Partitioning Sequential Programs for CAD Using a Three-Step Approach

FRANK VAHID
University of California, Riverside

Many computer-aided design problems involve solutions that require the partitioning of a large sequential program written in a language such as C or VHDL. Such partitioning can improve design metrics such as performance, power, energy, size, input/output lines, and even CAD tool runtime and memory requirements, by partitioning among hardware modules, hardware and software processors, or even among time-slices in reconfigurable computing devices. Previous partitioning approaches typically preselect the granularity at which the program is partitioned. In this article, we define three distinct partitioning steps: procedure determination, preclustering, and N-way partitioning, with the first two steps focusing on granularity selection. Using three steps instead of one can provide for a more thorough design space exploration and for faster partitioning. We emphasize the first two steps in this article since they represent the most novel aspects. We illustrate the approach on an example, provide results of several experiments, and point to the need for future research that more fully automates the three-step approach.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*; *Hardware description languages*; *Optimization*

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Partitioning, hardware/software partitioning, behavioral partitioning, functional partitioning, system level partitioning

1. INTRODUCTION

Sequential programs serve as input to many computer-aided design (CAD) tools. A sequential program may be captured in a language such as C, C++, Java, VHDL, Verilog, or SystemC. Partitioning a sequential program into several smaller communicating programs has become an important solution for many problems addressed by CAD tools.

Numerous hardware/software codesign tools, for example, partition a program among a microprocessor and customized digital logic to improve performance.[1] Recently, researchers have found that such partitioning can also reduce power or energy [Henkel 1999; Wan et al. 1998; Stitt and Vahid 2002].

[1]Please see Balboni et al. [1996], Eles et al. [1996], Ernst et al. [1994], Gupta and De Micheli [1993], Knudsen and Madsen [1996], and Gajski et al. [1998].

Author's address: F. Vahid, Dept. of Computer Science and Engineering, University of California, Riverside, CA 92521; email: vahid@cs.ucr.edu, http://www.cs.ucr.edu/~vahid.

Partitioning a program before running behavioral or register-transfer synthesis can also yield numerous advantages, such as reduced synthesis run-time, reduced power or energy, improved performance, simplified physical design, or satisfaction of packaging or module input/output constraints. For example, we have demonstrated 40% power reduction with little change in performance, by partitioning before synthesis [Hwang et al. 1999], obtained because several smaller mutually exclusive processors consume less power per operation than one big processor. Likewise, we have demonstrated reductions in otherwise long synthesis run-times and improved input/output constraint satisfaction obtained through such partitioning [Vahid et al. 1998].

Reconfigurable computing tools also partition a program to make better use of a given amount of a field-programmable logic device through time-multiplexing the mapping of functionality onto the device [Callahan et al. 2000].

We can thus see that partitioning a sequential program is an important problem in many areas of CAD. There has been extensive research in partitioning multiple concurrent programs, called processes or tasks, among multiple processing elements [Kalavade and Lee 1994; Wolf 1997], but that problem emphasizes scheduling and is thus quite different from partitioning a single program. Approaches that focus instead on a single sequential program typically predetermine the granularity at which the program will be partitioned, using atomic operations such as arithmetic/control operations [Lagnese and Thomas 1991], finite-state machine states [Hwang et al. 1999], basic blocks [Ernst et al. 1994; Balboni et al. 1996], or even procedures [Gajski et al. 1998], listed in order of fine-grained to coarse-grained operation. Such approaches expand a program into those atomic operations, and then apply an N-way partitioning algorithm that assigns those operations among N groups, where what each group represents depends on the CAD problem being addressed.

Using such a one-step N-way partitioning approach with predetermined granularity limits the potential quality of partitioning results. Henkel and Ernst [1997] sought to overcome this limitation using a dynamically determined granularity, in which they hierarchically grouped blocks to form a hierarchical set of operations. Their N-way partitioning algorithm, simulated annealing, then moves randomly selected operations among hardware and software groups; some of those operations may be fine-grained, others coarse-grained.

Related work in the compiler domain [Hall et al. 1996; Ruf and Weise 1991; Arnold et al. 2000; Cooper et al. 1993] tends to have a very different focus, derived primarily from the fact that compilers are expected to have run-times on the order of seconds, not minutes or hours. Partitioning for CAD assumes that a much more thorough exploration of the solution space can be tolerated, since CAD tool run-times on the order of hours are common.

We introduce a novel three-step partitioning approach, highlighted in Figure 1, that overcomes the limitations of predetermined granularity. We tailor this approach specifically to sequential program partitioning, although the steps could be tailored to some other partitioning domains as well. A *procedure determination* step creates, copies, merges, or eliminates procedures, to find the best initial granularity for partitioning. A *preclustering* step pregroups

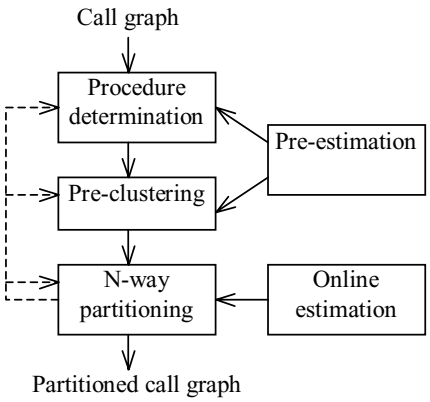

Fig. 1. Three-step partitioning overview.

certain very close procedures to prune inferior regions of the solution space. An *N-way partitioning* step assigns the remaining procedures to N groups. The first two steps focus on selecting the appropriate granularity. Our results show that those first two steps play a big role in the quality of results. As N-way partitioning has been widely studied, most of this article emphasizes the first two steps, and experimental results illustrate the impact that various procedure determinations and preclusterings can have on N-way assignment. We conclude by pointing to the need for further research into automatic integration of these three steps.

2. PROBLEM STATEMENT

We are given a single sequential program, consisting of hundreds to thousands of lines of code, describing a complex repeating sequential computation. This program may include typical sequential program features such as loop and branch statements, assignment statements, procedures, local and global variables, and so on. We do not currently consider programs with recursion or pointers, leaving those features for future work. Program execution starts with a main procedure, which proceeds to execute its statements sequentially.

For example, Figure 2(a) shows a small part of a 720-line microwave transmitter controller program. The key global variable and procedure declarations are shown, followed by part of the main program. The program first initializes and clears a liquid-crystal display (LCD), and then sequences through several modes. Each mode is a procedure that calls several other procedures. We use procedures and global variables as our starting granularity of operations for partitioning. We say starting granularity because our partitioning steps may decompose or merge these operations to favorably change the granularity. From this point on, we refer to global variables as procedures themselves, with variable accesses treated as procedure calls.

Partitioning assigns the procedures among N groups, where N can be 2 or more. Each group may represent functions to be mapped to a software processor, hardware processor, time-slice on a programmable logic device, or some other construct depending on the CAD problem being addressed. The

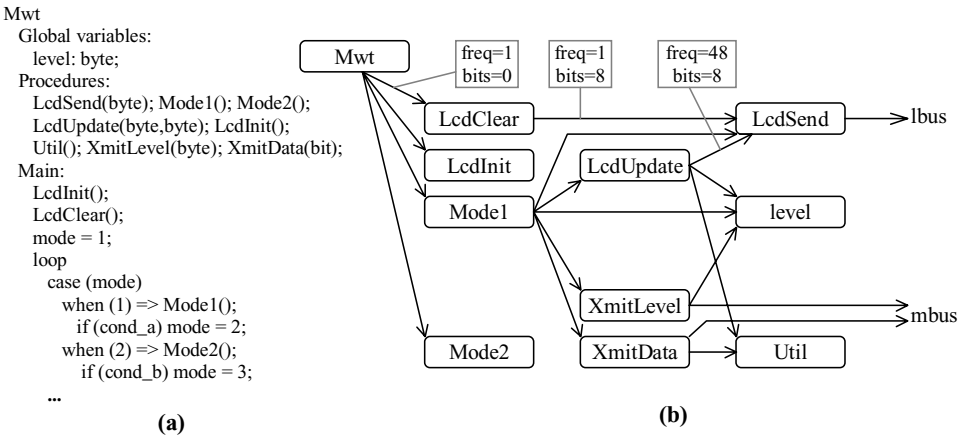

Fig. 2. Microwave transmitter example: (a) original sequential program; (b) initial call-graph with some annotations.

groups can each represent a different type, as is the case in hardware/software partitioning. These groups may exist on the same integrated circuit (IC) or separate ICs. Program execution after partitioning is the same as before, except that a procedure in one group that calls another procedure in a different group requires communication of control and data information between those groups. We do not restrict how procedures will be implemented internally in a group after partitioning is completed; they could be inlined, implemented as an independent component, and so on. Furthermore, subsequent scheduling can introduce concurrency, either among procedures within a group, or among procedures between groups, as long as data and timing dependencies are observed.

We first convert the sequential program into a call graph representation, where each node represents a procedure, and each edge represents a procedure call. Note that this model, unlike the more traditional dataflow graph model used in many earlier approaches, supports programs having procedures called from many different places in the program, with arbitrarily deep nesting of such calls. We can describe the call graph more specifically as follows.

The call graph is a directed acyclic graph $CG = <F,E>$. Each node in the set $F = \{f1, f2, \ldots, fn\}$ corresponds to a procedure, and each node $fi = \{S\}$, where S is the set of statements in that procedure. We refer to a node's statements as $fi.S$. Each edge in the set $E = \{e1, e2, \ldots, em\}, ei = <fsrc, fdst>, fsrc \in F$, $fdst \in F$, $fsrc \neq fdst$ corresponds to one or more calls from procedure *fsrc* to procedure *fdst*.

When the call graph serves as input to a CAD tool, that tool will typically heavily annotate the graph using profilers and other *pre-estimators*, and those annotations will be used to speed up *online estimators* used during N-way partitioning [Gajski et al. 1998], as illustrated in Figure 1. The details of such annotation and estimation are beyond the scope of this article. However, some basic annotations that are relevant to our work include an extended edge definition $ei = <fsrc, fdst, freq, bits>$, where *freq* is the frequency with which the procedure

fsrc calls *fdst* during a fixed time of program execution, and *bits* is the number of data bits transferred as parameters during each call. Figure 2(b) shows the initial call-graph for the earlier example. A few edge annotations are shown.

3. PARTITIONING STEP 1: PROCEDURE DETERMINATION

A key trade-off in partitioning is the selection of the granularity of the procedures input to N-way partitioning. Selecting fine-grained procedures results in a large number of procedures, whereas coarse-grained means a smaller number. For example, a hypothetical 1000-line program could be divided into 500 2-line procedures (let's call this Case 1) or 2 500-line procedures (Case 2). Fine-grained has the advantage of exposing more of the solution space to an N-way partitioning algorithm. For example, an algorithm that explores n^2 partitionings will examine 250,000 partitionings for Case 1 but only 4 for Case 2. In contrast, coarse-grained has the advantage of enabling powerful partitioning algorithms with high run-time complexity to still complete in reasonable run-times. For example, an n^2 algorithm requiring 100 milliseconds per partitioning will run for 25,000 seconds for Case 1 but only 0.4 seconds in Case 2. Coarse-grained also has the advantage of enabling extensive pre-estimation which in turn speeds up online estimation. For example, we can pre-estimate the run-time for Case 2's two procedures, and then simply add those times up during online estimation. However, we can do little pre-estimation for Case 1, since the run-time of a large set of 2-line procedures in one group will be mostly determined by interprocedural optimizations. This means online estimation will have to be even more complex to achieve decent accuracy, thus further adding to run-time for the fine-grained situation.

In earlier work [Gajski et al. 1998], we performed N-way partitioning directly on the initial call graph, but we then observed the significant dependency that the programmer's use of procedures had on quality. We thus sought to decrease this dependency. We developed a new partitioning step, which we called *procedure determination*, whose goal is to divide the sequential program into a set of procedures best suited for N-way partitioning. It generally strives to create procedures such that: (1) procedures are as coarse-grained as possible, to enable powerful N-way algorithms and extensive pre-estimation, and (2) statements are grouped into a procedure only if their separation would yield inferior solutions.

We now briefly describe the set of transformations that we have developed to support procedure determination. Additional transformations that form the core of compiler front-end processing, such as loop unrolling, constant propogation, and the like, could also be included in this step [Hall et al. 1996].

3.1 Procedure Inlining

Procedure inlining is a well-known transformation that replaces a procedure call by the procedure's contents. Inlining updates our call graph as follows, as illustrated in Figure 3(a).

Given a node $f1$ and a statement $s \in f1.S$ calling $f3$, replace s by $f3.S$ (replacing parameters appropriately), calling the new node $f1a$. For the edge ei connecting $f1$ and $f3$, reduce *ei.freq* by the frequency of s. If this reduction makes *ei.freq* zero, delete ei from the call graph. If deleting this edge results in $f3$ having no incoming edges, delete $f3$.

An advantage of inlining is that it can make granularity coarser, since inlined procedures may themselves be deleted, whereas calling procedures get more complex. Another more subtle advantage is *call differentiation*: estimations will be more accurate since different parameter values from different call locations can be differentiated for each inlined set of procedure statements. For example, a particular procedure might always be called from one location with parameter value 10 and from another location with value 10,000. When inlined, these are constants that can be propagated separately through the inlined procedure statements, yielding perhaps very different performance estimates.

A disadvantage of inlining is growth in code size. This occurs when a procedure is called from more than one location. This growth is even worse for deeply nested procedure call hierarchies, since the growth is multiplicative. For example, suppose procedure A calls B from 5 locations, B calls C from 5 locations, and C calls D from 5 locations. Assume all procedures had 10 statements originally. Then C will be of (approximate) size $10 + 5^*10 = 60$ after inlining D. B will be of size $10 + 5^*(60) = 310$, and A will thus be of size $10 + 5^*(310) = 15,500$ lines, much more than the original 40 lines. Furthermore, inlining prevents separation of procedures that perhaps should have been partitioned to separate groups.

Thus, inlining must be done reservedly. We currently assume inlining is done manually. Automated heuristics in the compiler community have focused on reducing procedure call overhead or on call differentiation while limiting code size growth [Arnold et at. 2000]. Inlining for partitioning has very different goals, so future work on heuristics for such inlining might include factors such as a called procedure's size, the static number of call locations, the dynamic frequency of calls, the nesting level of calls in the call hierarchy, the variation in parameter values among the calls, and the differences between the calling procedures themselves.

3.2 Procedure Cloning

The *procedure cloning* transformation makes a copy of a procedure for exclusive use by a particular caller. More specifically, cloning updates the call graph as follows, as illustrated in Figure 3(b).

Given a node $f3$ that has at least two incoming edges, and a node $f2$ that is the source of one of those edges e, copy $f3$ to a new node $f3a$, setting e's destination to $f3a$. Copy all of $f3$'s outgoing edges, and set the copies' source to be $f3a$.

This transformation is a compromise between inlining and not inlining. A multiply called procedure when inlined might lead to multiplicative growth in code size, but if not inlined might become a communication bottleneck. Cloning eliminates the bottleneck while avoiding excessive growth.

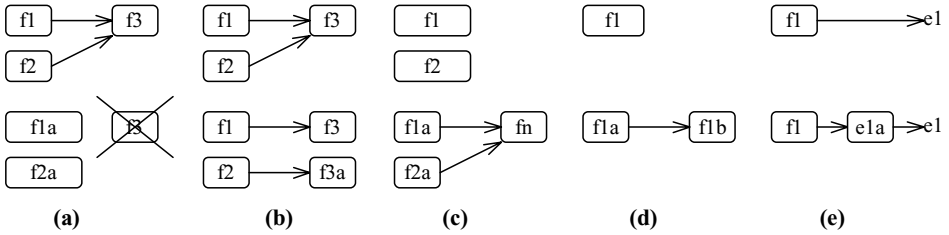

Fig. 3. Procedure determination: (a) inlining; (b) cloning; (c) redundancy exlining; (d) computation exlining; (e) port calling.

We can distinguish certain types of cloning heuristics during the granularity selection step. One is *max-cloning*, in which cloning is repeated until no procedure has more than one accessor; that is, all call graph nodes have only one incoming edge. If max-cloning is done after N-way partitioning, then we can restrict max-cloning to only those nodes with at least one same-part accessor and one different-part accessor; such postpartition max-cloning can find good cloning candidates, which can in turn yield improvements during a subsequent N-way partitioning. A second type of cloning is *best-cloning*, in which only those nodes deemed by some cost function as good cloning candidates are cloned. A good candidate would in general be small, and would be accessed by procedures that themselves have little else in common. In either case, after partitioning (including possible iterations among the three steps) is completed, clones in the same group would be uncloned (merged back together).

Cloning cannot be performed for nodes representing global variables, since the variable's value must be consistent across all accessors. More generally, any procedure with state should not be cloned (such as a procedure with a static local variable).

Cloning has been done in the compiler community for purposes of more accurate profiling information; see the above discussion on call differentiation [Cooper et al. 1993; Hall et al. 1996; Ruf and Weise 1991]. Our approach is different in its goals and thus its methods; details are to be found in Vahid [1999a].

3.3 Procedure Exlining for Redundancy Elimination

We have also developed a transformation called *procedure exlining*, which, as its name implies, is the inverse of inlining. We distinguish between two types of exlining: redundancy and computation exlining. Redundancy exlining seeks to replace two or more near-identical sequences of statements from different procedures by calls to one procedure [Vahid 1995]. It updates the call graph as follows, as illustrated in Figure 3(c).

Given $f1$ and $f2$ and a sequence of statements T, $T \subset f1.S, T \subset f2.S$, create a new procedure *fn* such that $fn.S = T$. Replace $f1$ by $f1a$, which is identical to $f1$ except that T is replaced by a single statement that calls *fn*; do the same for $f2$. Add an edge from $f1a$ to *fn*, and another from $f2a$ to *fn*.

Advantages and disadvantages of redundancy exlining are the inverse of those for inlining.

The above definition of redundancy exlining is actually too strict; most redundant sequences of statements will not match exactly, but rather approximately. Differences could be simply due to different variable names or inconsequential statement reorderings. Automatically detecting redundant statement sequences is a hard problem; we use a manual but computer-assisted approach. We first encode each statement by its type and accessed symbols, and then use an existing approximate pattern matching [Wu and Manber 1992] tool to find candidate matches. The user then determines whether candidate matches should be exlined, and manually creates a new procedure. Parameters and branches inside the new procedure's statements may be necessary to account for real differences between the approximate matching sequences of statements.

3.4 Procedure Exlining for Distinct Computation Isolation

This second type of procedure exlining seeks to divide a large sequence of statements into several smaller procedures such that statements within a procedure are tightly related and would thus never be separated in a good N-way partitioning solution. Computation exlining modifies the call graph as follows, as illustrated in Figure 3(d).

Given $f1$ and a sequence of statements T, $T \subset f1.S$, create a new procedure $f1b$ such that $f1b.S = T$. Replace $f1$ by $f1a$, which is identical to $f1$ except that T is replaced by a single statement that calls $f1b$. Add an edge from $f1a$ to $f1b$.

Computation exlining provides for finer granularity, at the expense of more nodes.

In our approach to computation exlining, we use statements as our unit; an approach at the arithmetic-operation level is described in Lagnese and Thomas [1991]. We first convert the statements to a tree, where each node represents a statement, and each nonleaf node represents a hierarchical statement such as a loop or procedure call. We add special edges to the tree to indicate nodes that can be legally grouped, and then define tree transformations for the addition of a new procedure.

The distinct-computation exlining problem is thus to insert procedure nodes into this tree such that a cost function is minimized. We use a weighted sum cost function consisting of several terms. *Procedure size* is the variation from a user-provided desired number of statements per procedure. *Control transfer* is the number of transfers of control to procedures over the entire tree, obtained from pre-estimation annotations. *Data transfer* is the amount of data transferred to or from procedures over the tree. *Hardware size* is the total size of hardware assuming each procedure is synthesized independently, thus encouraging grouping of statements that can share hardware.

We developed three heuristics for solving this problem. A naive heuristic simply inserts a procedure node for every hierarchical statement that is not already a procedure. A clustering heuristic merges closest nodes until a closeness threshold is no longer exceeded, where closeness is defined as a weighted sum of the above metrics defined between the two nodes. A simulated annealing heuristic inserts and deletes procedure nodes according to a controlled

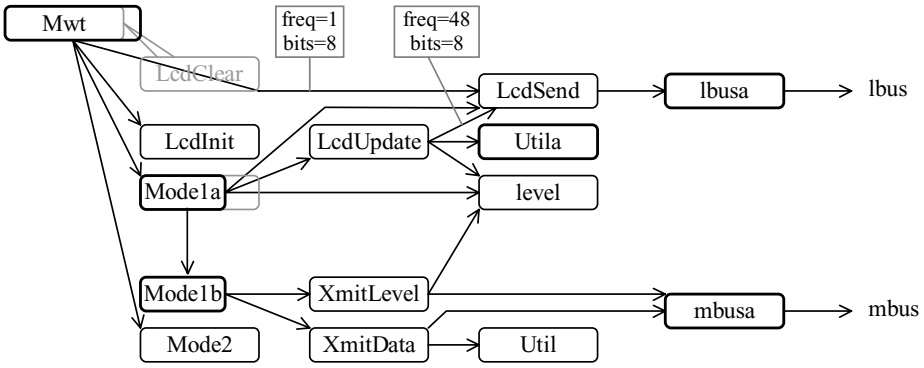

Fig. 4. Microwave transmitter example after the first partitioning step of procedure determination.

randomized process. The naive heuristic was obviously fastest but gave poor results. Clustering and simulated annealing both yielded good results, with the former being faster but at the expense of lower quality. Further exlining details can be found in Vahid [1995].

3.5 Port Calling

The port calling transformation seeks to allow us to distribute external input/output (I/O) ports among groups irrespective of which procedures access those ports. It accomplishes this by inserting new procedures responsible for I/O port access, and replacing direct port access in procedures by calls to these new procedures. Port calling modifies the call graph as follows, as illustrated in Figure 3(e).

Modify $f1.S$ to call $e1a$, with a parameter indicating read or write, rather than reading or writing $e1$ directly. Replace the edge from $f1$ to $e1$ by one from $f1$ to $e1a$, and another from $e1a$ to $e1$.

Thus, when the new node is moved to a group during N-way partitioning, the port follows it. An accessing procedure accesses the port by calling the new node's procedure, which results in a data transfer over a bus between the parts; this bus can be shared and be a fixed size, so no extra wires are needed between the parts for the transfer. This transformation is similar in concept to extended parallel I/O.

Port calling makes the granularity finer, by isolating port access behavior as distinct from accessing procedures' behaviors. Further details can be found in Vahid [1999b].

3.6 Example

Figure 4 illustrates sample call-graph modifications made during granularity selection on the earlier example. Computation exlining is used to split procedure *Mode*1 because it is large and accesses two different sets of procedures: those accessing an external LCD, and those accessing a transmitter. Such exlining separates the original procedure's statements into a procedure *Mode*1*a* that accesses the LCD and a procedure *Mode*1*b* that accesses the transmitter. Procedure inlining is used to eliminate the very small procedure *LcdClear*, which

consists of just one statement calling *LcdSend* with a particular parameter value. Procedure cloning is used to make a copy of procedure *Util* called *Utila*, for exclusive use by *LcdUpdate*, thus providing a good separation between the LCD and transmitter procedures. Port calling inserts nodes *lbusa* and *mbusa*, so that the corresponding ports can be moved freely among parts. Note: because *level* is a variable, its node can't be cloned, as discussed in Section 3.2.

4. PARTITIONING STEP 2: PRECLUSTERING

It is hoped that the first partitioning step, procedure determination, has created a coarse-grained set of procedures for N-way partitioning with as few procedures as possible, but still exposing the regions of the solution space in which all good solutions would exist. However, we can still prune away regions of the remaining solution space that we are confident would not include a good solution, but that instead represent regions that could not be pruned during procedure determination. We can do this by grouping certain procedures that we are confident should never be separated in a good partitioning. We call this *preclustering*.

Consider, for example, procedures *LcdUpdate* and *LcdSend* in Figure 4. These two procedures communicate heavily, with a frequency of 48 calls for the given interval, communicating 8 bits on each call. Those 48 calls come not from a loop that loops 48 times, but rather 48 separate calls, each with its own set of parameters. *LcdSend* contains 20 statements. Thus, inlining *LcdSend* into *LcdUpdate* would have increased code size significantly, by about $48^*20 = 960$ lines—more than the entire original program itself. Thus, inlining is not performed. Cloning would not help, nor would the other transformations. Yet we are confident that separating these two procedures would never be a good idea, because of their heavy communication, as well as their similarity in terms of relaxed timing constraints and similar functionality, consisting mostly of control statements, with no significant difference in, say, arithmetic operations (such as a multiply in one procedure but not the other). Thus we choose to cluster *LcdUpdate* and *LcdSend*. Likewise, we cluster *XmitLevel* and *XmitData*, since they both are accessed by *Mode*1*b* and since both access *mbusa*. The resulting call graph after preclustering is shown in Figure 5.

Designers will often want to manually precluster certain procedures. For automated preclustering, we use a hierarchical clustering heuristic. The procedures created after granularity selection are each converted to a graph node, and edges are created between every pair, weighed by the closeness of the nodes. The closest pair of nodes is merged into a new (hierarchical) node, and the merging repeats until no pair of nodes exceeds a minimum closeness threshold. The closeness between this new node and other nodes can be recomputed, or simply approximated as the minimum, maximum, or average of all edges being replaced. Closeness is defined using a weighted sum of several normalized closeness metrics. We must take care to define these metrics as a normalized number (between 0 and 1), because leaving such normalization to the user through the selection of weights, as many previous approaches require, is an extremely difficult if not impossible user task. The *connectivity* metric measures the size

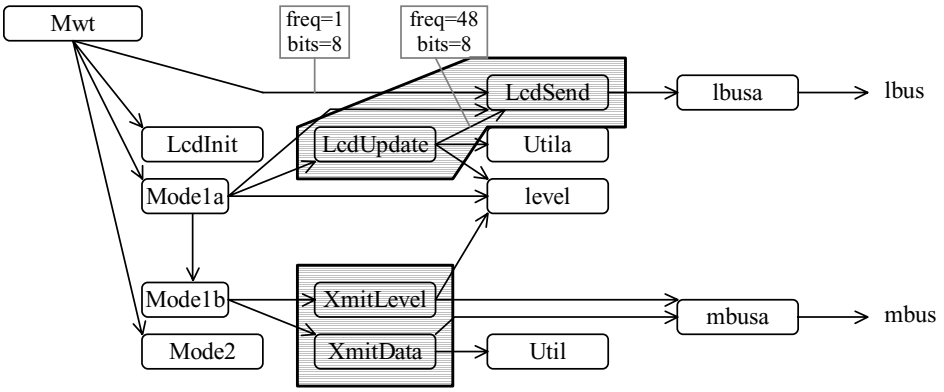

Fig. 5. Microwave transmitter example after the second partitioning step of preclustering.

of data shared among the two sets of nodes (irrespective of the frequency of access to those data), normalized by dividing by the total size of all data accessed by either set. *Communication* is the number of bits transferred between two sets of nodes, normalized by the bits transferred between either set and any other node; this metric requires profiling information. *Common accessors* can be used when profiling information does not exist; it measures the number of procedures that access nodes in both sets, divided by the number of accessors of either set. *Sequential execution* is the number of pairs of procedures, one in each set, that cannot execute concurrently, divided by the number of possible pairs. *Shared hardware* is a measure of the hardware that can be shared by two sets of nodes $B1$, $B2$, computed as $size(B1)+size(B2)-size(B1+B2)$, divided by the minimum of $size(B1)$ and $size(B2)$. Note that several of these metrics are similar in nature to those for logic-operation clustering and arithmetic-operation clustering [Lagnese and Thomas 1991].

The user influences automated preclustering by providing weights that multiply each normalized metric value before being summed into a single closeness value, and by providing a minimum closeness threshold indicating the minimum closeness value two nodes must have to be merged.

Note in Figure 1 that pre-estimation should be performed again after procedure determination, to appropriately annotate all nodes and edges in the new call graph.

5. PARTITIONING STEP 3: *N*-WAY PARTITIONING

The goal of N-way partitioning is to assign procedures among a set of N groups. Common heuristics include simulated annealing, extended versions of the Kehrnighan/Lin algorithm, Tabu search, hierarchical clustering, genetic evolution, dynamic programming, integer linear programming, and greedy heuristics, varying in time and space complexity and quality of results. This step has been widely studied and thus we omit details here. We point out that existing N-way partitioning algorithms need no modification to account for the first step, since the first step creates a new call graph. The second step requires very minor modification to N-way partitioning algorithms, to ensure that the clustered

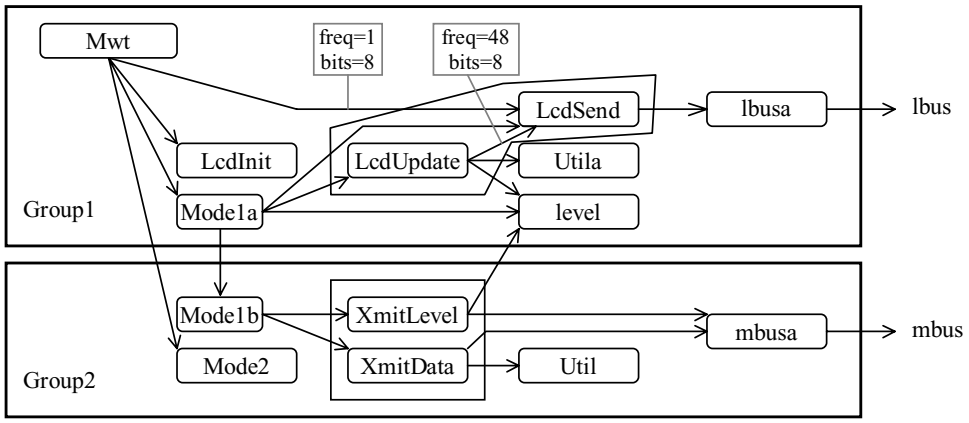

Fig. 6. Microwave transmitter example after third step of N-way partitioning.

nodes are treated as one node, but they cannot be merged into a new call graph node because even clustered nodes represent distinct procedures with different pre-estimation annotations used by the online estimators. See Figure 6.

6. EXPERIMENTS

We have conducted initial experiments to illustrate the impact that individual transformations and clustering can have on subsequent N-way partitioning. We have also conducted experiments in which we iterated among one transformation (procedure cloning) and N-way partitioning, and others in which we tried combining those two steps into one. Results show the advantages of the first two steps preceding the third, and thus demonstrate the importance of including the three steps in a partitioning approach, and also the need for further study on iterating and combining the steps. The experiments were performed using the SpecSyn system exploration tool from UC Irvine, which can read in a sequential program along with processor technology files, and performs partitioning and estimation using an extensive set of data structures and algorithms. Further details on SpecSyn can be found in Gajski et al. [1998].

6.1 Procedure Determination Experiments

We performed procedure cloning on five examples: a telephone answering machine, an Ethernet coprocessor, a fuzzy-logic controller, a set-top box, and a microwave transmitter controller. These examples ranged in size from 300 to 1000 lines of behavioral VHDL code, averaging 700 lines. The partitioning problem was to perform hardware/software partitioning among a microcontroller and a field-programmable gate array (FPGA), using simulated annealing as the N-way partitioning algorithm, with a cost function seeking to minimize execution time while also minimizing hardware size and input/output pins. We compared results of N-way partitioning applied to the call graph without cloning, to results of N-way partitioning applied to the call graph created after max-cloning. The average improvement in performance achieved was a rather good 30%, due to less communication between hardware and software resulting from

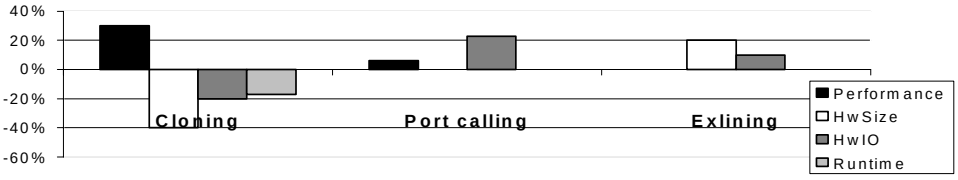

Fig. 7. Average improvements obtained by various procedure determination transformations followed by N-way partitioning (among hardware and software) using simulated annealing.

each having their own clones of critical procedures. Hardware size increased by 40% (over the original average of 8000 gates), hardware I/O by 20%, software size by 2%, and N-way partitioning run-time by 17%—all increases due to having more nodes in the call graph. Thus, cloning is clearly an excellent means for improving performance.

We performed port calling on the same five examples, without cloning, with the results as summarized in Figure 7. The average improvement in FPGA input/output lines was 23%, accompanied by an average performance improvement of 6%, with no penalty in hardware size or run-time. Thus, port calling demonstrated its effectiveness in reducing input/output requirements.

To examine the impact of procedure exlining, we needed to use other examples, since the above examples were written by us and already highly procedurized. Thus, we obtained three externally developed examples: an image processor, an MPEG decoder, and an encrypter. Their sizes were 800, 2200, and 800 lines of behavioral VHDL code, respectively. Exlining (for distinct computations) had the biggest impact on the MPEG example, increasing the 8 original procedures to 48 procedures. This finer granularity enabled N-way partitioning to obtain 30% less hardware (originally 27,000 gates) and 10 less input/output pins while still meeting timing constraints. The encryption example's performance did not change, but synthesis run-time was reduced by 75%, and hardware size by 18%. The image processor example was sped up slightly when exlining preceded N-way partitioning.

Average results for cloning, port calling, and exlining are summarized in Figure 7.

6.2 Preclustering Experiments

We experimented extensively with the preclustering step to determine its impact on run-time and partition quality. One set of experiments consisted of evaluating the run-time and resulting partition quality of simulated annealing when preceded by different amounts of clustering. A subset of the results is shown in Figure 8 for three of the examples. Clustering was done using a closeness function that was a weighted sum of connectivity, communication, and shared hardware. The X-axis shows the number of nodes after preclustering; note that we intentionally set preclustering's termination criteria to be the number of final nodes, in increments of five. The rightmost x-value of each example represents the number of nodes without preclustering. The Y-axis shows the partition cost after N-way partitioning is applied on the preclustered graph. The cost is a weighted sum of performance, hardware size, and hardware I/O,

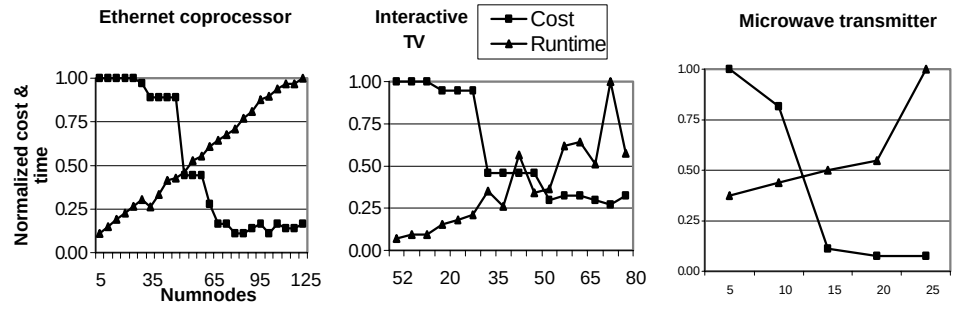

Fig. 8. Impact of different degree of clustering followed by N-way partitioning simulated annealing on three examples.

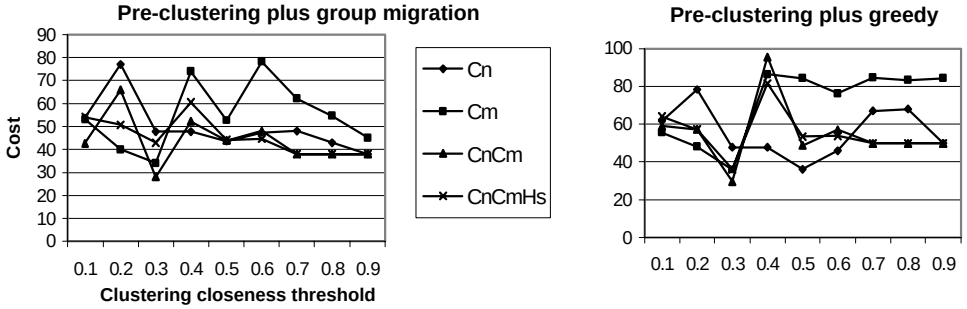

Fig. 9. Impact of different degrees of clustering followed by N-way partitioning with group migration and greedy improvement, showing average results over five examples.

with lower cost being better. The Y-axis also represents the run-time of N-way partitioning. Both cost and time are normalized to numbers between 0 and 1, by dividing by the maximum value of each, so that we can display them on the same plot. By comparing the cost points with the rightmost cost point of each example, we see that preclustering does not improve partition quality significantly, because simulated annealing is a powerful algorithm. However, we do see that preclustering done to just the right amount can reduce the algorithm's run-time by between 25 and 50%, without loss of quality. This can be important if we will be running N-way partitioning numerous times, perhaps to try mapping to different target processors. The maximum run-time for the three examples shown was 2100, 1700, and 64 seconds on a Sparc20.

If we do indeed plan to run N-way partitioning numerous times, then we might use a faster N-way partitioning algorithm, such as group migration [Gajski et al. 1998] (generalized Kehrnighan/Lin) or a greedy algorithm. Those algorithms are significantly faster: while simulated annealing required on the order of 1000 seconds per example, group migration required 100 seconds, and greedy only 10 seconds. Preclustering followed by N-way partitioning using group migration or greedy algorithms should yield better quality partitions compared to the case when no preclustering is performed before N-way partitioning using those algorithms. Figure 9 shows the average results for the previous five examples, when preceding group migration or a greedy algorithm by

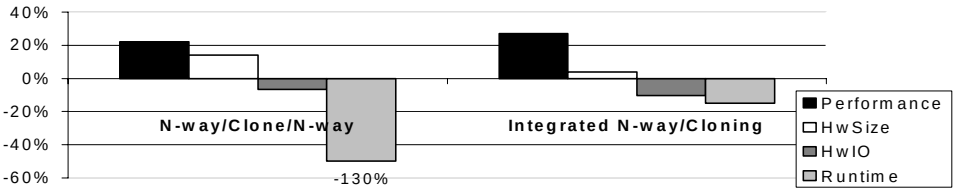

Fig. 10. Average improvements when iterating/combining some of the partitioning steps.

clustering. Here, we experimented with a termination criterion ranging from a closeness threshold of 0.1 to 0.9, plotted on the X-axis. For each possible threshold, we also experimented with 32 different combinations of closeness metrics, with the figure showing 4 of them: connectivity only (*Cn*), communication only (*Cm*), connectivity and communication (*CnCm*), and connectivity and communication and hardware sharing (*CnCmHs*). The Y-axis shows overall cost, defined as above as a weighted sum of performance, hardware size, and hardware I/O, with lower cost being better. We see that the closeness function combining connectivity and communication seems to work best for these examples. A closeness threshold of 0.9 means nodes will only be merged if they are extremely close, so little merging will be done. A threshold of 0.1 means objects with nearly any relationship will be merged. We see that a threshold of 0.9 has little impact on N-way partitioning (except in one case), and that a threshold of 0.1 makes results worse, since too much merging is done. We also see that a threshold of 0.3 seems to give the best results (this is true for each of the five individual examples, so the average is a good representation). However, we would have expected the costs to decrease from 0.9 to 0.3, whereas it increases slightly and in fact jumps up at 0.4, so further investigation is necessary to understand the causes. Nevertheless, we can conclude from the data that a good preclustering can indeed improve the quality of subsequent fast N-way partitioning heuristics.

6.3 Iterating/Combining the Three Steps

We have conducted some initial experiments that involve iterating and combining the three partitioning steps outlined in this article. One set of experiments involved performing N-way partitioning (using simulated annealing) on the original call graph of the above five examples, followed by max-cloning, and followed again by N-way partitioning. Recall that max-cloning performed on an already partitioned call graph only clones a procedure if it is accessed from within its own group and from another group. Results are shown in Figure 10, labeled N-way/Clone/N-way. Notice that we still obtain excellent performance improvement of 22%, this time with hardware size improvement rather than penalty, but at the expense of longer run-time since N-way partitioning must be run twice.

Another set of experiments consisted of iterating even further, by following the above N-way/Clone/N-way with another iteration of max-cloning and N-way partitioning. However, improvements were very minor, so we do not plot those data in a figure.

A third set of experiments involved actually combining cloning with N-way partitioning. Our simulated annealing N-way partitioning originally consisted of a step that randomly selected a node to move from one group to another. We expanded this step to also consider with some probability to clone or unclone a random node rather than moving a node. We experimented with several probabilities and found that a small probability, like 0.05, worked best. Results are shown in Figure 10, labeled Integrated N-way/Cloning. We see again excellent performance improvement of 26%, with only a minor run-time penalty of 15%.

Future experiments could include extending the iterated approach with more procedure determination transformations. Likewise, we could extend the combined approach by adding more transformation alternatives to the simulated annealing move step, with each transformation possibly having different probabilities.

In addition, clustering could be inserted into the iterations, using closeness metrics that take into account whether two nodes are in the same group after N-way partitioning. Furthermore, clustering could be combined with N-way partitioning, by adding the clustering or declustering of close nodes as yet another alternative during simulated annealing's move step. Note that this combined method is very similar to the "dynamic granularity" approach taken by Henkel and Ernst [1997], which showed excellent results.

7. CONCLUSIONS

We have described three steps that together form a good basis for partitioning of a sequential program for purposes of CAD. Each step uses different methods and has somewhat different goals. We have focused on the first two steps of procedure determination and preclustering, which represent the more novel aspects of our work. We have shown the significant impact that these first two steps can have on the resulting quality and run-time of the widely studied third step, N-way partitioning. We have also provided results from some experiments showing the benefits of iterating or combining some of these steps. Results showed that improvements of 20 to 25% in performance, hardware size, hardware I/O, or run-time were possible in different cases. We believe this work illustrates the benefits of the three steps and the potential gains to be made, and thus provides the basis and motivation for future work investigating further the three steps and their integration.

REFERENCES

ARNOLD, M., FINK, S., SARKAR, V., AND SWEENEY, P. F. 2000. A comparative study of static and profile-based heuristics for inlining. *SIGPLAN Not. 35*, 7, (July), 52–64.

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.

CALLAHAN, T. J., HAUSER, J. R., AND WAWRZYNEK, J. 2000. The Garp architecture and C compiler. *IEEE Comput.* (April).

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

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 Comput.* (Dec.), 64–75.

GAJSKI, D. 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. VLSI Syst. 6*, 1, 84–100.

GUPTA, R. AND DE MICHELI, G. 1993. Hardware/software cosynthesis for digital systems. *IEEE Des. Test Comput.* (Oct.), 29–41.

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.

HENKEL, J. 1999. A low power hardware/software partitioning approach for core-based embedded systems. In *Proceedings of the Design Automation Conference (DAC)*.

HENKEL, J., AND ERNST, R. 1997. A hardware/software partitioner using a dynamically determined granularity. In *Proceedings of the Design Automation Conference.*

HWANG, E., VAHID, F., AND HSU, Y. C. 1999. FSMD Functional Partitioning for Low Power. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference* (March), 22–28.

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 Codesign*, 42–48.

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.

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

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

STITT, G. AND VAHID, F. 2002. The energy advantages of microprocessor platforms with on-chip configurable logic. *IEEE Des. Test Comput.*, Nov.-Dec.

VAHID, F. 1995. Procedure exlining: A transformation for improved system and behavioral synthesis. In *Proceedings of the International Symposium on System Synthesis* (September), 84–89.

VAHID, F. 1999a. Procedure cloning: A transformation for improved system-level functional partitioning. *ACM Trans. Des. Autom. Electron. Syst. 4*, 1, 70–96.

VAHID, F. 1999b. Techniques for minimizing and balancing i/o during functional partitioning. *IEEE Trans. CAD 18*, 1 (Jan.), 69–75.

VAHID, F., LE, T. D. M., AND HSU, Y. C. 1998. Functional partitioning improvements over structural partitioning for packaging constraints and synthesis-tool performance. *ACM trans. Des. Autom. Electron. Syst. 3*, 2, 181–208.

WAN, M., ICHIKAWA, Y., LIDSKY, D., AND RABAEY, J. 1998. An Energy conscious methodology for early design exploration of heterogeneous DSP's. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, 111–117.

WOLF, W. H. 1997. An architectural co-synthesis algorithm for distributed, embedded computing systems. *IEEE trans. VLSI Syst. 5*, 2 (June), 218–229.

WU, S. AND MANBER, U. 1992. Fast text searching allowing errors. *Commun. ACM 35*, 10, 83–91.

Received December 1998; revised May 2001; accepted February 2002