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

FRANK VAHID
University of California

Functional partitioning assigns the functions of a system's program-like specification among system components, such as standard-software and cusotm-hardwre 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. Heuristics are used to prevent the exponential size increase that could occur if cloning were done indiscriminately. We introduce a variety of cloning heuristics, highlight experiments demonstrating the improvements obtained using cloning, and compare the various cloning heuristics.

Categories and Subject Descriptors: B.0 [**Hardware**]: General; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids; *automatic synthesis*; *hardware description languages*; *optimization*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*computer-aided design* (CA)

General Terms: Design

Additional Key Words and Phrases: Behavioral synthesis, embedded systems, functional partitioning, hardware/software codesign, replication, system-on-a-chip, system-level design, transformations

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 software processors, custom hardware processors, and memories. Such partitioning may be among multiple packages or among blocks of a single system-on-a-chip, and must satisfy constraints on I/O, size, performance, and/or power. Many research efforts have shown the benefits of hardware/software functional partitioning among standard and custom

Author's address: Department of Computer Science, University of California, Riverside, CA 92521; email: vahid@cs.ucr.edu; http://www.cs.ucr.edu/~vahid.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1999 ACM 1084-4309/99/0100–0070 $5.00

processors [Gupta and DeMicheli 1993; Ernst et al. 1994; Antoniazzi et al. 1994; Thomas et al. 1993; Xiong et al. 1994; and Eles et al. 1992], leading to reduced system costs and/or improved performance. Recent experiments have also shown dramatic benefits, such as less I/O, fewer packages, better performance, reduced synthesis runtimes, and reduced power consumption, obtained by functional partitioning among hardware packages [Vahid et al. 1996] as compared to the current approach of structural partitioning [Johannes 1996]. 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.

Functional partitioning approaches convert the specification into an internal representation, whose objects are then partitioned among system components. The objects could be of various granularities, including statements, statement blocks, and procedures. 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) is 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 addition nodes for each addition operation for behavioral synthesis [Orailoglu and Gajski 1986]). However, this larger solution space provides a much harder problem to partitioning heuristics. Our experiments in this paper show that the number of nodes can increase by nearly an order of magnitude (and thus the solution space by an even greater factor), leading to grossly inferior solutions.

The former approach also has a drawback, but in this case our cloning transformation can eliminate it. The drawback is that a single procedure instance may be called from procedures on other components, requiring intercomponent communication, but in some cases creating local copies of a procedure for each call would eliminate this communication. To demonstrate, consider Figure 1(a), which shows a partial functional specification with four procedures A, B, C, and D. Figure 1(b) shows the calling relationships among those procedures, with the calling frequencies denoted as v, w, x, y, and z. Also shown are the execution times Cs and Ds for procedures C and D on a particular software processor s (such as an Intel 8051), and the times Ch and Dh on a custom hardware processor h (such

as a Xilinx FPGA). Figure 1(c) shows a partition of those procedures among parts s and h, where the interpart data transfer time is shown as u; without loss of generality, we assume in this example that the intrapart transfers take 0 time. Given this partition, the total times spent communicating with and executing procedures C and D are shown in Figure 1(d). Both C and D are accessed from both s and h, so each procedure requires intercomponent communication. Assuming a move of A or B would be undesirable (for reasons not visible in the example), we might try moving C or D to improve the execution time. Moving C would cause a time increase, as shown in Figure 1(d), from 220 to 230 because A communicates more with C than does B. Likewise, moving D would cause an increase from 250 to 2050, in this case because D executes much more slowly on s than on h. Hence, partitioning cannot improve the execution time. However, suppose we create copies of C and D on each part, changing communication with them from A or B from interpart to intrapart, as shown in Figure 1(e). The time related to C decreases, as shown in Figure 1(f), from 220 to 180. However, the time related to D increases from 250 to 1100, because even though communication time is reduced, execution time is increased since D is much slower on s.

The solution to the drawback is to develop an approach that clones only those procedures that yield improvement. In the above example, we would want to clone C, but not D, to obtain an improvement in time of $220 - 180 = 40$. 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 by one of its accessors. One might make an analogy with approaches that replicate gates during circuit partitioning. Heuristics to guide such cloning are then 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.

The paper is organized as follows. Section 2 discusses related work. Section 3 provides a problem description. Section 4 defines the clone and unclone transformations. Section 5 describes how estimation techniques must be modified to account for clones. Section 6 introduces several heuristics for performing cloning before, during, and after functional partitioning. Section 7 highlights numerous experiments demonstrating cloning's benefits and comparing our heuristics. Section 8 provides conclusions.

2. RELATED WORK

One of the first functional partitioning approaches was Aparty [Lagnese and Thomas 1989; Lagnese and Thomas 1991]. Aparty reads a specification into an arithmetic-level dataflow graph and then partitions the arithmetic operations into datapath blocks using a multistage clustering heuristic.

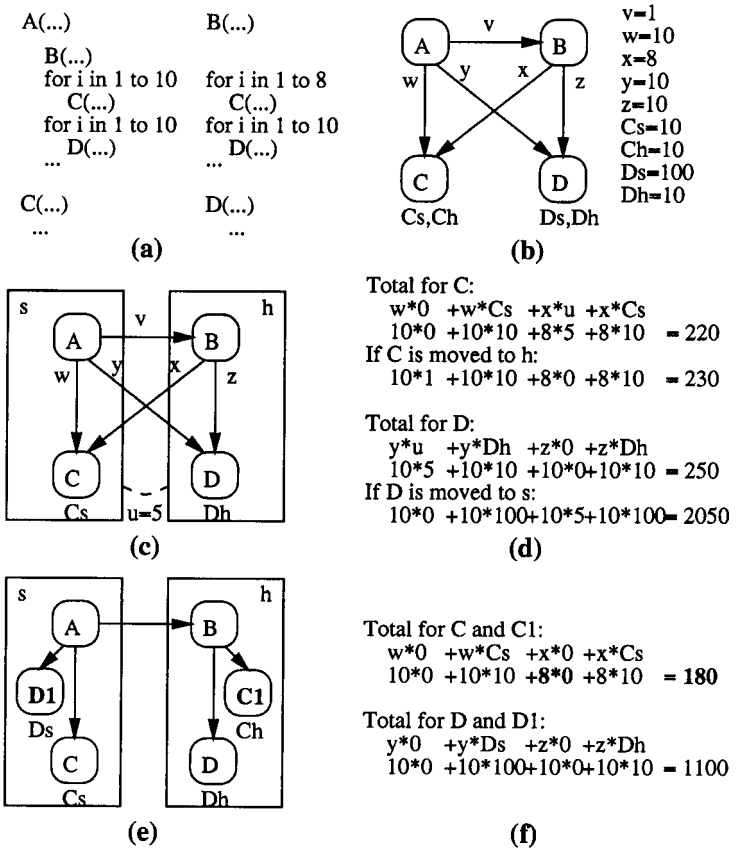

Fig. 1. Cloning example: (a) functional specification; (b) partially-annotated access graph; (c) first hardware/software partition; (d) time estimates for partition; (e) partition after cloning; (f) new time estimates after cloning.

Numerous closeness metrics were defined between operations for use during clustering.

Vulcan [Gupta and DeMicheli 1990] partitions a hierarchical control/dataflow graph's operations among hardware packages using iterative-improvement partitioning heuristics, such that size, I/O, and performance metric constraints are satisfied. Vulcan first partitions the graph at its coarsest granularity, decomposing certain nodes and repartitioning if constraints are not satisfied. CHOP [Kucukcakar and Parker 1991] partitions (manually) an arithmetic-level dataflow graph's operations among hardware packages. For each package, CHOP estimates metrics for a range of implementations of the operations on that package. The ranges for all packages are then combined into a few implementations that satisfy performance constraints. SpecSyn [Gajski et al.1994] partitions a procedural-level access graph's nodes among hardware packages to satisfy size, I/O, and performance constraints using a variety of heuristics, including clus-

tering and iterative improvement. A two-stage estimation approach (pre-estimation and online-estimation) is used, involving a sophisticated data structure that considers hardware sharing during size estimation [Vahid and Gajski 1995]. Multipar [Chen et al. 1994] reads a specification into an arithmetic-level control/dataflow graph and partitions the nodes among hardware packages using integer-linear programming or heuristics. Scheduling of the operations occurs simultaneously with partitioning.

All of the above efforts have hypothesized that functional partitioning among hardware packages was superior to the common approach of structural partitioning. Experiments in Vahid et al. [1996] support this hypothesis.

Numerous functional partitioning techniques have also been developed to address the problem of partitioning among a hardware/software architecture, to tradeoff the flexibility and low cost software with the speed of hardware. In Ernst et al. [1994], statement blocks are partitioned among a software and hardware processor using simulated annealing. In Gupta and DeMicheli [1993], statement threads are partitioned using a custom greedy-improvement heuristic. In Xiong et al. [1994], hierarchical clustering is used to merge statements together based on their suitability for hardware or software implementation. In Kalavade and Lee [1994], tasks in a dataflow graph are simultaneously partitioned and scheduled using a custom constructive heuristic. In Knudsen and Madsen [1996], basic blocks are partitioned among hardware and software using a dynamic programming algorithm, which takes communication into account. In Balboni et al. [1996], processes derived from a hierarchical state-machine are partitioned between a hardware and software processor, either manually or using hierarchical clustering; several formal transformations are incorporated such as parallelization. In Gajski et al. [1994], procedures and variables are partitioned among standard and custom processors, memories, and buses using a suite of automated heuristics (e.g., greedy improvement, group migration, clustering and simulated annealing), or interactively using hints and a spreadsheet-like display of metric values and constraints. Procedures can be decomposed into finer granularity using a technique called procedure exlining.

A survey of many of the above techniques can be found in Gajski et al. [1994] and Wolf [1994]. To our knowledge, the above techniques either: (a) treat each procedure as a single instance, or (b) expand each procedure-call into a dataflow graph node. Selectively cloning certain procedures has not been considered until now.

There has been cloning work performed in related domains. Procedure cloning has been performed in the compiler domain. Cooper et al. [1993] address the problem of interprocedural transformations. They focus specifically on using cloning to enable compiler optimizations that previously could only be obtained by procedure inlining, which is shown to result in exponential growth in code size and is thus undesirable. For example, the constant propagation optimization can eliminate the need for many conditional checks, and hence speed-up execution; but since procedures can be

called with different constants, inlining was previously used. Instead, their techniques evaluate the multiple calls to a given procedure under the light of a given optimization; clones are made only for those calls that might yield an optimization benefit. By being very selective about which procedures are cloned, they avoid exponential code increase while obtaining many of the optimization benefits of inlining. The developers of the object-oriented SELF language compiler [Chambers 1992] noted a problem that occurred when a single method (e.g., *Print*) could be invoked with different data types (e.g., characters and Cartesian point). The problem was that most object-oriented systems generated one compiled-code method for each source-code method; that one method would have to be general enough to handle different types, with this generality coming at the expense of performance. They therefore cloned source-code methods for each invoking type, so that each cloned method could then be inlined into the invoking code for better optimization—they referred to this technique as customization. Noting that such cloning could lead to exponential growth, they used dynamic compilation to ensure that only the currently used methods were cloned. In FUSE [Ruf and Weise 1991], basic blocks are cloned, or specialized in their terminology, for different input data values in order to be able to optimize those blocks for given values. However, the developers note that even different values may yield identical blocks, meaning that the cloning unnecessarily increased code size without yielding optimization improvements. Techniques were incorporated to remerge (or reuse, in their terminology) such unnecessary clones.

A second related domain is that of circuit partitioning [Johannes 1996], in which a network of gates is partitioned among physical packages with input/output (I/O) and size constraints. As circuit partitioning tends to be driven by the I/O constraints, much emphasis has been placed on minimizing the wires crossing between packages (i.e., the cut). An approach described in Hwang and Gamal [1995] takes a given partition and replicates (clones) nodes such that the cut is minimized. An improvement in Yang and Wong [1995] not only minimizes the cut, but also minimizes the number of clones, in order to reduce the final design size. The approach in Liu et al. [1995] focuses on two-way partitioning, striving to reduce interpackage signal delays, and thus improve performance by converting the given network into what they refer to as a replication graph, which includes two copies of every node, and then applying two-way partitioning to that graph.

3. PROBLEM DESCRIPTION

In this section we describe the functional specification input, the internal representation, and the computation and implementation models of our approach.

The functional specification is one repeating process written using sequential-programming constructs, such as those in VHDL or C, including loops, conditionals, and procedure calls. We later discuss extensions for

multiple processes, but this work focuses on systems with a small number of large processes, not a large number of small processes. Several reasons for partitioning a large process into smaller ones are given in Vahid et al. [1996]. We assume that a large process is composed of a number of procedures; if not, exlining techniques can be used to group a large process' statements into procedures [Vahid 1995]. Because procedures will form the objects to be partitioned among components, exlining typically seeks to only group statements that are closely related (e.g., share data or co-exist within a loop) to prevent creating too many partitioning objects. An example (partial) specification is shown in Figure 1(a).

We convert the specification into an Access-Graph (AG) representation-[Vahid and Gajski 1995; Vahid and Le 1996], which is very similar to a call-graph commonly used in software profiling, as shown in Figure 1(b). Each AG node represents a procedure and each AG directed edge represents a procedure call. The edge direction indicates the accessor and accessee, but *not* the direction of data flow, which can be in either or both directions. Note that the structure of the AG can be determined statically, i.e., without having to execute the behavior. Loops and conditionals are fully supported, since these can appear within any procedure node. We treat each large variable as a procedure and each read or write of that variable as a procedure call. For the purposes of estimation, mentioned later, we assume the specification is nonrecursive, meaning that the AG is acyclic. The AG is actually part of a larger format that includes component representation, called the System-Level Intermediate Format, or SLIF [Vahid and Gajski 1995].

We annotate each node 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. The above annotations must be determined using estimators and profilers (which we refer to as the pre-estimation stage); we use the SpecSyn estimators [Gajski et al. 1994; Gajski et al. 1997], whose details are beyond the scope of this paper. We have developed equations that quickly combine these annotations (during a stage we call online estimation) for a given partition to compute size, I/O, and execution times including communication; these equations are described in Vahid and Gajski [1995], and a special technique for quickly estimating hardware size while considering hardware sharing among procedures is described in Vahid and Gajski [1995]. A very small subset of annotations is shown in Figure 1(b).

The computation model is one of executing each procedure sequentially. The main procedure (process) executes until it calls another procedure. Control is transferred to that procedure, which then executes to completion (perhaps calling other procedures) and returns control to the calling procedure. The main procedure then continues executing until completion, and then repeats. While such a sequential model is currently used, there is

nothing to prevent use of the cloning techniques with more advanced models that allow concurrent procedure execution; such a model would require more complex estimation equations, but the cloning techniques would not be affected. The user typically specifies a constraint on a single execution of the main procedure, though any number of procedures can be constrained.

The implementation model consists of one processor per part, where that processor may be a standard one, such as a microcontroller, or a custom one, such as obtained using synthesis. Partitioning consists of distributing procedures (AG nodes) among parts. All procedures on a single part will be executed by a single processor on that part. Ideally, partitioning groups procedures that call one another, so that most calls occur within a single processor. Such internally called procedures can be implemented in many different ways, such as a control subroutine, inlined, a distinct datapath module, or even a concurrent processor. Of course, there will be some procedure calls between parts, meaning that each processor must wait until one of its procedures is called and then must activate that procedure. Interpart communication can be accomplished by creating one bus for each AG edge that is cut between parts, plus some handshaking lines with each bus to start the called procedure and to indicate procedure completion. Alternatively, we can create a single bus for all interpart communication, passing the address of the called procedure followed by any parameter data over the bus; that approach is the focus of Vahid [1997]. In this paper experimental data was obtained using the former, edge-crossing, approach. Once again, if the implementation model was extended for multiple input processes, and hence for multiple processors per part or concurrently-executing processors across parts, the estimation equations would need to be modified, but the cloning techniques would remain the same.

Each part is characterized for partitioning by a type and by I/O and size constraints. A single chip may contain numerous parts. (A much more detailed characterization is used by the estimators that determine the annotations described above; but during partitioning, we already have the annotations.) The type is used to select the corresponding node annotations for a given partition. The I/O and size constraints are compared to estimated values obtained using the above-mentioned estimation equations. The goal of partitioning is to minimize an *objective function* that computes a weighted sum of I/O, size, and execution-time constraint violations.

4. CLONE/UNCLONE TRANSFORMS

In this section we define the clone transformation and provide an algorithm, define an unclone transformation that is required by our heuristics, and contrast cloning with inlining.

4.1 Cloning

We introduce the cloning transformation through a simple example. Consider the AG of Figure 2(a). *Node* has two accessors, *Accessor*1 and

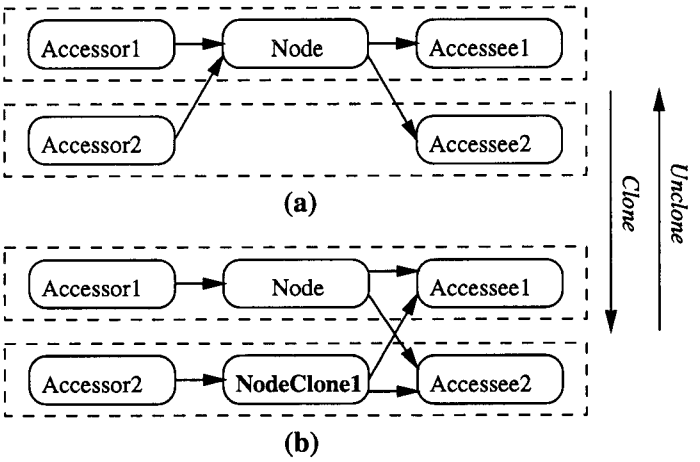

Fig. 2. Cloning an AG node.

*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*. Also, *NodeClone*1 accesses the same nodes (*Accessee*1 and *Accessee*2) that *Node* accesses. 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 for *NodeClone*1 and *Accessee*2 for *Node*, 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 originally n had a fanin of 1, cloning need not be performed because a is already the sole accessor of n.

Algorithm 4.1 provides an algorithm for cloning an AG node n for an accessor a:

Algorithm 4.1. **Clone**(AG, n, a)

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

To determine the complexity of cloning, note that each step can be performed in constant time, except for copying the node's outgoing edges, which is of order $O(|n.outedges|)$; this is the complexity of cloning. Although the number of possible edges in a directed acyclic graph is $O(n^2)$, and hence the amortized number of outgoing edges of a node is $O(n^2/n) = O(n)$, an AG with n outgoing edges per node would represent an absurd specification. In particular, this would mean that every procedure calls nearly every other procedure. Clearly, people do not write specifications in this manner. Instead, procedures serve to modularize a program, meaning that each procedure calls only a small subset of other procedures; our experiments on several examples indicate this subset size is usually less than seven [Vahid and Le 1996]. Therefore, the number of outgoing edges is usually a constant, so the amortized complexity of cloning for nondegenerate examples is a constant.

We currently allow cloning of nodes representing procedures only, not variables, even though both node types can have fanin > 1. Cloning of a variable node prevents 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 the node should be allowed.

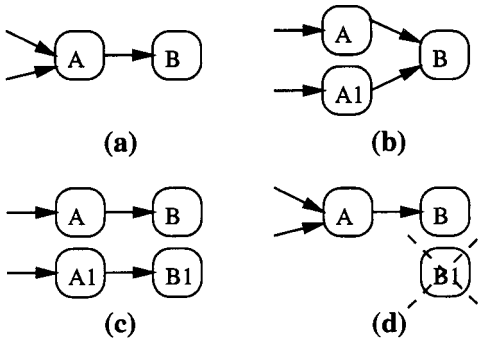

Fig. 3. Sequence leading to a stranded node.

4.2 Uncloning

Uncloning is required by our heuristics. 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}.
- 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 has 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 $Accessee2Clone$ will 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), $B1$ will be stranded. Thus, uncloning must delete such stranded nodes. Deleting a stranded node may yield further stranded nodes, which must also be deleted.

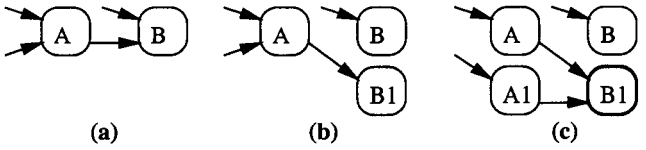

Fig. 4. Sequence leading to a clone with multiple accessors.

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 another clone of n. We use the terminology of uncloning a node *to* another node, i.e., we unclone n_{clone} to 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 found easily by looking at the clone's incoming edge. 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, we 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 any identical node, 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. An uncloning algorithm first redirects n_{clone}'s incoming edges to point to n_{ident}. Next, it passes n_{clone} to a function $Delete$. This function removes the given node and all outgoing edges, and then checks each accessee of outgoing edges; if an accessee is now stranded, the $Delete$ function recursively calls itself with the accessee.

4.3 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 [Hall et al. 1996], 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, such as a control subroutine, a custom processor, a datapath functional unit, or as inlined. Also, consider the case of having multiple clones or inlined procedure contents on the same part. With multiple clones, we can easily modify size estimates to consider each clone only 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 nontrivial task.

5. 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. We'll see that performance estimation techniques need not be modified (though the estimates themselves will of course be different).

5.1 Modifying Size Estimation

There are two methods for estimating size: weight based, and design based. Design-based methods [Vahid and Gajski 1995] 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 5.1. 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 5.1. **SizeEstBefore**(P)

```
for each n ∈ P.N loop
   P.size + = n_i.size
end loop
```

Modifying such size estimation to take clones into consideration means that we must use same-base nodes only once for a given part. A modified size estimation algorithm is given in Algorithm 5.2. 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 require that we keep a list of bases along with a count of the number of instances of each base; when a node is added or deleted, its base 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 5.2. **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
```

5.2 Modifying I/O Estimation

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

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 are cut. However, those edges point to the same node, so I/O estimation counts only 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 sees two cut edges that point to different nodes, so each edge contributes to the I/O estimate. However, as discussed above, we know that clones on the same part are always 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 5.3 . 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 and the accessee's part.

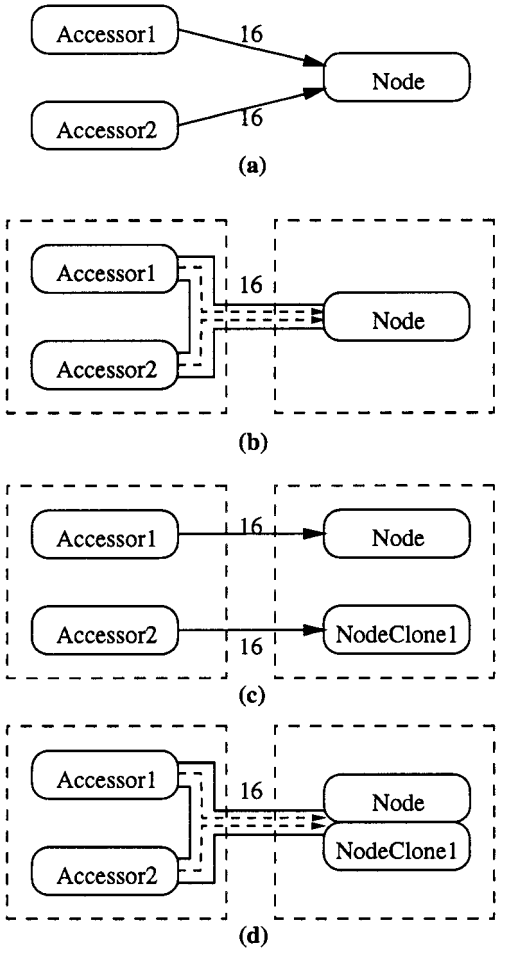

Fig. 5. Modifying I/O estimation for cloning.

Algorithm 5.3. **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
```

The above assumes I/O implementation using the cut-edges approach described in Section 3. If using the Vahid [1997] single interpart bus approach, then the number of I/O for interpart communication is fixed, but the access to external ports is still computed using the above technique.

5.3 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. Equations and further discussion can be found in Vahid and Gajski [1995]. Those equations assume sequential execution of the procedures—future work will include extending the equations to account for possible forking, and hence concurrent procedure execution and possible bus contention.

Because a clone's annotations are 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.

6. CLONING HEURISTICS

Now that we have defined the clone and unclone transformations and discussed how estimation techniques can be modified for the existence of clones, we can discuss various heuristics for cloning procedures before, during, or after partitioning.

In our approach to system design, there are three tasks to be performed:

(1) *allocation*: selecting the components (which may coexist on one chip), such as standard processors, microcontrollers, and ASIC blocks, on which to implement the system's functions;

(2) *partitioning*: assigning the system's functions among allocated components; and

(3) *transformation*: applying modifications to the system's functions without changing their input/output relationships. Such transformations include cloning, inlining, exlining, process merging, loop unrolling, etc.

All these tasks seek to minimize the value of the system's objective function. These tasks are highly interdependent; applying one first affects the possible solutions of another. We can approach these tasks either sequentially or simultaneously, and our cloning heuristics include both approaches. In this work we assume allocation has already been done, and any other transformations are either already done or will be done later. Future work might attempt to more closely integrate allocation as well as other transformations.

We can classify cloning heuristics into three categories:

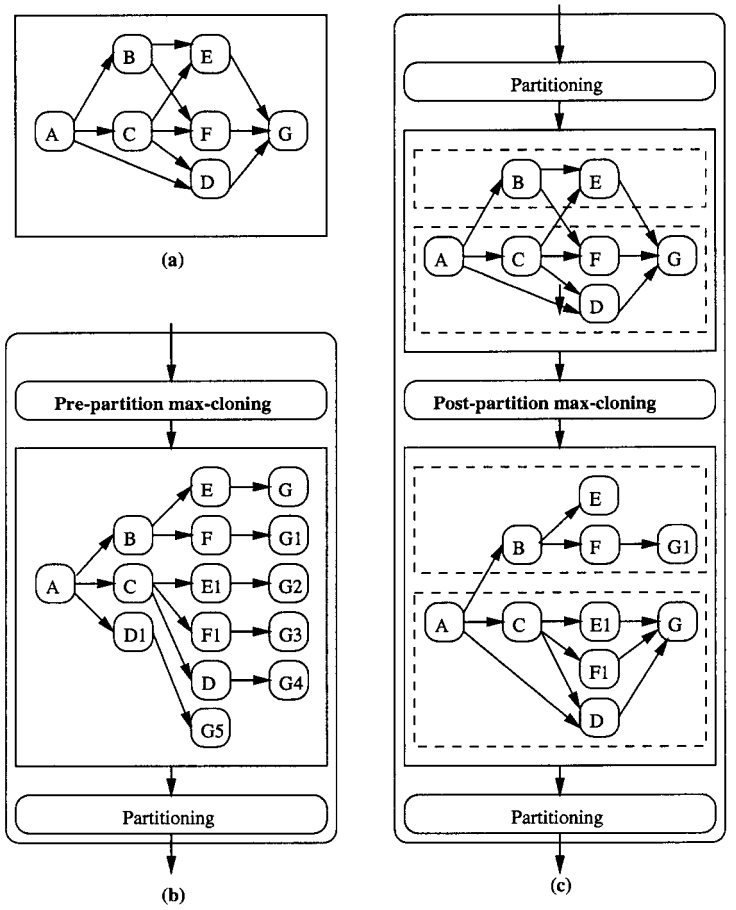

Fig. 6. Two cloning heuristics: (a) input AG; (b) prepartition max-cloning; (c) postpartition max-cloning.

(1) *Prepartition cloning*: we first clone some subset of procedures and then apply partitioning on the new AG.

(2) *Postpartition 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.

6.1 Prepartition Cloning

In *prepartition 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 initially appears 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 involves distinct data, each call requires a distinct node.

However, max-cloning results in a large increase in the number of AG nodes, which we find leads to inferior results, as discussed in Section 7. One might initially assume that this increase is bad because it causes less accurate estimates; however, Section 5 describes estimation modifications so that clones 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, the heuristics consider a subset of solutions, and the large increase in nodes means that the best partitions may never 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 after max-cloning. In general, the increase depends 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. So we can compute the number of nodes after max-cloning by using Algorithm 6.1. 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 the number of instances of this node that will appear after max-cloning. *total* keeps track of the total number of nodes.

Algorithm 6.1. **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.incomingedges 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) are visited first; those nodes are never cloned, since cloning is defined for a node and its accessor only. 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 limiting max-cloning to cases where the resulting number of nodes do not exceed some specified limit, without actually performing max-cloning.

A second prepartition cloning technique is best-cloning. In *prepartition best-cloning*, we attempt to predict which clones of nodes for accessors will best improve the cost after partitioning, much as clustering uses closeness metrics to predict which node groupings yield the best final partition [Gajski et al. 1994]. 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 interpart 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 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 prepartition best-cloning.

6.2 Postpartition 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 yields an improvement, then partitioning probably 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 be 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 is seen only after repartitioning.

In *postpartition best-cloning*, we again attempt to predict which clones of nodes for accessors improve the cost after repartitioning best. Of course, now that we have an initial partition, we have more information than prepartition 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 prepartition cloning).

6.3 Integrated Partitioning and Cloning

Many iterative-improvement partitioning heuristics make thousands of changes to a given partition, usually by moving a node from one part to another, using a control strategy that overcomes local cost minima without making excessive moves. A third approach to cloning, different from prepartition and postpartition cloning, modifies the definition of a "change" from just a node move, to either a move, clone, or unclone.

The simulated annealing heuristic is a popular one, and its modification is straightforward. The modified heuristic is shown in Algorithm 6.2. In the unmodified version of the heuristic, a function $RandMove$ is called in the inner loop. In the modified version, a function $RandChange$ is called. The function has three parameters in addition to 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 Section 4. 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 find that good results are obtained using clone/unclone probabilities that are small relative to the move probability (e.g., 0.05 each).

Algorithm 6.2. **Simulated annealing with cloning**

$temp$ = initial temperature
$cost$ = Objfct(P)
while not Frozen **loop**
  while not Equilibrium **loop**
    $Ptentative$ = **RandChange**(P, f_{move}, f_{clone}, $f_{unclone}$)

```
      cost_tentative = Objfct(P_tentative)
      Δcost = cost_tentative − cost
      if (! Reject(Δcost, temp)) then
        P = P_tentative
        cost = cost_tentative
      end if
    end loop
    temp = DecreaseTemp(temp)
  end loop
```

The heuristic begins with a high initial temperature and a cost of the initial partition. It then begins an inner loop, in which it generates a random change to the partition using *RandChange*, and determines the amount of change in cost, $\Delta cost$. *Reject* uses this amount and the current temperature to determine whether to accept or reject the change; improvements are always accepted, while other changes are frequently accepted at high temperatures but rarely at low ones. When some number of inner loop executions, or changes, has failed to yield improvement for some time, the heuristic is said to have reached *equilibrium*. The heuristic then decreases the temperature and repeats the inner loop again, unless the temperature is so low that the heuristic has reached the *frozen* state, in which case the heuristic terminates. There are many extensions to the above basic heuristic, such as one that selects the node for the next change from the "neighbors" of the node of the previous change; see Lengauer [1990] for further details.

Simulated annealing is rather slow as a partitioning heuristic and some other heuristics show better results [Eles et al. 1996]. Nevertheless, because of its simplicity, simulated annealing provides a means for incorporating new transformations, and is quite independent of the details of the objective function (as seen above), and therefore is a useful heuristic to use while new transformations and objective functions are being introduced and modified.

6.4 Max-Uncloning

Regardless of the cloning heuristic, 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 *postpartition max-unclone* transformation, which unclones all same-base nodes on a single part through repeated application of the unclone transformation.

It should be clear that all of the above heuristics can be applied, regardless of the number and types of components among which a system is partitioned. Prepartition max-cloning, for example, transforms the access graph only, independent of components. Postpartition max-cloning clones nodes accessed by another part—the number and types of those other parts

	None				Premax-u				Postmax-u				Integ-u			
Example	HwSize	HwIO	SwSize	Exec	HwSize	HwIO	SwSize	Exec	HwSize	HwIO	SwSize	Exec	HwSize	HwIO	SwSize	Exec
ans	2614	80	520	700	2849	76	483	425	2396	80	507	564	2533	80	478	398
ether	3753	74	1658	3767	10658	159	1019	2248	1238	80	2018	3237	2795	80	1807	3236
fuzzy	16405	180	15500	9530	16405	180	15500	9530	16405	180	15500	9530	16405	180	15500	9530
itv	12202	186	6341	13237	12402	198	6330	12853	12503	203	6293	11637	12565	197	6270	12057
mwt	4800	86	478	2266	4976	73	747	748	4954	99	344	760	4976	118	739	760
Avg % change	0.0%	0.0%	0.0%	0.0%	39.7%	20.2%	2.1%	-29.9%	-13.9%	6.5%	-1.9%	-22.4%	-4.4%	10.2%	10.9%	-26.5%

(a)

Example	None	None2	Premax	Postmax	Integ
ans	79	289	86	170	96
ether	75	258	99	180	109
fuzzy	66	233	66	149	61
itv	119	469	128	280	124
mwt	50	201	67	115	59

(b)

Example	None	Premax	Premax-u	Postmax	Postmax-u	Integ	Integ-u
ans	45	52	46	46	46	46	46
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

(c)

Fig. 7. Examples: (a) metric values; (b) runtimes; (c) number of nodes.

are of no concern. Max-uncloning always merges clones of a single part to one, regardless of the component type (e.g., hardware or software).

7. EXPERIMENTS

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

Figure 7 summarizes results of five examples. The examples include a telephone-answering machine (*ans)*, an Ethernet coprocessor (*ether*), a fuzzy-logic controller (*fuzzy*), an interactive TV processor (*itv*), and a microwave transmitter controller (*mwt*).

After converting each example to an AG and extensively annotating the AG with estimations obtained using estimators from UC Irvine's SpecSyn tool [Gajski et al. 1994], we partitioned each example between a hardware/software architecture consisting of an 8086 processor and a Xilinx XC4000-series FPGA. Each example has 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 Gajski et al. [1994] and Vahid and Gajski [1995]. 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.

We applied five heuristics to each example. *None* represents results without any cloning, using simulated annealing. The cooling schedule parameters include a starting temperature of 50, a stopping temperature of 1, a temperature reduction factor of 0.93, an equilibrium condition of 200 changes without improvement, and an acceptance function as described in Gajski et al. [1994]. 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) having parameters of 75, 1, 0.97, and 300; *none2* represents results using that schedule. *Premax* is prepartition max-cloning, *postmax* is

postpartition 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. After each heuristic was applied, postpartition max-uncloning was performed to merge same-part clones—such uncloning is denoted using a *-u* in the figure.

Figure 7(a) provides the design metric values of hardware size (*HwSize*) in gates, hardware input/output pins (*HwIO*), software size (*SwSize*) in bytes, and execution time (*Exec*) in cycles. The average percent improvements over results obtained without any cloning (i.e., results of *none*) are also shown. *none2* yields nearly no improvements over *none*, so results are not shown. As expected, all three cloning techniques obtained good reductions in execution time on several examples. We expect that such reductions would come at the expense of some penalties in the other metrics. For example, *integ-u* yields minor increases in hardware I/O and software size (though all constraints are still met). Surprisingly, though, *postmax-u* obtains its execution-time improvements with nearly no penalty on the other metrics. As expected, *Premax-u* yields very large penalties in hardware size. We have omitted results for postpartition best-cloning because of the (rather surprising) fact that it does not result in improvement for any of our examples. Apparently, after partitioning, no single clone will reduce cost; instead, a sequence of clones and moves are required to escape a local minimum. Future work might focus on finding such sequences. Figure 7(b) provides the runtimes for the five heuristics on a 166MHz Pentium. Note that *postmax* is nearly double the others because partitioning must be run twice, once before cloning and once after cloning. Finally, Figure 7(c) provides the number of nodes for each heuristic. Note that *premax* yields big increases in the number of nodes for some examples, indicating that those examples had a reconvergent procedure-calling hierarchy.

To further evaluate the cloning heuristics, we automatically generated 14 other examples, ranging in size from 10 to 130 nodes, using the techniques described in Vahid and Le [1996]. Figure 8 summarizes the results. Figure 8(a) shows the percentage improvement in the cost function (the weighted sum described above) over no cloning (*none*). Note that *premax* does well for very small examples, up to 30 nodes, but then begins to perform very poorly for bigger examples, often resulting in a nearly 200% cost penalty. *Postmax* and *Integ*, in contrast, consistently improve the cost, or at least make no change to the cost. Since not all examples have a procedure-call hierarchy that will necessarily benefit from cloning, we expect that some examples will have no change in cost. Figure 8(b) shows the runtimes for the heuristics. Although *postmax* runs partitioning twice, *premax* actually requires more time on large examples, since it results in such large numbers of nodes to be partitioned. Finally, Figure 9 provides the numbers of nodes for each heuristic. The number of nodes for *none* always equals the original number, so it is not shown. *Premax*'s number of nodes becomes so

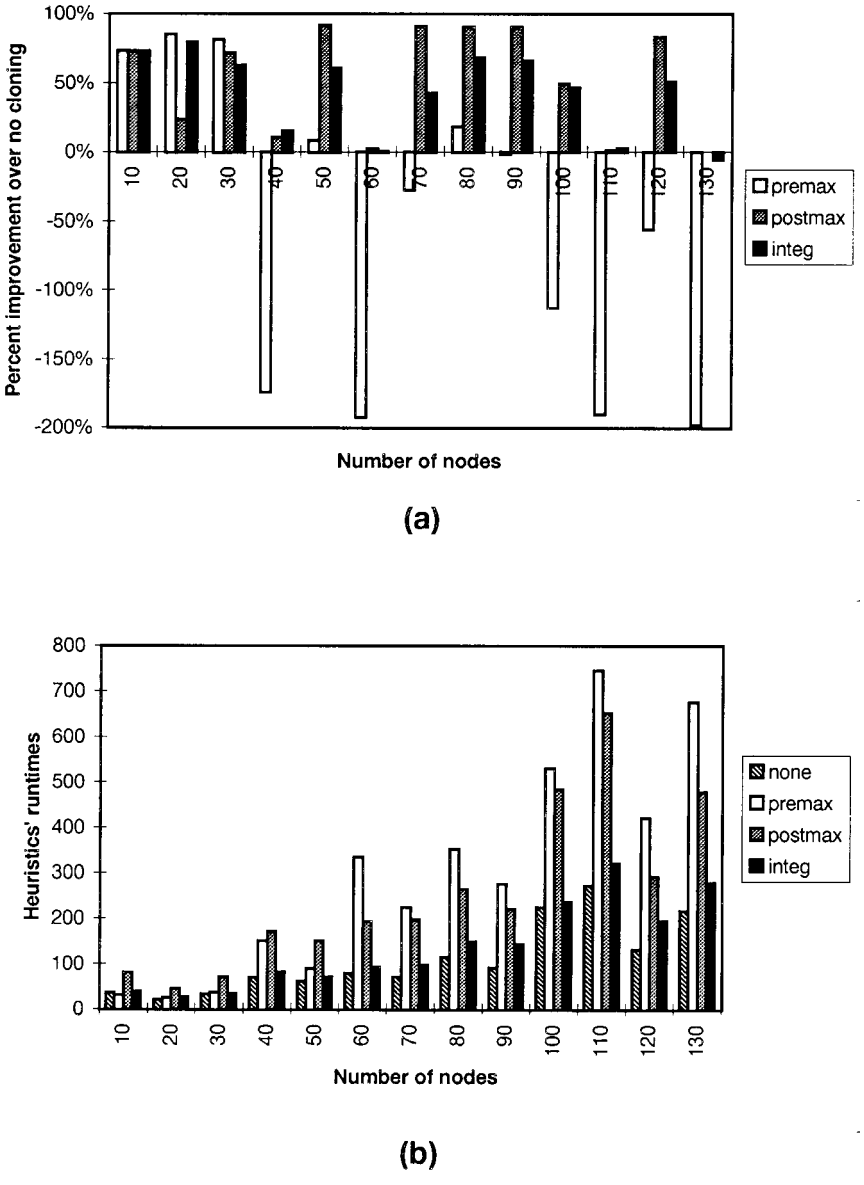

Fig. 8. Generated examples: (a) cost improvements; (b) runtimes.

large as to literally be off the chart, so it is not shown beyond 40 original nodes. For example, numbers for examples 100 through 130 are 810, 922, 741, and 998, respectively.

None and *none2* yield identical costs, implying that partitioning alone could not yield the improvements that cloning does. We also evaluated *integ* with clone and unclone probabilities of 0.1 each, and with move probability of 0.8. Its results were inconsistent. We also tried performing

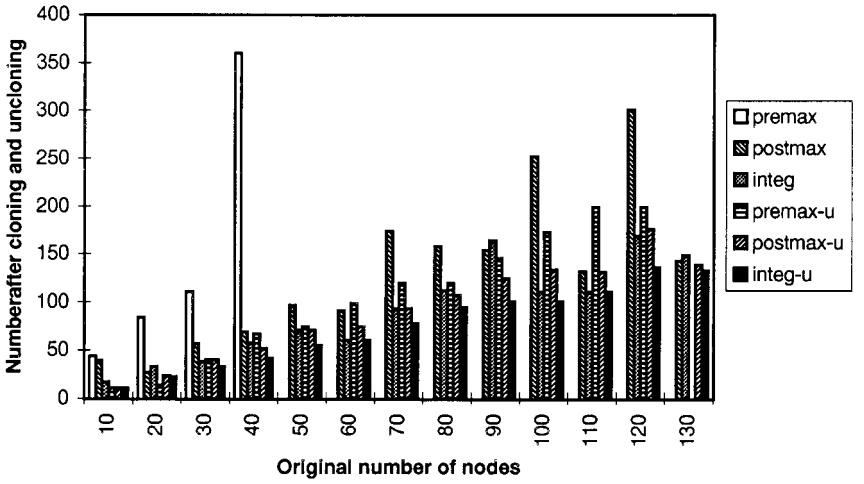

Fig. 9. Generated examples: numbers of nodes.

two iterations of postpartition max-cloning, i.e., first partitioning, then max-cloning, then repartitioning, then max-cloning again, followed by another repartitioning. Results were again inconsistent.

Limitations: We have not yet found a cloning heuristic that yields the best costs as well as the best runtimes and final number of nodes compared to the other heuristics. However, it is important to note that both *integ* and *postmax* yield much better costs than no cloning at all. A prepartition or postpartition best-cloning technique might find the best costs and best final number of nodes, but their runtimes will likely be longer; nevertheless, they may be worth looking into. A second limitation is that, while integrating partitioning and cloning for simulated annealing is straightforward, integrating the two for other partitioning heuristics might not be. For example, there is no obvious or simple method for integrating cloning with the Kernighan/Lin partitioning heuristic [Kernighan and Lin 1970], whose extensions have proven to run extremely fast and to yield low-cost results.

In summary, both the *postmax* and *integ* cloning heuristics yield excellent cost improvements over partitioning without cloning. *Postmax* yields the best costs, while *integ* yields the best runtimes and final number of nodes. The choice of a heuristic thus depends on the relative importance of those three factors. The results also demonstrate the inferiority of *premax*; because *premax* is akin to creating a dataflow graph representation versus an access graph. Such results are significant to those considering using a dataflow graph during system design to represent highly-procedural functional specifications.

8. CONCLUSIONS

We have demonstrated that significant improvements can be gained by incorporating procedure cloning with functional partitioning. We have

shown that postpartition max-cloning and integrated partitioning 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 (prepartition max-cloning). The success of cloning makes it an essential task in any system-level functional partitioning tool.

REFERENCES

ANTONIAZZI, S., BALBONI, A., FORNACIARI, W., AND SCIUTO, D. 1994. A methodology for control-dominated systems codesign. In *Proceedings of the International Workshop on Hardware-Software Co-Design*. 2–9.

BALBONI, A., FORNACIARI, W., AND SCIUTO, D. 1996. Partitioning and exploration strategies in the Tosca co-design flow. In *Proceedings of the International Workshop on Hardware-Software Co-Design*. 62–69.

CHAMBERS, C. D. 1992. The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages. Ph.D. Dissertation. Stanford University, Stanford, CA.

CHEN, Y., HSU, Y., AND KING, C. 1994. MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures. *IEEE Trans. Very Large Scale Integr. Syst. 2*, 1 (Mar.), 21–32.

COOPER, K., HALL, M., AND KENNEDY, K. 1993. A methodology for procedure cloning. *Comput. Lang. 19*, 2.

ELES, P., PENG, Z., AND DOBOLI, A. 1992. VHDL system-level specification and partitioning in a hardware/software co-synthesis environment. In *Proceedings of the International Workshop on Hardware-Software Co-Design*. 49–55.

ELES, P., PENG, Z., KUCHCINSKI, K., AND DOBOLI, A. 1996. Hardware-software partitioning with iterative improvement heuristics. In *Proceedings of the International Symposium on System Synthesis*. 71–76.

ERNST, R., HENKEL, J., AND BENNER, T. 1994. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test* (Dec.), 64–75.

GAJSKI, D. D., VAHID, F., NARAYAN, S., AND GONG, J. 1994. *Specification and Design of Embedded Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ.

GAJSKI, D., VAHID, F., NARAYAN, S., AND GONG, J. 1998. Specsyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Trans. Very Large Scale Integr. Syst. 6*, 1, 84–100.

GUPTA, R. K. AND DE MICHELI, G. 1993. Hardware-software cosynthesis for digital systems. *IEEE Des. Test 10*, 3 (Sept.), 29–41.

GUPTA, R. AND DEMICHELI, G. 1990. Partitioning of functional models of synchronous digital systems. In *Proceedings of the International Conference on Computer-Aided Design*. 216–219.

HALL, M., ANDERSON, J., AMARASINGHE, S., MURPHY, B., LIAO, S., BUGNION, E., AND LAM, M. 1996. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Comput. 29*, 12 (Dec.), 84–89.

HWANG, L. AND GAMAL, A. E. 1995. Min-cut replication in partitioned networks. *IEEE Trans. CAD 14* (Jan.), 96–106.

JOHANNES, F. 1996. Partitioning of VLSI circuits and systems. In *Proceedings of the 33rd Conference on Design Automation*.

KALAVADE, A. AND LEE, E. 1994. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proceedings of the International Workshop on Hardware-Software Co-Design*. 42–48.

KERNIGHAN, B. AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* (Feb.).

KNUDSEN, P. AND MADSEN, J. 1996. PACE: A dynamic programming algorithm for hardware/software partitioning. In *Proceedings of the International Workshop on Hardware-Software Co-Design*. 85–92.

KÜKÇAKAR, K. AND PARKER, A. C. 1991. CHOP: A constraint-driven system-level partitioner. In *Proceedings of the 28th ACM/IEEE Conference on Design Automation* (DAC '91, San Francisco, CA, June 17–21, 1991), A. R. Newton, Ed. ACM Press, New York, NY, 514–519.

LAGNESE, E. D. AND THOMAS, D. E. 1989. Architectural partitioning for system level design. In *Proceedings of the 26th ACM/IEEE Conference on Design Automation* (DAC '89, Las Vegas, NV, June 25–29, 1989), D. E. Thomas, Ed. ACM Press, New York, NY, 62–67.

LAGNESE, E. AND THOMAS, D. 1991. Architectural partitioning for system level synthesis of integrated circuits. *IEEE Trans. Comput.-Aided Des. 10* (July), 847–860.

LENGAUER, T. 1990. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., New York, NY.

LIU, L., KUO, M., HU, T., AND CHENG, C. 1995. Performance-driven partitioning using a replication graph approach. In *Proceedings of the European Conference EURO-DAC '95 with EURO-VHDL '95 on Design Automation* (Brighton, UK, Sept. 18–22), G. Musgrave, Ed. IEEE Computer Society Press, Los Alamitos, CA, 206–210.

ORAILOGLU, A. AND GAJSKI, D. 1986. Flow graph representation. In *Proceedings of the Conference on Design Automation*. 503–509.

RUF, E. AND WEISE, D. 1991. Using types to avoid redundant specialization. *SIGPLAN Not. 26*, 9 (Sept.), 321–333.

THOMAS, D., ADAMS, J., AND SCHMIT, H. 1993. A model and methodology for hardware/software codesign. *IEEE Des. Test*, 6–15.

VAHID, F. 1997. I/O and performance tradeoffs with the FunctionBus during multi-FPGA partitioning. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays* (FPGA). 27–34.

VAHID, F., LE, T., AND HSU, Y. 1996. A comparison of functional and structural partitioning. In *Proceedings of the International Symposium on System Synthesis*. 121–126.

VAHID, F. AND LE, T. 1996. Towards a model for hardware and software functional partitioning. In *Proceedings of the International Workshop on Hardware-Software Co-Design*. 116–123.

VAHID, F. AND GAJSKI, D. D. 1995. Incremental hardware estimation during hardware/software functional partitioning. *IEEE Trans. Very Large Scale Integr. Syst. 3*, 3 (Sept.), 459–464.

VAHID, F. 1995. Procedure exlining: A transformation for improved system and behavioral synthesis. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15, 1995), P. G. Paulin and F. Mavaddat, Eds. ACM Press, New York, NY, 84–89.

VAHID, F. AND GAJSKI, D. 1995. SLIF: A specification-level intermediate format for system design. In *Proceedings of the European Conference on Design and Test* (EDTC). 185–189.

WOLF, W. 1994. Hardware-software co-design of embedded systems. *Proc. IEEE 82*, 7 (July), 967–989.

XIONG, X., BARROS, E., AND ROSENSTIEL, W. 1994. A method for partitioning UNITY language in hardware and software. In *Proceedings of the European Conference on Design Automation* (EURO-DAC '94, Grenoble, France, Sept. 19–23, 1994), J. Mermet, Ed. IEEE Computer Society Press, Los Alamitos, CA, 220–225.

YANG, H. H. AND WONG, D. F. 1995. New algorithms for min-cut replication in partitioned circuits. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD-95, San Jose, CA, Nov. 5–9, 1995), R. Rudell, Ed. IEEE Computer Society Press, Los Alamitos, CA, 216–222.

Received: August 1996; revised: January 1997; accepted: November 1997