A Three-Step Approach to the Functional Partitioning of Large Behavioral Processes

Abstract
Earlier work has demonstrated that partitioning one large behavioral process into smaller ones before synthesis can yield numerous advantages, such as reduced synthesis runtime, easier package constraint satisfaction, reduced power consumption, improved performance, and hardware/software tradeoffs. In this paper, we describe a novel three-step functional partitioning methodology for automatically dividing a large behavioral process into mutually-exclusive subprocesses, and we define the problems and our solutions for each step. The three steps are granularity selection, pre-clustering, and N-way assignment. We refer to experiments throughout that demonstrate the effectiveness of the solutions.

1 Introduction
Functional partitioning divides a system’s functional specification into multiple sub-specifications. Each sub-specification represents the functionality of a system component, such as a custom-hardware or software processor, and is eventually synthesized down to gates or compiled down to machine code. While much earlier research focuses on the partitioning of a large number of processes among processors, we have found that many applications consist of only one or a small number of very large processes.

In such cases, we have found numerous benefits to automatically partitioning, before synthesis or compilation, one large process into smaller sub-processes that execute in mutual exclusion [1]. One benefit is an order-of-magnitude reduction in logic synthesis runtimes (tens of hours down to tens of minutes). Such reductions occur because synthesis tool heuristics are usually nonlinear, so the sum of runtimes for synthesizing several small processes (where we assume the common situation of a synthesis tool synthesizing each process to its own custom processor) can be much less than the runtime for one large process. A related benefit is improved system performance sometimes achieved when the smaller processes can be synthesized into custom processors with shorter clock periods than one large processor, and when these shorter periods outweigh the overhead of inter-processor communication. In fact, synthesis tool manuals often recommend that a designer functionally partition a system manually before synthesis for these very reasons. (Of course, further performance benefits can be gained by allowing the multiple processes to execute concurrently, a natural extension of the work in this paper). Another benefit is the improved satisfaction of input/output (I/O) and size capacity constraints on a package such as an FPGA, resulting from nearly order-of-magnitude reductions in inter-package signals compared to structural partitioning, which is known to be I/O limited [2]. Such improved satisfaction leads to fewer packages and thus smaller board sizes and lower cost systems, while also having implications for easing the mapping problem of logic emulators. Recent experiments have demonstrated 50% reductions in power consumption achievable through functional partitioning [3], since only one of the small mutually-exclusive processors is active at a time, meaning the other processors have no switching activity and thus consume almost no power. Some hardware/software codesign researchers have shown the benefits of partitioning a single process to reduce hardware cost or increase software speed (e.g., [4, 5]). Other potential benefits include concurrent synthesis or design, easier debug, and possibly fewer physical design problems when dealing with several smaller modules rather than one large one.

Our earlier research has focused on functional partitioning heuristics, rapid estimation during partitioning, transformations for improving partitioning results, and experiments demonstrating the benefits of such partitioning. In this paper, we describe a novel three-step functional partitioning methodology necessary to put together the earlier pieces into a coherent automated approach. We highlight experiments throughout. We also summarize related work.

2 Problem description

2.1 Model
The input consists of a single behavioral process \( X \), such as a C program or a VHDL process. The process describes a complex repeating sequential computation, often consisting of numerous models of operation, and typically requiring many hundreds or thousands of lines of sequential program code. This process can be viewed as consisting of a set of procedures \( F = \{ f_1, f_2, ..., f_n \} \), with one representing a main procedure (in VHDL, the process body is the main procedure). A variable is treated as a simple procedure, with reads and writes being procedure calls. Execution of \( F \) consists of procedures executing sequentially, starting with the main procedure, which in turn calls other procedures; at any given time, one and only one procedure is active — in other words, the procedures are mutually exclusive.

Functional partitioning creates a partition \( P \) consisting of a set of parts \( \{ p_1, p_2, ..., p_m \} \), such that every procedure \( f_i \) is assigned to exactly one part \( p_j \), i.e., \( p_1 \cup p_2 \cup ... p_m = F \) and \( p_i \cap p_j = \emptyset \) for all \( i, j, i \neq j \). Each \( p_j \) represents the functions to be implemented on a single processor. Execution of \( F \) is the same as above. Since only one procedure is active at a time during ex-
cution, then only one processor will be active at a time, so the processors are mutually exclusive.

Each part $p_j$ is converted to a single process before synthesis; this process consists of a loop that detects a request for one of the part’s procedures, receives the necessary input parameters, calls the procedure, and sends back any output parameters. All parameter passing occurs over a single bus between the processors, called a FunctionBus, consisting of shared address/data lines, an address request control line, and a data request control line. The protocol is one of putting the destination procedure’s address on the bus and pulsing the address request, then putting each parameter on the bus and pulsing the data request, possibly sending a parameter in several chunks if its size exceeds the bus size.

Synthesis converts a process into a custom processor component $c_i$ for the applications we target, $c_i$ consists of a non-trivial datapath and a complex controller with hundreds of states. A procedure on $c_i$ may be implemented using any of various techniques, such as a control subroutine, a datapath component, or even inlined. Synthesis may implement some of a process’ procedures in parallel, as long as data dependencies are not violated – therefore, while the procedures are not necessarily mutually exclusive after partitioning, the processors still are mutually exclusive.

We will focus in this paper primarily on a process destined for synthesis, although compilation could be used to convert the process into software running on a standard processor, differing from a synthesized custom processor in its cost, performance, power, design time, etc. Hence, the approach described can also be applied to some cases of hardware-software partitioning.

Note that processors are orthogonal to packages. A package, such as an ASIC or an FPGA, is a physical structure that implements processors. Conceptually, a package may implement more than one processing element, as is often the case now with system-on-a-chip technology, or a processor may be implemented on more than one package, which is becoming quite rare. We address the problems that assume either one processor per package or multiple processors per package, the difference manifesting itself in interprocessor communication times and I/O constraints.

2.2 Tasks

There are five main tasks necessary to achieve a good functional partitioning. Model creation converts the input specification to an internal model; we use a call graph model. For example, Figure 1(a) shows an extremely simplified input (the example comes from a 720 VHDL example of a microwave-transmitter controller) with one process $Mwt$ one global variable, and numerous procedures. Figure 1(b) shows the call graph, where each node represents a procedure, and each edge a procedure call. Note that by using a call graph, we can easily handle specifications having many multiply-called and deeply nested procedures, which is a common situation often overlooked by other approaches. Allocation is the task of selecting and instantiating processors of varying types; we assume this is done beforehand, but future work may attempt to automate this. Partitioning is the task of dividing the input process among the allocated processors – this task is the focus of this paper. Transformation modifies the input process into one with a different organization but same overall functionality, leading to a better partition, and is also mentioned in this paper. Estimation provides data used to create values for design metrics. We use two types of estimation. Pre-estimation computes estimation informa-
tion for a given graph node (procedure) or edge, and is usually done once, before partitioning heuristics are applied. It can thus take seconds or minutes. It is usually achieved by performing a rough synthesis, using statistical models to bypass certain time-consuming synthesis tasks. Examples of pre-estimated data include the number of bytes, gate, or clock cycles for a functional object to execute on each possible component. For example, Figure 1(b) shows values for calling frequency and bits per transfer for three edges. Online-estimation determines actual metric values for a given implementation by combining pre-estimated values, and is usually done hundreds or thousands of times during application of a partitioning heuristic. It thus should take only milliseconds. It usually involves sophisticated data structures that maintain a design model, and can be incrementally updated as objects are moved among components. In both estimation types, accuracy must be traded off with speed. Estimation is an extremely important task and is closely related to partitioning, but we have described our techniques in other papers [6, 7].

3 Partitioning methodology

We apply a three-step partitioning methodology to obtain an implementation. They occur in sequence as shown in Figure 2, though iterating over the sequence is certainly desirable. We now describe each step.

3.1 Step 1: granularity selection

The goal of this partitioning step is to extract procedures from the specification, which are to be assigned to processors during subsequent N-way assignment. The essential feature of this step is that it determines the granularity of subsequent N-way assignment. Granularity is a measure of the complexity of each procedure. Fine-granularity means many procedures of low complexity, while coarse granularity means few procedures of high complexity.

Improper granularity selection can have negative effects on subsequent partitioning results. First, fine-granularity means little useful pre-estimation can be done, which in turn means online estimation may be less accurate. Alternatively, online estimation may be made more complex to achieve accuracy, requiring more time and thus prohibiting use of powerful N-way assignment heuristics that need many estimates.

Second, fine-granularity means there will be numerous procedures, preventing the use of powerful partitioning heuristics that have high, such as quadratic, runtime complexities.

Third, coarse-granularity means that many behaviors are grouped together into an inseparable unit, so any possible solution that separates those behaviors is excluded. If coarse-grained procedures are not created carefully, they can prevent good solutions.

From the above effects, we can see the importance of a distinct partitioning step that chooses the procedures very carefully. In our approach to this step, we treat each statement as an atomic unit. Granularity selection is thus the problem of partitioning statements into procedures, such that: (1) procedures are as coarse-grained as possible, to enable maximum pre-estimation and application of powerful N-way heuristics, and (2) statements are grouped into a procedure only if their separation would yield inferior solutions.

The most straightforward heuristic for this partitioning step is to choose a particular specification construct to represent a procedure, such as each statement, each block, or user-defined procedure. In our approach, we start by selecting each user-defined procedure (along with each global and non-scalar variable) as a procedure for partitioning. User-defined procedures tend to be excellent coarse-grained groupings of statements, but there is often a need for some improvement, which can be obtained through transformations.

Procedure inlining is the well-known transformation of replacing a procedure call by the procedure’s contents, thus making granularity coarser (the inlined procedure disappears, while all calling procedures are made more complex).

Procedure cloning makes a copy of a procedure for exclusive use by a particular caller. This transformation is a compromise between inlining and not inlining. A multiply-called procedure when inlined might lead to excessive size growth, but if not inlined might become a communication bottleneck. Cloning eliminates the bottleneck while avoiding excessive size growth. Prior experiments demonstrate performance and I/O improvements obtainable through cloning [8].

Procedure exclining, as its name implies, is the inverse of inlining. It replaces a subsequence of a procedure’s statements by a call to a new procedure containing only that subsequence. This transformation makes the granularity finer, as it essentially replaces one procedure by two simpler ones. We have isolated two forms of exclining. Redundancy exclining seeks to replace two
or more near-identical sequences of statements by one procedure. We use an interactive approach based on approximate-string matching, in which each statement is encoded into a set of characters and appended to a string representing the entire program, and then approximately-redundant strings are searched for. Distinct computation exiling 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 assignment solution. Once again, we make statements our atomic unit; an approach at the arithmetic-operation level is described in [9]. We transform the statements into a tree, and then perform one of three heuristics (varying in their complexity) that insert procedure nodes into the tree with the goal of minimizing a cost function including procedure size and communication. Further exiling details can be found in [10].

Figure 1(c) illustrates sample call-graph modifications made during granularity selection on the earlier example. The procedure Model is large and accesses procedures that access both an external LCD and a transmitter. Exiling introduces a new procedure Mod1a, which makes the granularity finer and expanding the partitioning solution space to hopefully include better possible solutions. In particular, it separates the original procedure's statements into those that access the LCD and those that access the transmitter. On the other hand, LedClear is very small, consisting only of one call to LedSend with a particular parameter value. Thus, LedClear is inlined, making the granularity coarser and thus eliminating hopefully inferior partitioning solutions.

Experiments illustrating the partitioning improvements obtained by exiling can be found in [10].

3.2 Step 2: pre-clustering

Given a set of procedures, the goal of pre-clustering is to reduce the number of procedures for subsequent N-way assignment, by merging procedures whose separation among parts would never represent a good solution.

This step is distinguished from granularity selection by the fact that the procedures being clustered here may not be such that they could have been exlined into a single new procedure; i.e., the calls to these procedures do not appear as adjacent statements. Typically, calls to such procedures are scattered throughout the specification. This step is distinguished from N-way assignment by the fact that each cluster does not represent a processor (many clusters may be assigned to each processor), so this step cannot be guided by direct design metric estimates as can N-way assignment.

We perform pre-clustering by using a hierarchical clustering. 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. 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 approaches require, is an extremely difficult if not impossible user task. We describe our closeness metrics in [11]. Some of those metrics are similar to those for logic-operation clustering [12] and arithmetic-operation clustering [13, 9].

Figure 1(d) illustrates results of pre-clustering on the earlier example. Two procedures LedUpdate and LedSend communicate very heavily: 48 times per call to LedUpdate, transmitting 8 bits each time. These two procedures should probably never be separated. Since LedSend actually appears 48 times inside LedUpdate, inlining during granularity selection was not a reasonable option. We merge these two procedures during pre-clustering; they are subsequently treated as one object.

We demonstrated in [11] that pre-clustering can reduce the runtime of N-way assignment using simulated annealing or Kernighan/Lin by between 25% and 30%. For N-way greedy heuristics, pre-clustering can actually improve the final results significantly. To illustrate the benefits of pre-clustering, we experimented with two-way partitioning of an Ethernet coprocessor example, consisting of 125 procedures to be partitioned. The tradeoffs between the amount of pre-clustering versus the subsequent runtime of the Kernighan/Lin heuristic and the cost of the final result are shown in Figure 3. Cost is unit-less number representing a weighted sum of normalized size, I/O and performance metrics; smaller means better. We see that when pre-clustering merges until there are only 85 objects, runtime is decreased by 30%, and a better-cost partition is found by the Kernighan/Lin heuristic. Merging to less than 55 objects yielded much higher costs, due to distant objects being forcibly merged.

![Fig. 3: Pre-assignment clustering effects on iterative-improvement runtime and resulting partition cost.](image-url)
3.3 Step 3: N-way assignment

The goal of N-way assignment is to distribute the procedures among the given set of processors. Granularity selection and pre-clustering should have already created a good set of procedures.

Constructive heuristics are used to create an initial partition, and include random distribution, as well as clustering using the same closeness metrics defined above. One additional metric is used: Balanced size is the size of an implementation of both sets of nodes, divided by the size of all nodes. This metric favors merging small sets over large ones, thus preventing one set from getting too large and defeating the purpose of functional partitioning.

Iterative improvement heuristics improve a given partition by moving procedures from one part to another in order to reduce the value of a cost function. We use three heuristics: Greedy improvement is a linear-time heuristic that moves nodes that reduce cost until no further such node is found. Simulated annealing uses randomized hill climbing to avoid local minima at the expense of potentially long runtimes. An extended version of Kernighan/Lin uses a more restricted method of hill climbing to avoid some local minima, along with a tightly-coupled data structure, yielding only \( O(n \log(n)) \) runtime complexity. We demonstrated in [6] that this last heuristic obtains results nearly as good as simulated annealing, but in just a few seconds rather than many minutes.

We apply two transformations during N-way assignment to improve partitioning results. Cloning, a transformation defined earlier, can be performed selectively along with this partitioning step, rather than trying to clone only during granularity selection [8]. A second transformation, port-calling, can be used to better balance I/O among the parts and to reduce total I/O by eliminating the case where two parts have I/O accessing one external port. This transformation involves replacing all accesses to each external port by a port-call procedure, and then moving this procedure among parts just like any other procedure. The port-call procedure acts as a broker, accessing the port, and sending/receiving data over the FunctionBus to/from the original accessing procedure.

Figure 1(c) shows the earlier example’s call graph after being partitioned into two parts. While not entirely obvious from the figure because not all annotations are shown, the partition minimizes communication, isolates external I/O access to one or the other part, and creates roughly balanced parts, each of which is roughly half the size of the original input.

We have performed numerous experiments illustrating the relative tradeoffs among various N-way heuristics [6, 11]. We have also performed experiments showing I/O and performance improvements obtained through cloning [8] as well as through port-calling [14].

4 Related work and limitations

Several researchers have developed functional partitioning techniques for partitioning among hardware packages. Most efforts partitioned arithmetic-level operations. Aparty [9] partitioned operations among datapath modules or packages using multi-stage clustering. Vulcan [15] partitioned operations among packages using iterative improvement heuristics. Chop [16] partitioned operations among packages, focusing on providing a suite of feasible solutions for each package that would satisfy overall constraints. Multipar [17] and Gebotys’ techniques [18] partitioned operations among packages simultaneous with scheduling and allocation, using integer-linear programming. Other efforts partitioned procedural-level operations. SpecPart [7] partitioned procedures among packages using a suite of heuristics, including clustering and iterative improvement. Peng and Kucinski [19] similarly partitioned among packages. None of these efforts applied a three-step methodology as described in this work, although the idea behind multi-stage clustering in Aparty is along similar lines, while being limited to just clustering for the partitioning heuristic and focusing on finer-grained operations.

Recently, much research effort has focused on developing functional partitioning techniques for hardware-software partitioning (for example, see [20]). Most of these techniques partition and possibly schedule multiple processes among a custom-hardware and software processor architecture [21, 4, 5, 7, 22], while some focus on simultaneous processor allocation and partitioning [23]. Since these techniques focus on multiple processes, they do not apply directly to the single-process problem we address. However, Henkel and Ernst have investigated granularity during hardware-software partitioning and developed a technique to dynamically modify granularity during partitioning [24].

Our work can be viewed as filling a large gap between these two research domains. Most functional partitioning work initially dealt with extremely fine-granularity (arithmetic operations) when they were developed as extensions of behavioral synthesis and package sizes were small, but today most work focuses on very coarse granularity (processes) and deals primarily with hardware-software partitioning. We focus on medium to coarse granularity (procedures), with an emphasis on solving important synthesis problems (runtime, low power, package constraints), though still being applicable to some single-process or few-process hardware-software partitioning problems.

One limitation of the approach for partitioning before synthesis is that total hardware increase may be large for examples that have very simple controllers but large datapaths. Another limitation is that we don’t address the problem of partitioning large numbers of
small processes that problem is very much a scheduling problem. Third, our model dictates that procedures on different processors do not execute in parallel. Allowing parallel execution could improve performance, but at the expense of introducing the need for bus arbitration and other complexities.

5 Conclusions

In this paper, we described a novel three-step partitioning methodology for partitioning a large behavioral process among custom hardware or software processors. While earlier work has described detailed solutions to various functional partitioning problems, this work combines those solutions into a single coherent approach.

This work described focuses on partitioning a process destined for behavioral synthesis. However, there is presently much register-transfer level synthesis occurring in practice. We are therefore investigating the application of this three-step methodology down to the problem of partitioning finite-state machines with datapaths, or FSMD's, using a similar execution and communication model.

Furthermore, there are many parameters, described in this paper, that can be used to influence the results of each partitioning steps. Finding the right parameter values can be hard, and modifying them when iterating the three steps can be very challenging. Thus, an automatable "meta-algorithm" for setting and modifying those parameters during iteration among the steps may be an interesting research direction.

References


