estimation modifications

The existence of clones in an AG requires modification of estimation techniques. Estimation of metrics, such as performance, size, and I/O, must be made for each partition examined by partitioning heuristics. If we don't modify existing estimation techniques, then when we have multiple same-base nodes (clones) on one part, we would add those nodes' sizes while computing the part size. However, recall that we cloned a procedure for a behavior to prevent communication by the behavior with another part. If multiple clones are on the same part, that means that multiple behaviors on that part access the clones. Those behaviors can share a single version of those clones and still not have to communicate with another part, so we always unclone after partitioning until each part has at most one node with a given base. We therefore see that we must modify our size estimation to account for multiple clones on the same part. A similar problem must be solved for I/O estimation.

3.1 Modifying size estimation

There are two methods for estimating size: weight based, and design based. Design-based methods [11] maintain a rough design for each part in a partition, along with contributions

made to that design by each node on that part (e.g., the number of control steps, the datapath paths, the control lines between the control unit and datapath, the number of temporary values, etc.). When a node is moved to or from a part, the part's design is incrementally modified based on the node's contributions. Design-based methods can be made fast, by ensuring that the incremental modifications take constant-time, but they are more complex to implement and far more complex to discuss in a paper such as this one. In contrast, weight-based methods are much easier to discuss, since they simply associate a size with each node, and then compute a part's size by summing its nodes' sizes. We therefore limit our discussion to weight-based methods, but extensions can be made for design-based methods also.

If we do not consider clones, a part P's size can be estimated simply by summing its nodes' sizes, as shown in Algorithm 3.1. In addition, we can easily incrementally update a part's size when a node is added or deleted, which occurs when a node is moved from one part to another during partitioning, by subtracting $n.size$ from the source part, and adding it to the destination part.

Algorithm 3.1 SizeEstBefore(P)
```
for each n ∈ P.N loop
    P.size+ = n_i.size
end loop
```

Modifying such size estimation to consider clones means that we must consider same-base nodes only once for a given part. A modified size estimation algorithm is given in Algorithm 3.2. The algorithm maintains a list L of bases seen so far, and only adds a node's size the first time a base is seen. Incremental update routines would require that we keep a list of bases along with a count of the number instances of each base; when a node is added or deleted, its base's count is incremented or decremented, and the size is only updated when the count changes from 0 to 1 or from 1 to 0.

Algorithm 3.2 SizeEstWithCloning(P)
```
for each n ∈ P.N loop
    base = n_i.base
    if ( !P.L.seek(base) ) then
        P.L.append(base)
        P.size+ = n_i.size
    end if
end loop
```

3.2 Modifying I/O estimation

I/O is defined as the number of input/output pins required on a part.

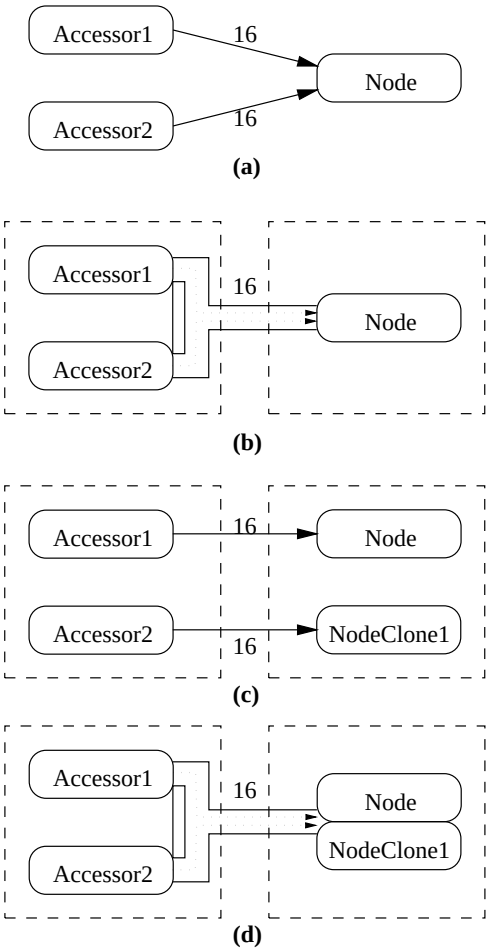


Fig. 5: Modifying I/O estimation for cloning.

Figure 5 demonstrates the modification necessary for I/O estimation when considering cloning. First, we consider I/O estimation without clones. Figure 5(a) shows an AG where *Node* has two accessors. When the node is separated from its accessors, as in Figure 5(b), two edges would be cut. However, those edges point to the same node, so I/O estimation would only count one edge, because sequential accesses to the same node can always share the same I/O. If clones are present, as in Figure 5(c), the above estimation technique would see two cut edges that point to different nodes, so each edge would contribute to the I/O estimate. However, as discussed above, we know that clones on the same part would always be uncloned, so we must consider those two edges as pointing to the same node, as shown in Figure 5(d).

An algorithm for modified I/O estimation is shown in Algorithm 3.3. The algorithm maintains a list M of bases seen so far. The algorithm only adds a cut edge's width the first time that the edge's accessee's base is seen. This list of accessee bases is maintained both for the accessor's part as well as the accessee's part.

Algorithm 3.3 IoEstWithCloning(P)
```
for each e ∈ P.cutedge loop
    base = e.accessee.base
    if ( !P.M.seek(base) ) then
        P.M.append(base)
        P.IO+ = e.wires
    end if
end loop
```

3.3 Modifying performance estimation

We compute a node's performance, or execution-time, for a given partition by adding the node's internal computation time (ict) and its communication time. We compute communication time as the time spent transferring data to/from accessed nodes, plus the execution-time of those nodes. Thus, this recursive technique for computing execution time considers: (1) the unique ict value for the part to which a node is partitioned, (2) the different data-transfer times that occur for same-part and different-part accesses, and (3) the different execution-times of the accessed nodes for the given partition. Further discussion can be found in [9].

Because a clone's annotations will be identical to its base's annotations, we do not need to modify the execution-time estimation technique for clones. When a clone is moved to a different part, the technique will automatically use a different ict value and different data-transfer times when computing that clone's execution-time.

4 Cloning heuristics

Now that we have defined the clone and unclone transformations, and have discussed how estimation techniques can be modified for the existence of clones, we can discuss various heuristics for finding the best procedures to clone.

We can classify cloning heuristics into three categories:

1. *Pre-partition cloning*: we first clone some subset of procedures, and then apply partitioning on the new AG.

2. *Post-partition cloning*: we apply partitioning to the original AG, and then clone some subset of procedures based on the partition. We can then reapply partitioning on the new AG.

3. *Integrated partitioning and cloning*: we extend an iterative-improvement partitioning heuristic to not only move nodes among parts, but to also clone and unclone nodes, in its search for a lower-cost partition.

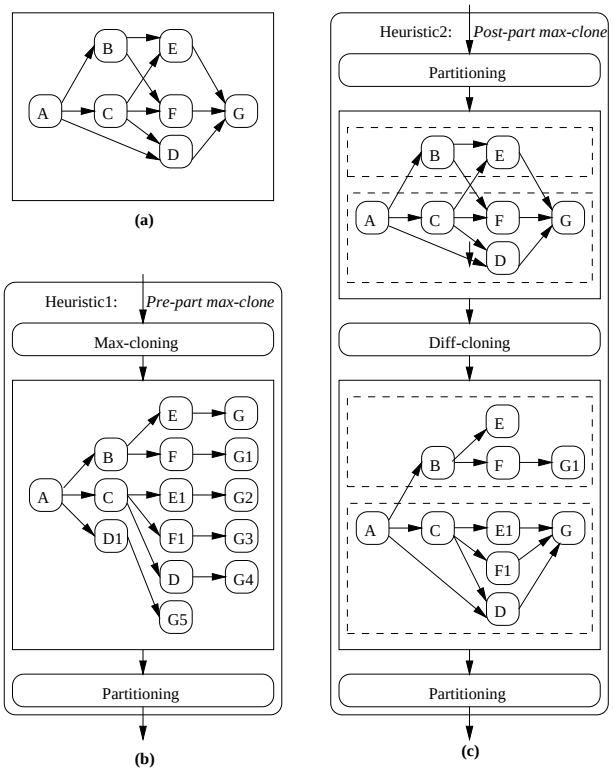


Fig. 6: Two cloning heuristics: (a) input AG, (b) pre-partition max-cloning, (c) post-partition max-cloning.

4.1 Pre-partition cloning

The most straightforward pre-partition cloning technique is max-cloning. In *pre-partition max-cloning*, we clone procedures until no procedure has more than one accessor, as illustrated in Figure 6(b). Max-cloning provides the largest possible solution space to subsequent partitioning. As is the case with all cloning heuristics, clones on the same part after partitioning will be uncloned, so this technique would appear initially to be the best since it exposes the most solutions. We point out that max-cloning is very similar to creating a dataflow graph instead of an access-graph from the original specification; since each procedure call would involve distinct data, each call would require a distinct node.

However, max-cloning results in a large increase in the number of AG nodes, which we have found leads to inferior results, as will be discussed in Section 5. One might initially assume that this increase is bad because it will cause less accurate estimates; however, Section 3 described estimation modifications that cause clones to have no effect on accuracy. Instead, the increase is bad because partitioning heuristics do *not* consider the entire solution space, due to the NP-completeness of the partitioning problem. Instead, they consider a subset of solutions, and the large increase in nodes means that the best partitions may never even be considered. In addition, the large number of nodes usually means much longer partitioning runtimes.

How much of an increase in nodes does max-cloning yield? The example of Figure 6(a) initially consists of 7 nodes, which increases to 15 nodes after max-cloning. In general, the increase will depend on each node's fanin and depth (the longest path from a root to the node). The number of instances of a particular node will equal the number of accessors of the node after all predecessors have been max-cloned. We can thus compute the number of nodes that would exist after max-cloning by using Algorithm 4.1. The algorithm uses a function $TopologicalSort$, which returns a topological ordering of the AG nodes, i.e., a list such that no node appears in the list before any of its accessors. Each node has a *count* field that indicates how many instances of this node would appear after max-cloning. *total* keeps track of the total number of nodes.

Algorithm 4.1 ComputeNumMaxCloneNodes(AG)

```
nodes = TopologicalSort(AG)
total = 0
for each n ∈ nodes loop
   if n.accessors.len = 0 then
      n.count = 1
   else
      n.count = 0
      for each e ∈ n.incoming_edges loop
         n.count+ = e.count
      end loop
   end if
   total+ = n.count
end loop
return total
```

The algorithm traverses the AG nodes in topological order. Nodes with no accessors (root nodes) will be visited first; those nodes will never be cloned since cloning is only defined for a node and its accessor. Thus, we set the count field of those nodes to 1. The number of accessors of subsequent nodes is computed as the sum of the count fields of accessing nodes.

Quickly computing the number of nodes for max-cloning using the above algorithm is useful when we wish to limit max-cloning to cases where the resulting number of nodes would not exceed some specified limit, without actually performing max-cloning.

A second pre-partition cloning technique is best-cloning. In *pre-partition best-cloning*, we attempt to predict which clones of nodes for accessors would best improve the cost after partitioning. One predictor might be the number of node accessors: the more accessors, the more likely that an accessor will be on a different part and hence benefit from a clone. A second predictor might be the node size; the smaller the node, the less effect cloning the node has on part sizes. A third predictor might be the access frequencies of incoming edges: the higher the frequencies, the greater the penalty for inter-part communication and hence

the more likely the benefit of cloning. The various predictor values could be normalized and summed into a single value indicating the probable improvement after cloning, and some number of those best candidates, perhaps those whose value exceeds some threshold, could then be cloned. Because of the excellent results of our other heuristics, we have not yet investigated pre-partition best-cloning.

4.2 Post-partition cloning

As with pre-partition cloning, we can distinguish between two post-partition cloning techniques: max-cloning and best-cloning.

In *post-partition max-cloning*, we clone every node (with fanin > 1, of course) *for every accessor on a different part* than the node itself, as long as the node also has an accessor on the same part. Contrast this with pre-partition max-cloning, in which we cloned a node for every accessor, not just those on different parts (since pre-partition implies, of course, that parts do not even exist yet). Figure 6(c) provides an illustration.

To understand the intuition behind requiring one same-part and one different-part accessor, consider the following possibilities for a node with at least two accessors:

1. *All accessors are on the same part as the node*: In this case, there is no need to clone since all accessors already have a same-part version of the node.

2. *All accessors are on different parts than the node*: In this case, if providing a clone on one of the accessor's parts would yield an improvement, then partitioning probably would have already placed the node on that accessor's part. The fact that the node appeared on a distinct part probably means that the node won't fit on the other parts, or that the node's ict was much better on its current part but the accessor could not be moved there.

3. *At least one accessor is on the same part and another is on a different part*: In this case, the same-part accessor could have been the reason that the node didn't appear on another accessor's part. Cloning will definitely reduce communication and I/O; whether this reduction outweighs the increase in the accessor's part size and the possible reduction in the node's ict will be seen only after repartitioning.

In *post-partition best-cloning*, we again attempt to predict which clones of nodes for accessors would best improve the cost after repartitioning. Of course, now that we have an initial partition, we have more information than pre-partition best-cloning. One approach to best-cloning is to clone those nodes for accessors that improve the partition's cost (which was not known during pre-partition cloning).

4.3 Integrated partitioning and cloning

Iterative-improvement partitioning heuristics change a given partition, usually by moving a node from one part to another part, hundreds or thousands of times using a control strategy that attempts to overcome local cost minima without examining excessive numbers of moves.

A third approach to cloning, quite distinct from pre-partition and post-partition cloning, is to modify the definition of a "change" in such partitioning heuristics from being just a move of a node to either a move, clone, or unclone.

The simulated annealing heuristic is a popular partitioning heuristic for which such a modification is straightforward. The modified heuristic is shown in Algorithm 4.2. In the unmodified version of the heuristic, a function *RandMove* would be called in the inner loop. In the modified version, a function *RandChange* is called. The function has three parameters in addition to the partition P. Each represents the probability of performing each type of change, i.e., a move, clone, and unclone respectively. *RandChange* randomly chooses the type of change, using those probabilities. A move consists of choosing a random node and a random destination part, and then moving the node to that part. A clone consists of choosing a random node with fanin > 2 and a random accessor of that node, and then applying the clone transformation of Algorithm 2.1. An unclone consists of choosing a random clone node, and then uncloning that node to its base. As will be discussed in the experiments, we have found that good results are obtained using clone/unclone probabilites that are small relative to the move probability (e.g., 0.05 each).

Algorithm 4.2 : Simulated annealing with cloning

$temp$ = initial temperature
$cost$ = Objfct(P)
while not Frozen **loop**
  while not Equilibrium **loop**
    $P\_tentative$ =
      RandChange(P,f_{move},f_{clone},$f_{unclone}$)
    $cost\_tentative$ = Objfct($P\_tentative$)
    $\Delta cost = cost\_tentative - cost$
    if (! Reject($\Delta cost, temp$)) **then**
      $P = P\_tentative$
      $cost = cost\_tentative$
    end if
  end loop
  $temp$ = DecreaseTemp($temp$)
end loop

4.4 Max-uncloning

Regardless of the cloning heuristic used, the resulting partition will likely have multiple same-based nodes (clones) on a single part. These nodes can be uncloned to a single node, which will be shared by the accessors. We thus define a *post-partition max-unclone* transformation, which unclones all same-base nodes on a single part through repeated application of the unclone transformation.

eg	none	none2	premax	postmax	integ
ans	69	69	42	56	39
clone1	13589	13589	13507	13507	13507
ether	627	627	3764	323	323
fuzzy	978	978	978	978	978
itv	143	143	143	143	143
mwt	226	74	74	75	74

(a)

eg	none	none2	premax	postmax	integ
ans	79	289	86	91	96
clone1	9	36	8	11	10
ether	75	258	99	105	109
fuzzy	66	233	66	83	61
itv	119	469	128	161	124
mwt	50	201	67	65	59

(b)

Fig. 7: Examples: (a) costs, (b) runtimes

5 Experiments

We have conducted experiments on several examples to demonstrate the improvements that can be gained using cloning, and to compare the various cloning heuristics.

The examples included an answering machine (*ans)*, an Ethernet coprocessor (*ether*), a fuzzy-logic controller (*fuzzy*), an interactive TV processor (*itv*), a microwave transmitter controller (*mwt*), and the example from Section 1 (*clone1*). 14 other examples were generated, ranging in size from 10 to 130 nodes, using the techniques describe in [10]).

After converting each example to an AG and extensively annotating the AG with estimations obtained using estimators available from UC Irvine's SpecSyn tool ([12]), we partitioned each example among a hardware/software architecture consisting of an 8086 processor and a Xilinx XC4000-series FPGA. Each example had at least one node with an execution-time constraint; this node was usually a root node, so the execution-time included communication and execution of many other nodes. Details of the estimation techniques are beyond the scope of this paper; we refer the reader to [12, 9]. We used a cost function with three terms: the total execution time of the constrained nodes, the FPGA I/O constraint violation, and the FPGA size constraint violation. The latter constraints were weighed extremely heavily to ensure that they were not violated, so all final costs represent the execution time only.

The heuristics were then compared using three factors: (1) the example's cost after partitioning, (2) the heuristic's running time, and (3) the number of nodes after post-partition max-uncloning. A good heuristic should minimize all three factors. We obviously want to minimize cost (execution-time) and runtime. We want to minimize the last factor, which indicates the number of clones that were required to obtain the reduced execution-time, because fewer clones leads to smaller implementations, which is especially important in embedded systems.

Figure 7 summarizes execution-time costs (cycles x 100) and run times (seconds on a

eg	none	premax	unclone	postmax	unclone	integ	unclone
ans	45	52	46	46	46	46	46
c1	7	9	8	9	8	9	8
ether	123	142	125	126	125	140	124
fuzzy	70	70	70	70	70	70	70
itv	85	85	85	85	85	85	85
mwt	30	55	35	35	30	33	33

Fig. 8: Examples: numbers of nodes

166MHz Pentium) for the first six examples. *none* represents results without any cloning, using simulated annealing. To determine if further improvements could be gained without cloning, we ran simulated annealing again with a much more time-consuming cooling schedule (roughly 4 times longer than the previous schedule); *none2* represents results using that schedule. *premax* is pre-partition max-cloning, *postmax* is post-partition max-cloning, and *integ* is integrated partitioning and cloning using simulated annealing (with the same cooling schedule as used in *none* above), with clone and unclone probabilities of 0.05, and move probability of 0.9. *premax*'s runtime represents the time for partitioning after cloning, while *postmax*'s runtime represents the sum of the times for partitioning before and after cloning.

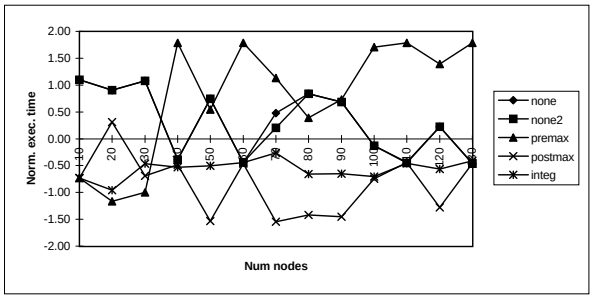
We have omitted results for post-partition best-cloning, because of the (rather suprising) fact that it did not result in improvement for any of the 19 examples. Apparently, after partitioning, no single clone would reduce cost; instead, a sequence of clones and moves would be required to escape a local minimum. Future work might focus on finding such sequences.

Cloning provides excellent improvements in execution time for *ans*, *clone1*, and *ether*. Those improvements could not be obtained without cloning, as indicated by the fact that *none2* had identical cost as *none*. No improvement was gained in the other three examples because they did not include many procedures called from multiple behaviors. In the *ether* example, *integ* yielded a clone of a procedure for sending data to other processes. In the *ans* example, *integ* yielded a clone of a procedure for activating a unit that produces a beep. Note that these are precisely the utility-type procedures that are most likely to yield best results when cloned.

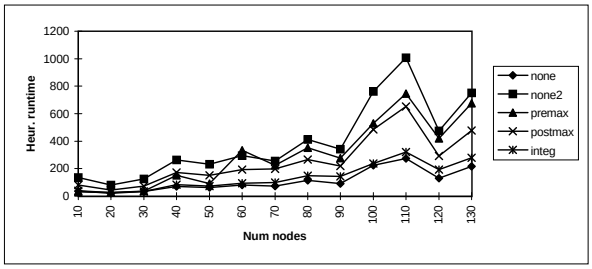
The *integ* heuristic performed the best. It resulted in the lowest costs for all six examples, had runtimes not much more than *none*, and resulted in the fewest final nodes of any of the cloning heuristics. Figure 8 lists the number of nodes remaining after each cloning heuristic, and also after final max-uncloning after the final partition is reached (the *unclone* column after each heuristic's column).

To get a better understanding of the improvements possible using cloning and the relative performance of the heuristics, we applied the heuristics to the 14 generated examples. These examples are highly procedural, and their sizes are from 10 to 130 nodes in increments of 10 nodes, so heuristic performance as a function of problem size can be ascertained. In this case, we show the costs and runtimes, and number of objects, as plots rather than tables, in Figure 9 and Figure 10. The costs have been normalized. Unnormalized costs are found in Figure 11.

Premax yielded the worst costs (often worse than no cloning at all!) for all but a few



(a)



(b)

Fig. 9: Generated examples: (a) normalized costs, (b) heuristic runtimes

examples. These poor results can be explained by the tremendous growth in the number of nodes that occurs, as seen in Figure 10, which in turn presents a much larger solution space to the partitioning heuristic. For example, the example with 60 nodes grew to 572 nodes after max-cloning. (Numbers for examples 100 through 130 were off the chart; they were 810, 922, 741, and 998, respectively). In addition to poor results, *premax* also required long runtimes. (These results also demonstrate why an access graph is well-suited as compared to a dataflow graph for system-level design from highly-procedural specifications).

Postmax yielded the best costs for most of the examples. However, its runtime was roughly double that of *integ*, since partitioning has to be applied twice, once before the cloning, and once after. In addition, *postmax* yielded many more objects than *integ* after max-uncloning.

Integ yielded consistent improvements over *none* (with only one exception) with only minor increases in runtimes. In addition, it resulted in the fewest final nodes after max-uncloning compared to all the other cloning heuristics.

We also evaluated *integ* with clone and unclone probabilities of 0.1 each, and with move probability of 0.8. Its performance was spotty. In some cases, it yielded a cost lower than any other heuristic; for example, it yielded a costs of 32 and 252 for *ans* and *ether*. In other cases, it yielded costs worse than the other heuristics; for example, it yielded a cost of 226 for *mwt*. We also tried performing two iterations of post-partition max-cloning, i.e., first partitioning, then max-cloning, then repartitioning, then max-cloning again, followed by another repartitioning. Improvements were found in some cases, but they were quite minor; in other cases, results were worse.

In summary, both *postmax* and *integ* cloning heuristics yielded excellent cost improvements

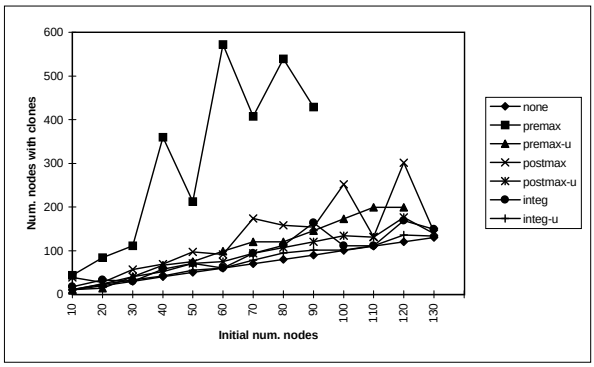


Fig. 10: Generated examples: numbers of nodes

over partitioning without cloning. *Postmax* yielded the best costs, while *integ* yielded the best runtimes and final number of nodes. The choice of a heuristic thus depends on the relative importance of those three factors.

6 Conclusions

We have demonstrated that significant improvements can be gained by incorporating procedure cloning with functional partitioning. We have shown that post-partition max-cloning and integrated partition and cloning are both good cloning heuristics, each having its own advantages. In the process, we have shown the inferiority of approaches that expose a bigger solution space by creating a dataflow graph (pre-partition max-cloning). The success of cloning makes it an essential task in any system-level functional partitioning tool.

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

[7] F. Vahid, T. Le, and Y. Hsu, "A comparison of functional and structural partitioning," in *International Symposium on System Synthesis*, 1996.

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

[9] F. Vahid and D. Gajski, "SLIF: A specification-level intermediate format for system design," in *Proceedings of the European Design and Test Conference (EDTC)*, pp. 185–189, 1995.

eg	none	none2	premax	postmax	integ
10	35118	35118	35058	35058	35058
20	555	555	59	413	108
30	4134	4134	1707	2062	2329
40	91577	91570	94118	91457	91401
50	12181	12181	12000	10099	11038
60	14497	14506	18912	14455	14491
70	3324	2961	4198	637	2336
80	7184	7184	6285	2666	4192
90	3720	3720	3790	126	1473
100	528643	528643	535216	526431	526582
110	589678	589674	590383	589675	589671
120	53163	53163	57826	47156	50022
130	709050	709041	718558	709053	709268

Fig. 11: Generated examples: costs

[10] F. Vahid and T. Le, "Towards a model for hardware and software functional partitioning," in *International Workshop on Hardware-Software Co-Design*, pp. 116–123, 1996.

[11] F. Vahid and D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," in *IEEE Transactions on Very Large Scale Integration Systems*, pp. 459–464, 1995.

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