3

EXTENDED PARTITIONING

In this chapter, we focus on the hardware-software partitioning problem for embedded signal processing applications. As discussed in Chapter 2, the task-level description of an application is specified as an SDF graph. The SDF graph is translated into a DAG representing precedences and this DAG is the input to the partitioning tools.

The *binary partitioning problem* is to map each node of the DAG to hardware or software, and to determine the schedule for each node. The hardware-software partitioning problem is not just limited to making a binary choice between a hardware or software mapping. As discussed in Chapter 1, within a given mapping, a node can be implemented using various algorithms and synthesis mechanisms. These implementations typically differ in area and delay characteristics; we call them “implementation bins”. The *extended partitioning problem* is the joint problem of mapping nodes in a DAG to hardware or software, and within each mapping, selecting an appropriate implementation bin (Figure 3.1).

Partitioning is, in general, a hard problem. The design parameters can often be used to formulate it as an integer optimization problem. Exact solutions to such formulations (typically using integer linear programming (ILP)) are intractable for even moderately small problems. We propose and evaluate heuristic solutions. The heuristics will be shown to be comparable to the ILP solution in quality, with a much reduced solution time.

The chapter is organized as follows. In Section 3.1, the *binary partitioning* problem is defined. This is followed by a motivation and a definition for the *extended partitioning* problem. In Section 3.2, we discuss the related work in the area of hardware-software partitioning. In Section 3.3, we present the GCLP algorithm to solve the binary partitioning problem. Its performance is analyzed in Section 3.4. In Section 3.5, we present the MIBS heuristic to solve the extended partitioning problem. The MIBS algorithm essentially solves two problems for each node in the precedence graph: hardware/software mapping and scheduling, followed by implementation-bin selection for this mapping. The first problem, that of mapping and scheduling, is solved by the GCLP algorithm. For a given mapping, an appropriate implementation bin is selected using a bin selection procedure. The bin selection procedure is described in Section 3.6. The details of the MIBS algorithm are described in Section 3.7, and its performance is analyzed in Section 3.8.

3.1 Problem Definition

In Section 3.1.1, we state the major assumptions underlying the partitioning problem. The binary partitioning problem is defined in Section 3.1.2. The extended partitioning problem is motivated and defined in Section 3.1.3.

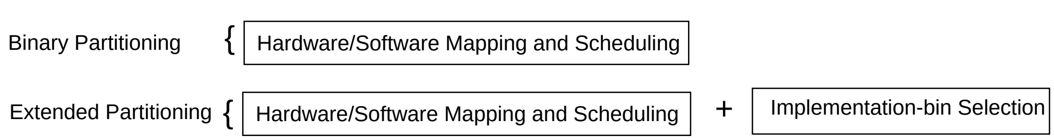

Figure 3.1. The Extended Partitioning Problem

3.1.1 Assumptions

The partitioning techniques discussed in this thesis are based on the following assumptions:

1. The precedences between the tasks are specified as a DAG ($G = (N, A)$). The throughput constraint on the SDF graph translates to a deadline constraint D, i.e., the execution time of the DAG should not exceed D clock cycles.
2. The target architecture consists of a single programmable processor (which executes the software component) and a custom datapath (the hardware component). The software and hardware components have capacity constraints — the software (program and data) size should not exceed AS (memory capacity) and the hardware size should not exceed AH. The communication costs of the hardware-software interface are represented by three parameters: ah_{comm}, as_{comm}, and t_{comm}. Here, ah_{comm} (as_{comm}) is the hardware (software) area required to communicate one sample of data across the hardware-software interface and t_{comm} is the number of cycles required to transfer the data. The parameter ah_{comm} represents the area of the interface glue logic and as_{comm} represents the size of the code that sends or receives the data. In our implementation we assume a self-timed blocking memory-mapped interface. We neglect the communication costs of software-to-software and hardware-to-hardware interfaces.
3. The area and time estimates for the hardware and software implementation bins of every node are assumed to be known. The specific techniques used to compute these estimates are described in Section 3.4.1.
4. In this work we assume that there is no resource sharing between nodes

mapped to hardware. The issue is discussed further in Chapter 4.

3.1.2 Binary Partitioning

The binary partitioning problem (*P1*): Given a DAG, area and time estimates for software and hardware mappings of all nodes, and communication costs, subject to resource capacity constraints and a deadline D, determine for each node i, the hardware or software mapping (M_i) and the start time for the execution of the node (schedule t_i), such that the total area occupied by the nodes mapped to hardware is minimum.

P1 is combinatorial in the number of nodes ($O(2^{|N|})$ by enumeration). *P1* can be formulated exactly as an integer linear program (ILP) as shown in Appendix A1. It is shown to be NP-hard in Appendix A2.

3.1.3 Extended Partitioning

As discussed in Chapter 1, in addition to selecting a hardware or software mapping for the nodes, there is yet another dimension of design flexibility — for a particular mapping, a node can have different implementation alternatives. These implementation alternatives correspond to different algorithms, transformations, and synthesis mechanisms that can be used to implement a node. Figure 3.2 shows the Pareto-optimal points in the area-time trade-off curves for the hardware implementation of typical nodes. The nodes shown include a pulse shaper, a timing recovery block, and an equalizer. The design points on the curves represent different implementations for the node resulting from different synthesis techniques. The sample period is shown on the X axis, and the corresponding hardware area required to implement the node is shown on the Y axis. The left-most point on the X axis for each curve corresponds to the critical path of that node; it represents the fastest possible implementation of the node. As the sample period increases, the same node can be implemented in a smaller hardware area. The right-most point on the X axis for each curve corresponds to the smallest possible area; it represents the point where only one resource of each type is used and the design cannot get any smaller. Thus, the curve represents the design space for the node for a particular algorithm in a hardware mapping. We have generated each of these curves by running a task-level Silage [Hilfinger85] description of the node through Hyper [Rabaey91], a high-level synthesis system. The various design points for a given node were obtained by computing the hardware area corresponding to different sample periods.

Similarly, different software synthesis strategies can be used to implement a given node in software, giving rise to similar area-time trade-off curves for software implementations. For instance, inlined code is faster than code using subroutine calls, but has a larger code size. Thus, there is a trade-off between code size

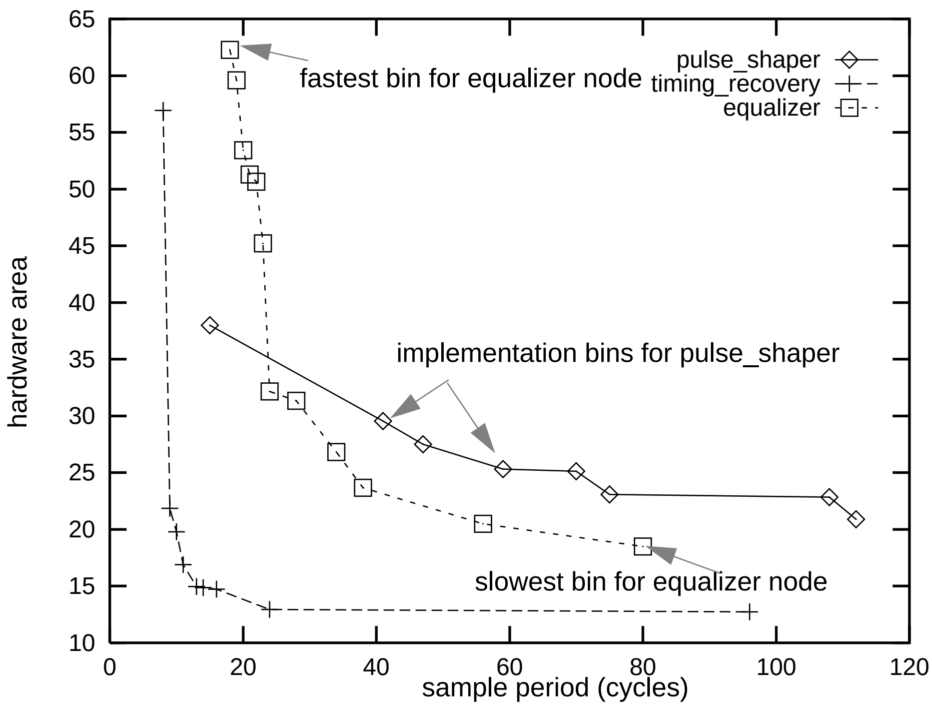

Figure 3.2. Typical hardware area-time trade-off curves, called hardware implementation curves. The design points are called implementation bins.

(program and/or data memory) and execution time. Software for each node can be synthesized using different optimization goals such as throughput, program memory, or data memory [Ritz93][Murthy94].

In Figure 3.2, a curve corresponds to various implementations of a node for a particular algorithm. Each node can also be implemented using different algorithms and transformations. Figure 3.3 generalizes this. Thus, several implementation alternatives (or implementation bins) exist for a node, within each mapping.

We assume that associated with every node i is a hardware implementation curve CH_i, and a software implementation curve CS_i. The implementation curve plots all the possible design alternatives for the node. $CH_i = \{(ah_i^j, th_i^j), j \in NH_i\}$, where ah_i^j and th_i^j represent the area and execution time when node i is implemented in hardware bin j, and NH_i is the set of all the hardware implementation bins (Figure 3.4). $CS_i = \{(as_i^j, ts_i^j), j \in NS_i\}$, where as_i^j and ts_i^j represent the program size and execution time when node i is implemented in software bin j, and NS_i is the set of all the software implementation bins. The fastest implementation bin is called L bin, and the slowest implementation bin is called H bin.

The extended partitioning problem (*P2*): Given a DAG, hardware and software implementation curves for all the nodes, communication costs, resource

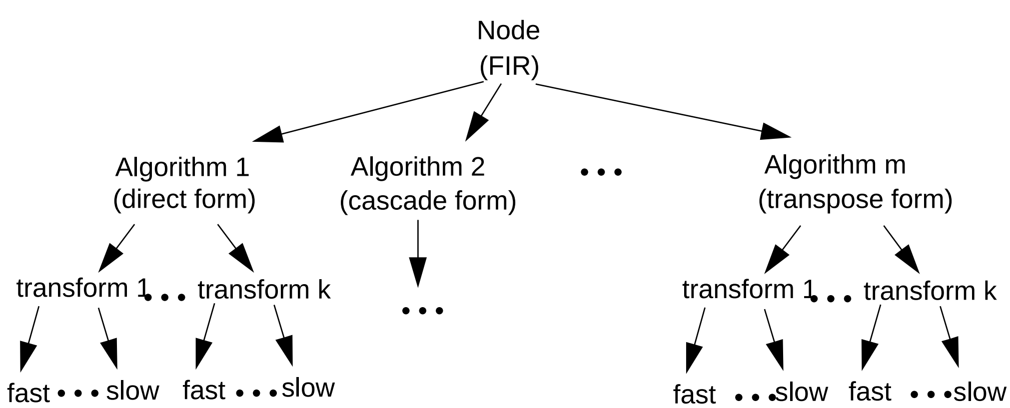

Figure 3.3. Implementation alternatives for a node within a mapping

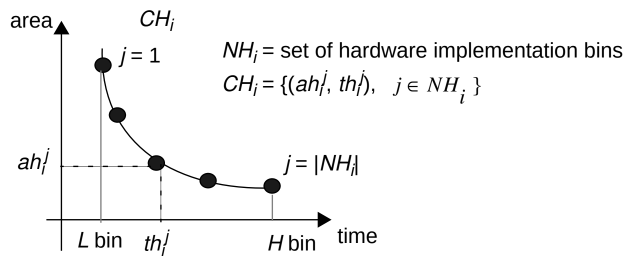

Figure 3.4. CH_i: Hardware implementation curve for node i. Each bin corresponds to a possible design option for the node.

capacity constraints, and a required deadline D, find a hardware or software mapping (M_i), the implementation bin (B_i*), and the schedule (t_i) for each node i, such that the total area occupied by the nodes mapped to hardware is minimum.

It is obvious that *P2* is a much harder problem than *P1*. It has $(2B)^{|N|}$ alternatives, given B implementation bins per mapping. *P2* can be formulated exactly as an integer linear program, similar to *P1*. An ILP formulation for *P2* is given in Appendix A3.

The motivation for solving the extended partitioning problem is two-fold. First, the flexibility of selecting an appropriate implementation bin for a node, instead of assuming a fixed implementation, is likely to reduce the overall hardware area. In this chapter, we investigate, with examples, the pay-off in using extended partitioning over just mapping (binary partitioning). Secondly, from the practical perspective of hardware (or software) synthesis, solution to *P2* provides us with the best sample period or algorithm to use for the synthesis of a given node.

3.2 Related Work

The binary partitioning problem has received some attention recently. In Section 3.2.1, we will discuss some of the other approaches used to solve the binary partitioning problem.

As of this writing, we are not aware of any work that formulates or solves the extended partitioning problem in system-level design. In Section 3.2.2, we will briefly summarize the related work in physical CAD and highlevel synthesis.

3.2.1 Binary Partitioning: Related Work

The hardware/software mapping and scheduling problem can be formulated as an integer linear program (ILP) and solved exactly. This formulation becomes intractable even for moderately-sized applications (Appendix A1). Some heuristics have been reported to solve the variants of this problem[2].

Gupta *et al.* [Gupta92] discuss a scheme where all nodes (except the data-dependent tasks) are initially mapped to hardware. Nodes are at an instruction level of granularity. Nodes are progressively moved from hardware to software subject to timing constraints in a manner described next. A hardware-mapped node is selected (this node is an immediate successor of the node previously moved to software). The node is moved to software if the resultant solution is feasible (meets specified throughput) and the cost of the new partition is smaller than the earlier cost. The cost is a function of the hardware and software sizes. The algorithm is greedy and is not designed to find a global minimum.

The scheme proposed by Henkel *et al.* [Henkel94] also assumes an instruction level of granularity. All the nodes are mapped to software at the start. Nodes are then moved to hardware (using simulated annealing) until timing constraints are met. Due to the inherent nature of simulated annealing, this scheme requires long run times and the quality of the solution depends on the cooling schedule.

---

2. Unless stated differently, the target architecture assumed in these approaches consists of a programmable processor and custom hardware modules that communicate via shared memory.

Baros *et al.* [Baros92] present a two-stage clustering approach to the mapping problem. Clusters are characterized by attribute values (degree of parallelism, amount of data dependency, degree of repetitive computations, etc.) and are assigned hardware and software mappings based on their attribute values. Clusters with a large number of repetitive computations are assigned to software. This approach ignores precedences and scheduling information; it solves only the mapping problem. Scheduling the clusters *after* mapping them to hardware or software leaves little flexibility in trying to achieve a desired throughput. The mapping process does not try to minimize hardware area or maximize software utilization.

D'Ambrosio *et al.* [Ambrosio94] describe a branch and bound-based approach for partitioning applications where each node has a deadline constraint (instead of an overall throughput deadline). Each node has three attributes: the deadline, the number of software instructions needed to execute it, and the type of hardware units it can be implemented on. The target architecture consists of a single software processor and a set of different hardware modules. The input specification is transformed into a set of constraints. The set of constraints is solved by an optimizing tool called GOPS, which uses a branch and bound approach, to determine the mapping. The approach suffers from limitations similar to those in a ILP formulation, that is, solving even moderate-sized problems can become computationally infeasible.

In the approach proposed by Eles *et al.* [Eles94], each node and arc in the graph is annotated with a weight. The weight of a node is a convex combination of four terms: computation load (number of operations executed), uniformity (the ratio of the number of operations in the node to the number of distinct types of operations in the node), parallelism (the ratio of the number of operations in the node to the length of the critical path of the node), and software-suitability (the ratio of the number of software-suitable operations in the node to the number of operations in the node. The software-suitable instructions include floating point operations, file accesses, etc.). The coefficients of these factors are decided by the designer. They claim that a node with a high weight is better suited to hardware. The weights on arcs correspond to the communication cost. The hardware-software partitioning problem is then formulated as the problem of partitioning the graph into two sets such that the sum of the communication costs across the partition is minimized, subject to the constraints that no node with a weight less (greater) than w_l (w_h) is assigned to hardware (software). The parameters w_l and w_h are set by the designer. The problem is solved by simulated annealing. Their scheme ignores throughput constraints; it does not solve the scheduling problem.

Jantsch *et al.* [Jantsch94] present an approach for improving the overall performance of an application by moving parts to hardware, subject to a constraint on the size (number of gates) of the allowed hardware. Initially, all the nodes are mapped to software and are assumed to execute according to a sequential schedule. They restrict themselves to the problem of finding the set of nodes that can be moved to hardware so as to get the maximum speedup, subject to the allowed hardware capacity. The problem translates to the knapsack packing problem (given k items, each with a utility and size, select the set of items that can be packed into a knapsack of a given capacity such that the total utility is maximized) and is solved using dynamic programming. Since they assume a schedule to be available, their approach does not consider overlapped hardware and software execution and hence could lead to poor resource utilization.

Thomas *et al.* [Thomas93] propose a manual partitioning approach for task-level specifications. They discuss the properties of tasks that render them suitable to either hardware or software mapping, such as types of arithmetic operations used, data parallelism, and existence of control structures. In their approach, the designer has to qualitatively evaluate these properties and make a mapping decision. As will be shown later, our approach not only quantifies such properties, but also incorporates mapping decisions based on algorithmic properties into an automated partitioning approach.

We will now briefly discuss some work in related areas such as software partitioning, hardware partitioning, and highlevel synthesis to evaluate the possibility of extending it to the binary partitioning problem.

The heterogeneous multiprocessor scheduling problem is to partition an application into multiple processors. The processors could be heterogeneous, i.e., the execution times on the different processors vary. Approaches used in the literature [Hamada92][Sih93] to solve this problem ignore the area dimension while selecting the mapping; they cannot be directly applied to the hardware/software mapping and scheduling problem.

Considerable attention has been directed towards the hardware partitioning problem in the highlevel hardware synthesis community. The goal in most cases is to meet the chip capacity constraints; timing constraints are not considered. Most of the proposed schemes (for example, [McFarland90b], [Lagnese91], and [Vahid92]) use a clustering-based approach first presented by Camposano *et al.* [Camposano87].

The approaches used to solve the throughput-constrained scheduling problem in highlevel hardware synthesis, (such as force directed scheduling by Paulin *et al.* [Paulin89]), do not extend to the hardware/software mapping and scheduling problem directly. Such approaches do not consider the area and time heterogeneity that is offered by the possibility of hardware or software mappings. Also, they solve only the scheduling problem — mapping (to a particular type of hardware) is assumed to be known. Decoupling the mapping and scheduling problems leaves little flexibility in trying to achieve a desired throughput. Further, most of these approaches use the notion of "slack" or "mobility", which is the difference between the ASAP and ALAP schedules, for scheduling nodes. In the hardware-software partitioning case, such ASAP and ALAP times cannot be determined exactly since each node has two possible values of execution time (corresponding to hardware and software mappings).

In Section 3.3, we will present an algorithm for automated hardware-software partitioning that overcomes most of the drawbacks of the approaches discussed here. It operates on task-level specifications and attempts to find a partition that meets the throughput constraints, while approximately minimizing the hardware area and maximizing the software utilization. It is an efficient ($O|N|^2$) heuristic that uses a combination of global and local measures while making the mapping and scheduling decisions. It also takes into account the inherent suitability of a node to either a hardware or a software realization based on intrinsic algorithmic properties. For the examples tested, the solutions generated by the algorithm are found to be reasonably close to the corresponding optimal solutions.

3.2.2 Extended Partitioning: Related Work

The authors are not aware of any published work that *formulates or solves* the extended hardware-software partitioning problem in system-level design. The problem of selecting an appropriate bin from the area-time trade-off curve is reminiscent of the technology mapping problem in physical CAD [Brayton90], and the module selection (also called resource-type selection) problem in highlevel synthesis [DeMicheli94], both of which are known to be NP-complete problems.

The technology mapping problem is to bind nodes in a Boolean network, representing a combinational logic circuit, to *gates* in the library such that the area of the circuit is minimized while meeting timing constraints. Gates with different area and delay values are available in the library. Several approaches ([Chaudhary92], among others) have been presented to solve this problem.

The module selection problem is the search for the best resource type for each *operation*. For instance, a multiply operation can be realized by different implementations, e.g., fully parallel, serially parallel, or fully serial. These resource types differ in area and execution times. A number of heuristics ([Ishikawa91], among others) have been proposed to solve this problem.

3.3 The Binary Partitioning Problem: GCLP Algorithm

In this section, we present the Global Criticality/Local Phase (GCLP) algorithm to solve the hardware/software mapping and scheduling problem (*P1*). The notation used is summarized in Table 1.

3.3.1 Algorithm Foundation

The underlying scheduling framework in the GCLP algorithm is based on *list scheduling* [Hu61]. The general approach in list scheduling is to serially traverse a node list (usually from the source node to the sink node in the DAG[3]) and for each node to select a mapping that minimizes an objective function. In the context of *P1*, two possible objective functions could be used: (1) minimize the *finish time* of the node (i.e., sum of the start time and the execution time), or (2) minimize the *area* of the node (i.e., the hardware area or software size). Neither of these objectives by itself is geared toward solving *P1*, since *P1* aims to minimize

3. Note that we are not restricted to a single source (sink) node. If there are multiple source (sink) nodes in the DAG, they can all be assumed to originate from (terminate into) a dummy source (sink) node.

area and meet timing constraints at the same time. For example, an objective function that minimizes finish time drives the solution towards feasibility from the viewpoint of deadline constraints. This solution is likely to be suboptimal (increased area). On the other hand, if a node is always mapped such that area is minimized, the final solution is quite likely infeasible. Thus a fixed objective function is incapable of solving *P1*, a constrained optimization problem. There is also a

Notation	Interpretation
	Graph Parameters
G	DAG $G = (N, A)$, where N is set of nodes representing computations, and A is set of arcs representing precedences
D	Deadline (or makespan) constraint. The execution time for the graph should not exceed D cycles.
ah_i	Hardware area estimate for node i
as_i	Software size estimate for node i
th_i	Hardware execution time estimate for node i
ts_i	Software execution time estimate for node i
$size_i$	Size of node i (number of atomic operations)
	Architectural Constraints
AH	Hardware capacity constraint (total area of nodes mapped to hardware cannot exceed AH)
AS	Software capacity constraint (total area of nodes mapped to software cannot exceed AS)
ah_{comm}	Hardware required for the transfer of one data sample between hardware and software
as_{comm}	Software required for the transfer of one data sample between hardware and software
t_{comm}	Number of cycles required to transfer one sample of data between hardware and software
	Algorithm Outputs
M_i	Mapping for node i (0 for software, 1 for hardware)
t_i	Start time for the execution of node i (schedule)

Table 1. Summary of notation

limitation with list scheduling; mapping based on serial traversal tends to be greedy, and therefore globally suboptimal.

The GCLP algorithm tries to overcome these drawbacks. It *adaptively* selects an appropriate mapping objective at each step[4] to determine the mapping and the schedule. As shown in Figure 3.5, the mapping objective for a particular node is selected in accordance with:

1. *Global Criticality* (*GC*): *GC* is a global look-ahead measure that estimates the time criticality at each step of the algorithm. *GC* is compared to a threshold to determine if time is critical. If time is critical, an objective function that minimizes finish time is selected, otherwise one that minimizes area is selected. *GC* may change at every step of the algorithm. The adaptive selection of the mapping objective overcomes the problem associated with a hardwired objective function. The "global" time criticality measure also helps overcome the limitation of serial traversal.

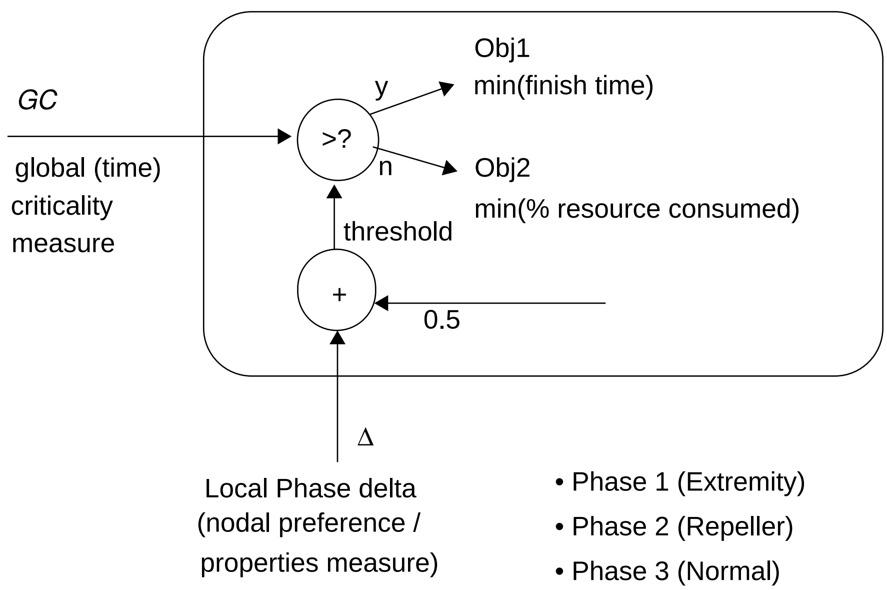

Figure 3.5. Selection of the mapping objective at each step of GCLP.

4. A *step* corresponds to the mapping of a particular node in the DAG.

2. *Local Phase* (LP): LP is a classification of nodes based on their heterogeneity and intrinsic properties. Each node is classified as an extremity (local phase 1), repeller (local phase 2), or normal (local phase 3) node. A measure called *local phase delta* quantifies the local mapping preferences of the node under consideration and accordingly modifies the threshold used in *GC* comparison.

The flow of the GCLP algorithm is shown in Figure 3.6. N represents the set of nodes in the graph. N_U(N_M) is the set of unmapped(mapped) nodes at the current step. N_U is initialized to N. The algorithm maps one node per step. At the beginning of each step, the global time criticality measure *GC* is computed. *GC* is a global measure of time criticality at each step of the algorithm, based on the currently mapped and unmapped nodes and the deadline requirements. The details of this computation will be given in Section 3.3.2. Unmapped nodes whose predecessors have already been mapped and scheduled are called *ready* nodes. A node is

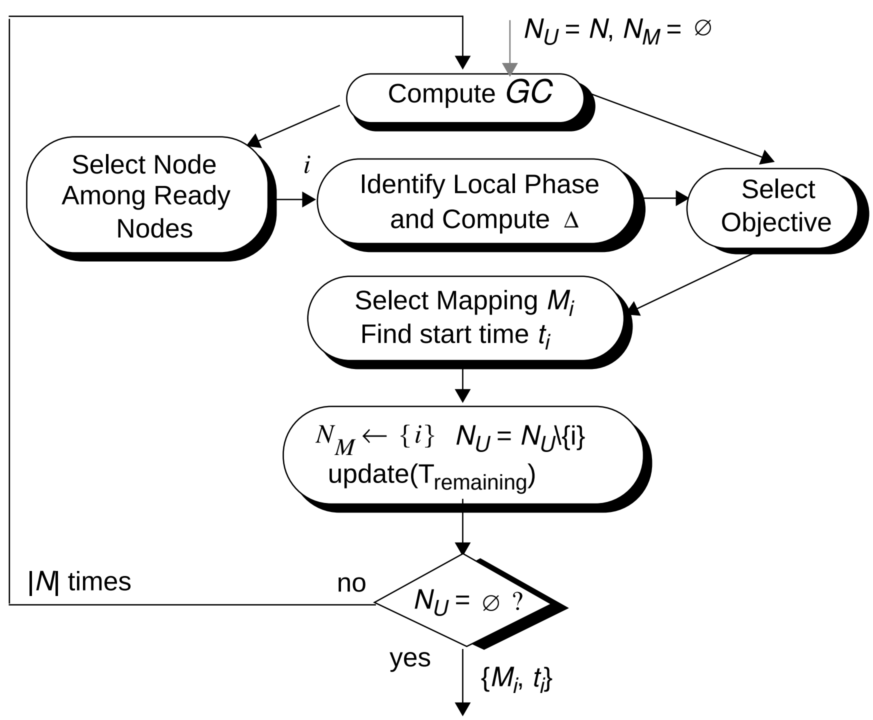

Figure 3.6. The GCLP Algorithm.

selected for mapping from the set of ready nodes using an urgency criterion, i.e., a ready node that lies on the critical path is selected for mapping. Details of this selection are given in Section 3.3.4. The local phase of the selected node is identified and the corresponding local phase delta is computed. The details of this computation will be given in Section 3.3.3. *GC* and the local phase delta are then used to select the mapping objective. Using this objective, the selected node is assigned a mapping (M_i). The mapping is also used to determine the start time for the node (t_i). The process is repeated $|N|$ times until no nodes are left unmapped.

Next, we describe the computation of the global time criticality.

3.3.2 Global Criticality (*GC*)

GC is a global lookahead measure that estimates time criticality at each step of the algorithm. Figure 3.7 illustrates an example used for computing *GC*. At a given step, the hardware/software mapping and schedule for the already mapped nodes is known (Figure 3.7-a). Using this schedule and the required deadline *D*, the remaining time T_{rem} is first determined. Next, all the unmapped nodes (nodes 4 and 5 in this example) are mapped to software and the corresponding finish time T^S is computed, as shown in Figure 3.7-b. Suppose that T^S exceeds the allowed deadline *D*. Some of the unmapped nodes have to be moved from software to hardware to meet the deadline[5]. Define this to be the set $N_{S \to H}$. Suppose that in the example, $N_{S \to H} = \{5\}$. The finish time (T^H) is then recomputed as shown in Figure 3.7-c. *GC* at this step of the algorithm is defined as the fraction of unmapped nodes that have to be moved from software to hardware, so as to meet feasibility. A high value of *GC* indicates that many as-yet unmapped nodes need to be mapped to hardware so as to get a feasible solution, or in other words, time as a resource is more critical.

5. Assuming there is at least one feasible solution to *P1* satisfying the deadline constraint.

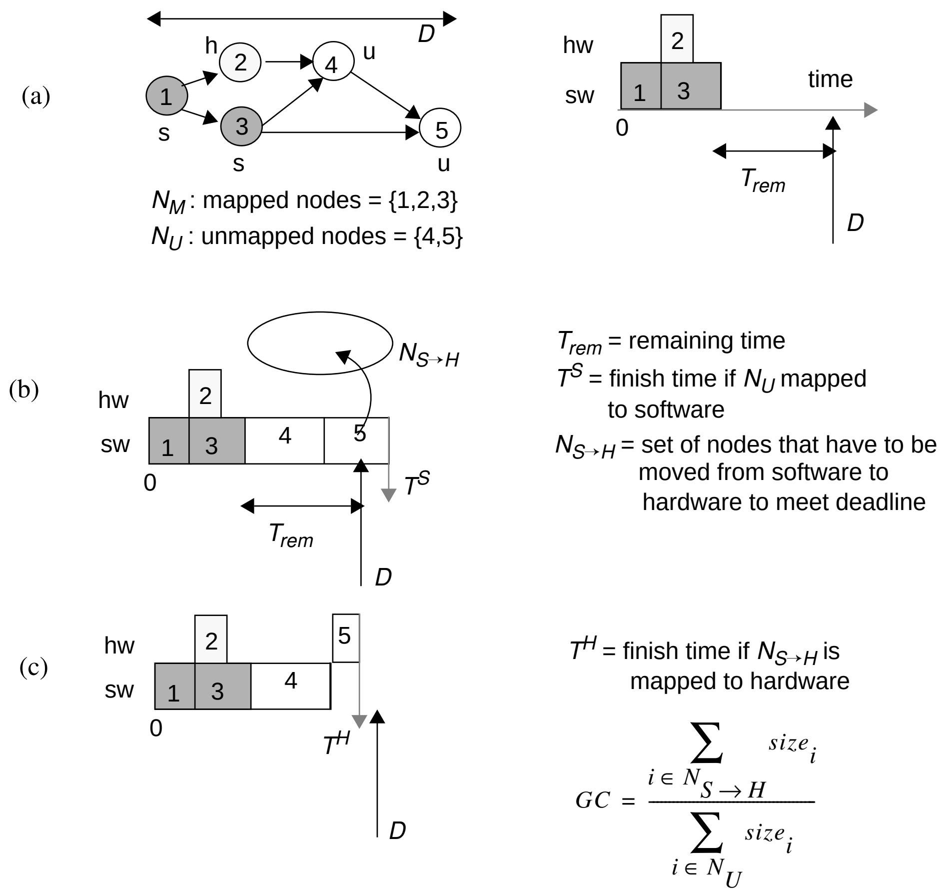

Figure 3.7. Computation of Global Criticality

GC is thus a measure of global time criticality at each step. The following procedure summarizes the computation of *GC*.

<u>Procedure:</u> **Compute_GC**

<u>Input:</u> Mapped (N_M) and Unmapped (N_U) nodes, D, ts_i, th_i, $size_i$, $\forall i \in N$

<u>Output:</u> *GC*

S1. Find the set $N_{S \to H}$ of unmapped nodes that have to be moved from software to hardware to meet deadline D

S1.1. Select a set of nodes in N_U, using a priority function *Pf*, to move from software to hardware

S1.2. Compute the actual finish time (T^H) based on these $N_{S \to H}$ nodes being mapped to hardware

S1.3. If $T^H > D$ go to S1.1

S2. $$GC = \frac{\sum_{i \in N_{S \to H}} size_i}{\sum_{i \in N_U} size_i}, \; 0 \le GC \le 1$$

In S1.1, the set of nodes to be moved to hardware is selected on the basis of a priority function *Pf*. One obvious *Pf* is to rank the nodes in the order of decreasing software execution times ts_i. A second possibility is to use (ts_i/th_i) as the function to rank the nodes. This has the effect of first moving nodes with the greatest relative gain in time when moved to hardware. A third possibility is to rank the nodes in increasing order of ah_i; nodes with smaller hardware area are moved out of software first. Our experiments indicate that the (ts_i/th_i) ordering gives the best results.

S1.2 determines whether moving this set $N_{S \to H}$ to hardware meets feasibility by computing the actual finish time T^H. The finish time can be computed by an $O(|A| + |N|)$ algorithm, as shown in Appendix A4. If the result is infeasible, additional nodes are moved by repeating steps S1.1 to S1.3.

GC is computed in S2 as a ratio of the sum of the sizes of the nodes in $N_{S \to H}$ to the sum of the sizes of the nodes in N_U. The size of a node is taken to be the number of elementary operations (add, multiply, etc.) in the node. Each node is represented by its size since nodes can be heterogeneous in general.

As indicated earlier, *GC* is a measure of global time criticality; a high *GC* indicates a high global time criticality. *GC* has yet another interpretation; it is a measure of the probability (simplistically speaking) that any unmapped node is mapped to hardware. This probability may change at each step of the algorithm.

3.3.3 Local Phase (LP)

GC is an averaged measure over all the unmapped nodes at each step. This desensitizes *GC* to the local properties of the node being mapped. To emphasize the local characteristics of nodes, we classify nodes as *extremities* (or local phase 1 nodes), *repellers* (or local phase 2 nodes), or *normal* nodes (or local phase 3 nodes).

3.3.3.1 Motivation for Local Phase Classification

Since nodes are at a *task* level of granularity, they are likely to exhibit area and time heterogeneity in hardware and software mappings. Nodes that consume a disproportionately large amount of resource on one mapping as compared to the other mapping are called *extremities* or local phase 1 nodes. For instance, a hardware extremity requires a large area when mapped to hardware, but could be implemented inexpensively in software. The mapping preference of such nodes, quantified by an *extremity measure*, modifies the threshold used in *GC* comparison.

Once a feasible solution is obtained, it is usually possible to further swap nodes between hardware and software so as to reduce the allocated hardware area. GCLP uses the concept of *repellers* or local phase 2 nodes to perform *on-line swaps* (as opposed to post-mapping swaps) of similar nodes between hardware and software. To do this, we identify certain intrinsic nodal properties (called repeller properties) that reflect the inherent suitability of a node to either a hardware or a software mapping. For instance, bit operations are handled better in hardware, while memory operations are better suited to software. As a result, a node with many bit manipulations, relative to other nodes, is a software repeller, while a node with a lot of memory operations, relative to other nodes, is a hardware repeller. Moving a node with many bit manipulation operations out of software is thought of as generating a *repelling force* from the software mapping. Hence the proportion of bit manipulation operations in a node is a software repeller property. Similarly, the proportion of memory operations in a node is a hardware repeller property. A repeller property is quantified by a *repeller value*. The combined effect of all the repeller properties in a node is expressed as the *repeller measure* of the node. All nodes are ranked according to their repeller measures. Given two nodes N1 and N2 with similar software characteristics, if N1 has a higher software repeller measure than N2, and given the choice of mapping one of them to hardware, N1 is preferred. The details of the computation of the repeller measure are deferred to Section 3.3.3.3.

Let us see how the repeller measures effect an *on-line swap* in GCLP. Suppose that the deadline constraint demands that only one of nodes N1 and N2 be mapped to hardware. In this case, the node with a smaller hardware area (say N1) should be selected for hardware mapping. Consider the scenario when N1 is mapped at an earlier step than N2 (due to the serial traversal of the graph). If time is not critical early on when N1 is mapped, mapping based on *GC* alone might map N1 to software. Later as time becomes critical, N2 is mapped to hardware. N1 is, however, a better candidate for hardware mapping than N2. In general, the preferences of all the nodes get modified by serial traversal and possibly suboptimal mapping choices are made. One way to address this is to swap nodes across mappings after the algorithm is done. Alternatively, we use the repeller measure of the node being mapped as an *on-line bias* to modify the threshold used in *GC* comparison. To use our example, the software repeller measure of N1 is used to bias its mapping out of software (towards hardware), thereby making it possible for the yet-unassigned node N2 to get mapped to software. The net effect of such on-line swaps is a reduction in the total hardware area.

Normal nodes (local phase 3) do not modify the default threshold value; their mapping is determined by *GC* consideration only.

In the following sections, we outline procedures to classify nodes into local phases and quantify their local phase measures.

3.3.3.2 Local Phase 1 or Extremity nodes

The bottleneck resource in hardware is area, while the bottleneck resource in software is time. Extremities are nodes that consume a disproportionately large amount of the bottleneck resource on a particular mapping (relative to the other mapping).

A *hardware extremity* node is defined as a node that consumes a large area in hardware, but a relatively small amount of time in software. A *software extremity* node is defined as a node that takes up a large amount of time in software, but a relatively small amount of area when mapped to hardware. The rationale in moving a hardware (software) extremity node to software (hardware) is obvious.

The disparity in the resources consumed by an extremity node i is quantified by an *extremity measure* E_i. The extremity measure is used to modify the threshold to which *GC* is compared when selecting the mapping objective (as shown in Figure 3.5); i.e., E_i is the local phase delta for an extremity node.

Extremity Measure

We now describe a procedure to identify extremity nodes in a graph and compute the extremity measure E_i for all such nodes.

Procedure: **Compute_Extremity_Measure**

Input: ts_i, ah_i, $\forall i \in N$, α, β percentiles

Output: E_i, $\forall i \in N$, $-0.5 \le E_i \le 0.5$

S1. Compute the histograms of all the nodes with respect to their software execution times (ts_i) and hardware areas (ah_i)

S2. Determine $ts(\alpha)$ and $ah(\beta)$ that correspond to α and β percentiles of the *ts* and *ah* histograms respectively

S3. Classify nodes into software and hardware extremity sets EX_s and EX_h respectively:

If ($ts_i \ge ts(\alpha)$ and $ah_i < ah(\beta)$), $i \in EX_s$ (software extremity)

If ($ah_i \ge ah(\beta)$ and $ts_i < ts(\alpha)$), $i \in EX_h$ (hardware extremity)

S4. Determine the extremity value x_i for node i:

If $i \in EX_s$, $x_i = \dfrac{ts_i/ts_{max}}{ah_i/ah_{max}}$, else $x_i = \dfrac{ah_i/ah_{max}}{ts_i/ts_{max}}$

where $ts_{max} = max_i\{ts_i\}$ and $ah_{max} = max_i\{ah_i\}$

S5. Order the nodes in EX_s (EH_h) by x. Denote the maximum and minimum extremity values as xs_{max} (xh_{max}) and xs_{min} (xh_{min}) respectively.

S6. Compute the extremity measure E_i for node i:

If $i \in EX_s$, $E_i = -0.5 \times \dfrac{x_i - xs_{min}}{xs_{max} - xs_{min}}$, $-0.5 \le E_i \le 0$

else if $i \in EX_h$, $E_i = 0.5 \times \dfrac{x_i - xh_{min}}{xh_{max} - xh_{min}}$, $0 \le E_i \le 0.5$

In S1, we compute a distribution of the nodes with respect to their software execution times ts_i and hardware areas ah_i. Parameters α and β represent percentile cut-offs for these distributions. For instance, in S3, a node i is classified as a software extremity node if it lies above α percentile in the *ts* histogram ($ts_i > ts(\alpha)$) and below β percentile in the *ah* histogram ($ah_i < ah(\beta)$). Similarly, a node i is classified as a hardware extremity if it lies above β percentile in the *ah* histogram ($ah_i > ah(\beta)$) and below α percentile in the *ts* histogram ($ts_i < ts(\alpha)$). Figure 3.8 shows typical histograms and the identification of

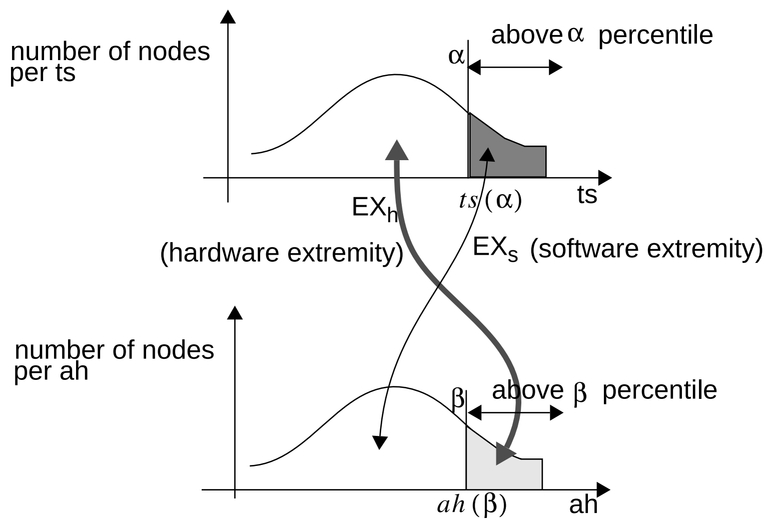

Figure 3.8. Hardware (EX_h) and software (EX_s) extremity sets

extremities. For the examples that we have considered, values of α and β in the range (0.5, 0.75) are used. A value outside this range tends to reduce the number of nodes that fit this behavior and consequently the extremities do not play a significant role in biasing the local preferences of nodes. The extremity value of a node is computed in S4. The extremity measure E_i of a node i is computed in S6, $-0.5 \leq E_i \leq 0.5$.

Threshold Modification using the Extremity Measure

Let GC_k denote the value of GC at step k when an extremity node i is to be mapped. If E_i is ignored, the threshold assumes its set value of 0.5. Since GC_k is averaged over all unmapped nodes, mapping of node i in this case is based just on GC_k. This leads to:

1. Poor mapping: Suppose node i is a hardware extremity. If $GC_k \geq 0.5$, Obj1 is selected in Figure 3.5 (minimize time), and i could get mapped to hardware based on time-criticality. However, i is a hardware extremity and mapping it to hardware is an obviously poor choice for *P1*.
2. Infeasible mapping: Suppose node i is a software extremity. If $GC_k < 0.5$,

Obj2 is selected in Figure 3.5 (minimize area) and *i* could get mapped to software. Node *i* is a software extremity, however, and mapping it to software could exceed the deadline.

To overcome these problems, the extremity measure E_i is used to modify the default threshold in the direction of the preferred mapping. The new threshold is 0.5 + E_i. GC_k is compared to this modified threshold. For software extremities, $-0.5 \le E_i \le 0$, so that $0 \le \text{Threshold} \le 0.5$, and for hardware extremities, $0 \le E_i \le 0.5$, so that $0.5 \le \text{Threshold} \le 1$.

3.3.3.3 Local Phase 2 or Repeller Nodes

The use of repellers to effect on-line swaps and reduce the overall hardware area was discussed in Section 3.3.3.1. In this section, we quantify a repeller node with a repeller measure and describe its use in GCLP.

Several repeller properties can be identified for each node. Bit-level instruction mix and precision level are examples of software repeller properties; while memory-intensive instruction mix and table-lookup instruction mix are possible hardware repeller properties. Each property is quantified by a property value. The cumulative effect of all the properties of a node is expressed by a repeller measure.

Let us consider the **bit-level instruction mix**, a software repeller property, in some detail. This property is quantified through its property value called *BLIM*. $BLIM_i$ is defined as the ratio of bit-level instructions to the total instructions in a node *i* ($0 \le BLIM_i \le 1$). For instance, consider the DAG shown in Figure 3.9-a. Suppose that node 2 in the graph is an IIR filter and node 5 is a scrambler. Figure 3.9-b shows the hypothetical *BLIM* values plotted for all the nodes in the DAG in Figure 3.9-a. Node 5, the scrambler, has a high *BLIM* value. Node 2, the IIR filter,

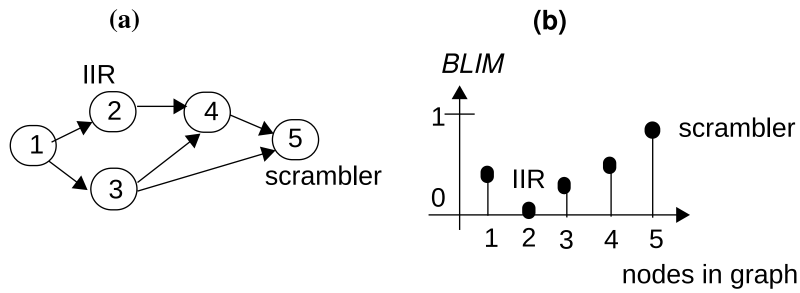

Figure 3.9. An example repeller property.

does not have any bit manipulations, and hence its *BLIM* value is 0. The higher the *BLIM* value, the worse is the suitability of a node to software mapping.

Consider two nodes N_1 and N_2, with software (hardware) areas as_1 (ah_1) and as_2 (ah_2) respectively. Suppose $BLIM_1 > BLIM_2$. Now, if $as_1 \approx as_2$, then $ah_1 < ah_2$ (because bit-level operations can be typically done in a smaller area in hardware). Thus N_1 is a software repeller relative to N_2, based on the bit-level instruction mix property. Based on the discussion in Section 3.3.1, given the choice of mapping one of N1 or N2 to hardware, N1 is preferred for hardware mapping on the basis of the *BLIM* property.

Other repeller properties mentioned earlier are similarly quantified through their property values. The cumulative effect of all the repeller properties in a node is considered when mapping a node. The *repeller measure* R_i of a node captures this aggregate effect. It is expressed as a convex combination of all the repeller property values of the node. The repeller measure is used to modify the threshold against which *GC* is compared when selecting the mapping objective (as shown in Figure 3.5); i.e., R_i is the local phase delta for repeller nodes.

Repeller Measure

The procedure outlined below describes the computation of the repeller measure (R_i) for each node i. Let *RH* be the set of hardware repeller properties, and *RS* be the set of software repeller properties. Let $P = RH \cup RS$ be the complete set of repeller properties.

Procedure: **Compute_Repeller_Measure**

Input: $v_{i,p}$ = value of repeller property p for node i, $i \in N$, $p \in P$

Output: Repeller measure R_i, $\forall i \in N$, $-0.5 \le R_i \le 0.5$

S1. Compute for each property p:

$\sigma^2(v_{i,p})$ = variance of $v_{i,p}$ over all i

$min(v_{i,p})$ = minimum of $v_{i,p}$ over all i

$max(v_{i,p})$ = maximum of $v_{i,p}$ over all i

Let $RX = RH$ *if* $p \in RH$ *or* $RX = RS$ if $p \in RS$

$$a_p = \frac{\sigma^2(v_{i,p})}{\sum_{p \in RX} \sigma^2(v_{i,p})} \text{ = weight of repeller property } p, \quad \sum_{p \in RX} a_p = 1,$$

S2. Compute the normalized property value $nv_{i,p}$ for each property p, of node i

$$nv_{i,p} = \frac{v_{i,p} - min(v_{i,p})}{max(v_{i,p}) - min(v_{i,p})}, \quad 0 \le nv_{i,p} \le 1 .$$

S3. Compute the repeller measure R_i for each node i

$$R_i = \frac{1}{2} \cdot \left(\sum_{p \in RH} a_p \cdot nv_{i,p} - \sum_{p \in RS} a_p \cdot nv_{i,p} \right), \quad -0.5 \le R_i \le 0.5 .$$

The value $v_{i,p}$ of each repeller property p for node i is obtained by analyzing the node. For instance, consider the bit-level instruction mix property. The bit-level operations (such as OR, AND, EXOR) are first identified in the node. The *BLIM* value of a node i is simply the ratio of the number of bit-level operations to the total number of operations in that node. Other repeller property values are similarly computed.[6]

In S1 of the above procedure, the variance, minimum, and maximum of each repeller property value are computed. The property values are normalized in S2. In S3, the repeller measure for each node is computed as a convex combination

6. Guerra *et al.* [Guerra94] present techniques for the quantification of algorithmic properties such as operator concurrency, temporal density, and regularity.

of the normalized repeller property values. The weight a_p of a property p is proportional to the variance of its value. This deemphasizes properties with small variances in their values. When the repeller measure is used to swap repeller nodes with comparable property values, there is hardly any area reduction; they are not worth swapping. The variance weight ensures this.

Threshold Modification using the Repeller Measure

As described in Section 3.3.3.1, repellers constitute an *on-line swap* to reduce the overall hardware area — a post-mapping swap is avoided. Recall that the repeller measure is a measure of the swapping gain of two similar local phase 2 nodes between hardware and software mappings. Given a choice of mapping just one of two nodes to hardware, the node with a higher software repeller measure is chosen.

Given a phase 2 node i with a sufficiently high software repeller measure, the algorithm tries to achieve a relative shift of i out of software. It modifies the threshold such that the objective selected will favor the complementary mapping. This swap frees up the current resource for an as-yet unscheduled node with a lower repeller property measure, thus reducing the overall allocated hardware area.

The repeller measure R_i is used to modify the threshold so that the new threshold is 0.5 + R_i. For software repellers, $-0.5 \le R_i \le 0$, so that $0 \le \text{Threshold} \le 0.5$, and for hardware repellers, $0 \le R_i \le 0.5$ so that $0.5 \le \text{Threshold} \le 1$.

3.3.3.4 Local Phase 3 or Normal Nodes

A node that is neither an extremity nor a repeller is defined to be a **local phase 3** node or a **normal** node. The threshold is set to its default value (0.5) when a normal node is mapped. Thus the mapping objective is governed by GC alone.

In summary, nodes are classified into three disjoint sets: extremity nodes, repeller nodes, and normal nodes. The local preference of each node is quantified by its measure, represented by a *local phase delta* (Δ). In particular, $\Delta = E_i$ for extremity nodes, $\Delta = R_i$ for repeller nodes, and $\Delta = 0$ for normal nodes. This local phase delta is used to compute the modified threshold: threshold = $0.5 + \Delta$.

3.3.4 GCLP Algorithm

Algorithm: **GCLP**

Input: ah_i, as_i, th_i, ts_i, E_i (extremity measure), and R_i, (repeller measure) $\forall i \in N$
Communication costs: ah_{comm}, as_{comm}, and t_{comm}, and constraints: AH, AS, and D.

Output: Mapping M_i ($M_i \in$ {hardware, software}), start time t_i, $\forall i \in N$

Initialize: N_U = {unmapped nodes} = N, N_M = {mapped nodes} = ϕ.

Procedure:

while $\{|N_U| > 0\}$ {

S1. Compute GC

S2. Determine N_R, the set of *ready* nodes

S3. Compute the effective execution time $t_{exec}(i)$ for each node i

If $i \in N_U$ $\quad t_{exec}(i) = GC \cdot th_i + (1\text{-}GC) \cdot ts_i$

else if $i \in N_M$ $\quad t_{exec}(i) = th_i \cdot I(M_i == \text{hw}) + ts_i \cdot I(M_i == \text{sw})$[7]

S4. Compute the longest path $longestPath(i)$, $\forall i \in N_R$ using $t_{exec}(i)$

S5. Select node i, $i \in N_R$, for mapping: max($longestPath(i)$)

S6. Determine mapping M_i for i:

S6.1. if $(E_i \neq 0)$ $\quad \Delta = \gamma \cdot E_i$ (local phase 1)
where γ is extremity measure weight, $0 \le \gamma \le 1$

else if$(R_i \neq 0)$ $\quad \Delta = \nu \cdot R_i$ (local phase 2)
where ν is repeller measure weight, $0 \le \nu \le 1$

else $\quad \Delta = 0$; (local phase 3)

7. $I(expr)$ is an indicator function that evaluates to 1 when $expr$ is true, 0 else.

S6.2. Threshold $= 0.5 + \Delta$, $0 \leq \text{Threshold} \leq 1$

S6.3. If $(GC \geq \text{Threshold})$ m: minimize(*Obj1*);
else m: minimize(*Obj2*);

S6.4. $M_i = m$; Set(t_i); $N_U = N_U \backslash \{i\}$; $N_M \leftarrow \{i\}$,[8]
Update($T_{remaining}$, $AH_{remaining}$, $AS_{remaining}$);

}

The algorithm maps one node per step. In S1, *GC* is computed using the procedure described in Section 3.3.2. In S2, we determine the set of ready nodes, i.e., the set of unmapped nodes whose predecessors have been mapped. One of these ready nodes is selected for mapping in S5. In particular, we select the node on the maximum longest path, the critical path of the graph. The critical path generally involves unmapped nodes; computing it can present problems since the execution times of such nodes is not known at the current step. To overcome this difficulty, we define the effective execution time ($t_{exec}(i)$) of an unmapped node i as the mean execution time of the node, assuming it is mapped to hardware with probability *GC* and to software with probability (1-*GC*). Here we use the notion of *GC* as a node-invariant hardware mapping probability (see Section 3.3.2). In S6, the mapping and schedule are determined for the selected node. If the node is an extremity (or a repeller) its extremity (or repeller) measure is used to modify the threshold. The contribution of the extremity and repeller measures can be varied by weighting factors γ and ν. In Section 3.8.4, we discuss the tuning of these weights. The mapping objective is selected in S6.3 by comparing *GC* against the threshold. If time is critical, an objective that minimizes the finish time is selected, otherwise one that minimizes resource consumption is selected. The objective functions are:

8. Using set-theoretic notation, $N_U = N_U \backslash \{i\}$ means element i is deleted from set N_U, and $N_M \leftarrow \{i\}$ means element i is added to set N_M.

Obj1: $t_{fin}(i, m)$, where $m \in$ {software, hardware}

$$t_{fin}(i,m) = max(max_{P(i)}(t_{fin}(p) + t_c(p,i)), tf_{last}(m)) + t(i,m)$$

where

$P(i)$ = set of predecessors of node i, $p \in P(i)$

$t_{fin}(p)$ = finish time of predecessor p

$t_c(p,i)$ = communication time between predecessor p and node i

$tf_{last}(m)$ = finish time of the last node assigned to mapping m

= 0 if m corresponds to hardware

$t(i,m)$ = execution time of node i on mapping m

Obj2: $$\frac{(as_i + as_{comm}^{tot})}{AS} \cdot I(m = sw) + \frac{(ah_i + ah_{comm}^{tot})}{AH_{remaining}} \cdot I(m = hw)$$

Obj1 selects a mapping that minimizes the finish time of the node. A node can begin execution only after all of its predecessors have finished execution and the data has been transferred to it from its predecessors. Also, a node cannot begin execution on the software resource until the last node mapped to software has finished execution.

Obj2 uses a "percentage resource consumption" measure. This measure is the fraction of the resource area of a node (nodal area plus communication area) to the total resource area. The area ah_{comm}^{tot} (as_{comm}^{tot}) takes into account the total cost of communication (glue logic in hardware and code in software) between node i in hardware (software) and all its predecessors. For the hardware resource, the resource area required by the node is divided by the available hardware area ($AH_{remaining}$). Obj2 thus favors software allocation as the algorithm proceeds.

As shown in Appendix A4, GCLP has a quadratic complexity in the number of nodes. The performance of the algorithm is analyzed in the next section.

3.4 Performance of the GCLP Algorithm

The examples used to analyze the algorithm performance are described in Section 3.4.1. We present two sets of experiments. The first experiment (Section 3.4.2) is a comparison of the solution obtained with GCLP to the optimal solution generated by an ILP formulation (described in Appendix A1). The second experiment (Section 3.4.3) demonstrates the effectiveness of classifying nodes into extremities and repellers. Section 3.4.4 discusses the algorithm behavior with the help of an example trace.

3.4.1 Examples

Two classes of examples are used to analyze the performance of the algorithm: practical examples, and random graphs.

Practical Examples

We consider practical signal processing applications with periodic timing constraints. Two examples are used: 32KHz 2-PSK modem, and 8 KHz bidirectional telephone channel simulator (TCS). These applications are specified as SDF graphs in the Ptolemy [Buck94a] environment. Figure 3.10 shows the receiver section of the modem example. A DAG is generated from the SDF graph representation. Nodes in the DAG are at a task level of granularity. Typical nodes in the modem include carrier recovery, timing recovery, equalizer, descrambler, etc. in the receiver section, and pulse shaper, scrambler etc. in the transmitter section. The nodes in the TCS include linear distortion filter, Gaussian noise generator, harmonic generator, etc. In the modem and TCS examples considered, the DAGs consist of 27 and 15 nodes respectively[9].

9. Much larger examples have been easily solved with the GCLP algorithm (as will be shown in Section 3.4.3), Here, we consider these relatively small examples that can be solved by ILP as well. The intent is to compare the GCLP solution to the optimal ILP solution to evaluate the quality of the heuristic.

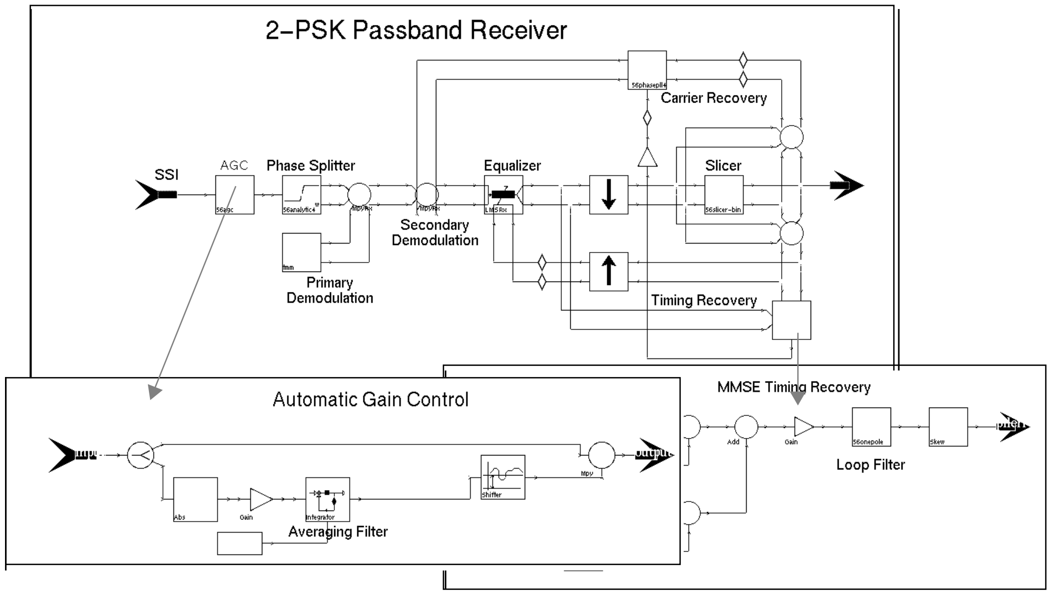

Figure 3.10. Receiver section of a modem, described in Ptolemy. Hierarchical descriptions of the AGC and timing recovery blocks are shown.

The area and time estimates (in hardware and software mappings) for each node in these DAGs are obtained by using the Ptolemy and Hyper environments (Figure 3.11). We assume a target architecture consisting of: (1) Motorola DSP 56000 [Motorola] for the software component, (2) standard-cell based custom hardware generated by Hyper [Rabaey91], and (3) self-timed memory-mapped I/O

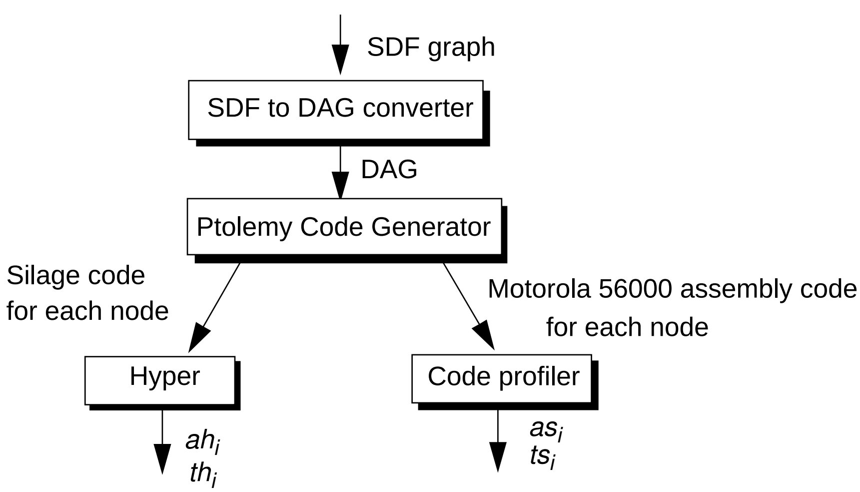

Figure 3.11. Estimation of area and time values in hardware and software.

for hardware-software communication. The code generation feature of Ptolemy is used to synthesize 56000 assembly code [Pino95a] and Silage code [Kalavade93] for each node in the DAG. Estimates of the software area (as_i) and software execution time (ts_i) for each node i are obtained by using simple scripts that analyze the generated DSP 56000 assembly code. The Silage code for each node i is input to Hyper, which generates estimates of the hardware execution time (th_i) and hardware area (ah_i) for the node. The hardware execution time is computed as the best-case execution time (corresponding to the critical path of the control-dataflow graph associated with the node[10]). The hardware area is computed by setting the sample period to the critical path. Further details of the estimation mechanisms can be found in Appendix A8.

Random Examples

A random graph generator is used to generate a graph with a random topology for a given number of nodes (graph size). The hardware-software area and time estimates of the nodes in the random graph are generated by taking into account the trend observed in real examples. Details of the techniques used to generate the random graphs are given in Appendix A7. For each size, we generate 10 random graphs differing in topology and area and time metrics. The heuristic is applied for each random graph and the average value of the result is reported for that size.

3.4.2 Experiment 1: GCLP vs. ILP

Examples are first partitioned using the GCLP algorithm. The ILP formulation for these examples is then solved using the ILP solver CPLEX[11]. Table 2 lists the GCLP and ILP solutions for the modem and TCS examples. The total hard-

10. This control-dataflow graph is generated by Hyper during the hardware synthesis process.
11. CPLEX is a commercially available optimization software.

example	size	algorithm	**total hardware area** (normalized with respect to hardware area required in ILP solution)	**DSP utilization** (1 - idle_time/D)*100	solution time
modem	27	ILP	1.0	93.8%	19190 s
		GCLP	1.1935	84.89%	0.535 s
TCS	15	ILP	1.0	73.5%	6656 s
		GCLP	1.0	73.5%	0.387 s

Table 2. Results from ILP and GCLP algorithm.

ware area (normalized with respect to the optimal solution), DSP utilization, and solution time for all the cases are compared. The solution time represents the CPU time required to generate the solution on a SPARCstation 10. The total hardware area obtained with the GCLP algorithm is quite close to the ILP solution. In the modem example, the GCLP mapping has all but one node identical with the ILP mapping. The GCLP mapping for the TCS is identical to the ILP mapping.

Figure 3.12 compares the total hardware area obtained with the GCLP algorithm to the optimal solution obtained with ILP formulation for a number of random examples. For all tested examples the generated GCLP solution is within **30%** of the optimal solution. Examples larger than 20 nodes could not be solved by ILP in reasonable time[12]. GCLP has been used to solve examples with up to

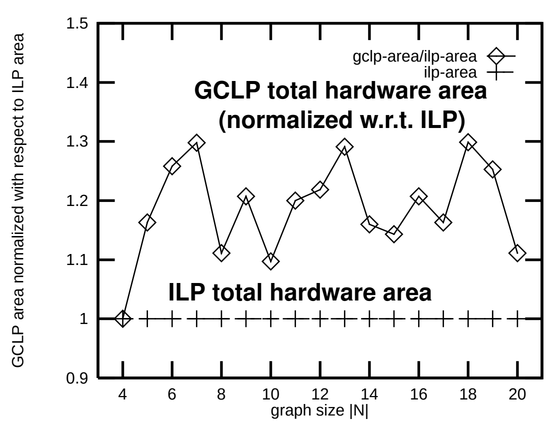

Figure 3.12. GCLP vs. ILP: Random examples

500 nodes with relative ease.

3.4.3 Experiment 2: GCLP with and without local phase nodes

To examine the effect of local phase nodes on the GCLP performance, the GCLP algorithm is applied under three cases:

Case 1. The local phase classification is not used — all nodes are normal nodes with $\Delta = 0$. The objective function is selected by comparing *GC* with the default threshold = 0.5.

Case 2. Nodes are classified as either repellers or normal nodes. For repeller nodes $\Delta = R_i$, otherwise $\Delta = 0$.

Case 3. Complete local phase classification, using extremities, repellers, and normal nodes. For extremity nodes $\Delta = E_i$, for repeller nodes $\Delta = R_i$, and for normal nodes $\Delta = 0$.

In the modem example, it was found that:

1. Repellers reduce the hardware area through on-line swaps (the solution obtained in case 2 is 13% smaller than the solution obtained in case 1).
2. Extremities are seen to match their expected mappings (ex: *Pulse Shaper,* a hardware extremity node, is mapped to software. *Carrier Recovery,* a software extremity node, is mapped to hardware).
3. Repeller nodes are also mapped to their intuitively expected mappings (ex: *Scrambler*, a high *BLIM* software repeller, is mapped to hardware).

Figure 3.13 plots the results of these three cases when applied to random examples. The total hardware areas are normalized with respect to the total hardware area obtained in case 1. It is seen that the use of repellers (case 2) significantly reduces the hardware area as compared to a purely *GC*-based selection in

12. Some fine-tuning of the ILP formulation could improve the ILP solution time slightly.

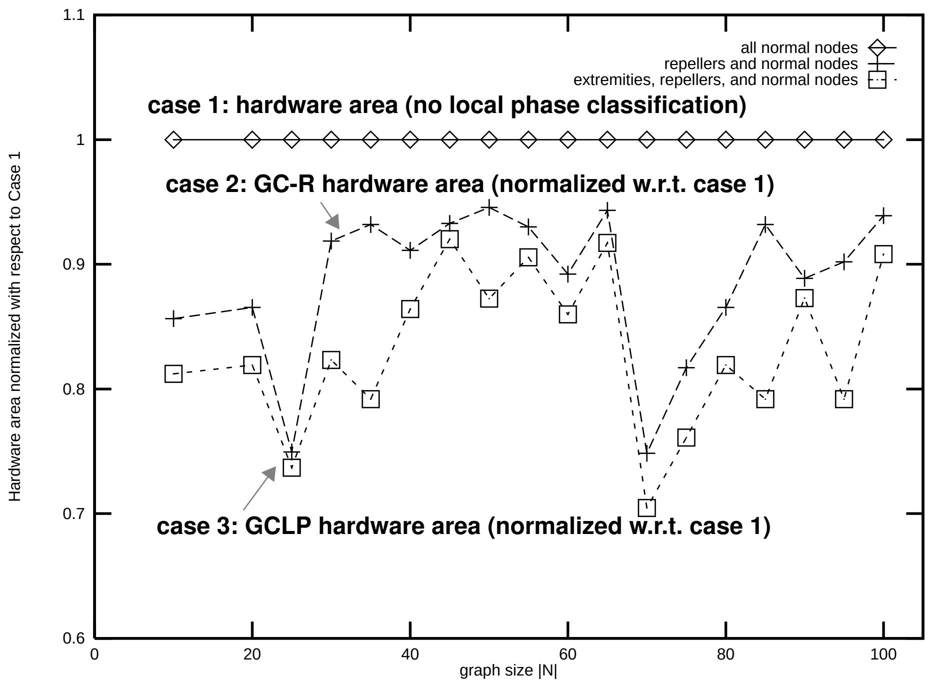

Figure 3.13. Effect of local phase classification on GCLP performance: case 1: mapping based on GC only, case 2: threshold modified for repellers, case 3: complete local phase classification.

case 1. This verifies our premise that repellers effect on-line swaps to reduce the total hardware area. Using both extremity and repeller nodes further improves the quality of the solution. On an average, the complete classification of local phase nodes reduces the total hardware area by **16.82%**, when compared with case 1.

3.4.4 Algorithm Trace

The behavior of GCLP with complete local phase classification of nodes is quite complex. To understand the relation between node classification, threshold, *GC*, and the actual mapping at each step of the algorithm, we illustrate these key parameters in algorithm traces for specific examples.

Figure 3.14 illustrates the *GC* variation and the mapping at each step, ignoring the local phase classification. When $GC > 0.5$, time is critical. Almost always, nodes get mapped to hardware. Eventually, this reduces the time criticality

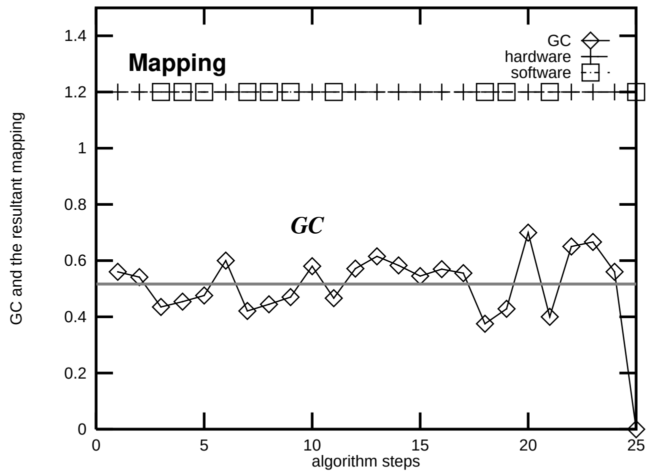

Figure 3.14. Algorithm trace: GC and the Mapping. All normal nodes, threshold = 0.5.

and *GC* reduces. When it drops below 0.5, area minimization is selected as the objective. In most cases, this objective selects software mapping. Subsequently, time becomes critical, *GC* increases, and further nodes get mapped to hardware. Thus *GC* (and the mapping) adapts continually.

Figure 3.15 illustrates the effect of adding extremity nodes to the above example. Nodes mapped in steps 8 and 10 are software extremities. In Figure 3.15-a, the extremity measure is not considered, in Figure 3.15-b, it is. We assume that nodes are mapped in the same order in both these cases. Mapping marked by a cross means software mapping and by a square means hardware mapping. In Figure 3.15-a, the extremity measures are not considered; the threshold assumes its default value (0.5). The software extremity node mapped in step 8 (marked *e1*) gets mapped to software, hence, time criticality increases. When compared to the corresponding mapping in Figure 3.15-b, nodes mapped subsequently in steps 15 and 16 (marked *h1*) get mapped to hardware. A similar effect is observed while mapping nodes in steps 22, 23, and 24 (marked *h2* in Figure 3.15-a). The generated solution has a total hardware area of 1253, and 14 nodes are mapped to hardware. Also, the sample mean of the *GC* over all steps is 0.50448. In Figure 3.15-b, extremity measures are used to change the default threshold. The extremity measure for nodes mapped in steps 8 and 10 lowers the threshold at the points marked *e1* and *e2* in Figure 3.15-b. This induces a hardware mapping. Mapping these software extremities to hardware reduces time criticality. Subsequently, nodes mapped in steps 15 and 16 (marked *s1*) get mapped to software. The total hardware area is 756, and 10 nodes are mapped to hardware. Thus by taking into account the local preference of nodes 8 and 10, the quality of the solution is improved by 40%. Also,

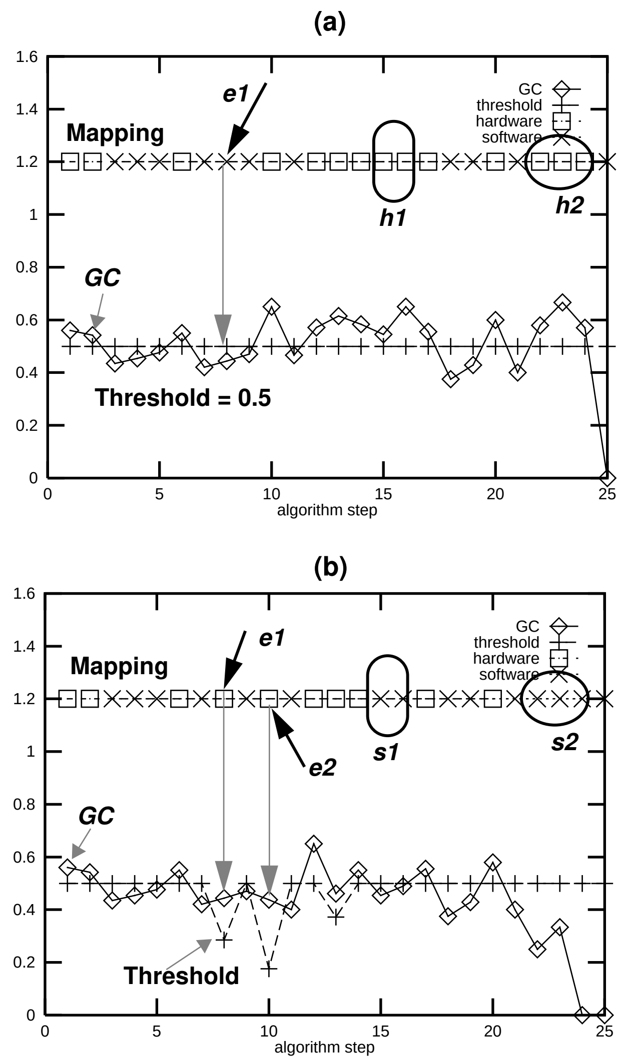

Figure 3.15. *GC* and mapping: (a) without extremity measures (b) with extremity measures.

the sample mean of the *GC* over all the steps is 0.4288.

Figure 3.16 illustrates the effect of repellers on *GC* and mapping at each step of the algorithm applied to a particular example. In this example, nodes mapped in steps 6 and 10 are software repellers; the node mapped in step 6 has a larger repeller measure than the node mapped in step 10. In Figure 3.16-a repellers are not considered, in Figure 3.16-b, they are. We assume that nodes are mapped in the same order in both the cases. In Figure 3.16-a, the repeller measures are not considered when mapping — the threshold assumes its default value of 0.5. In this

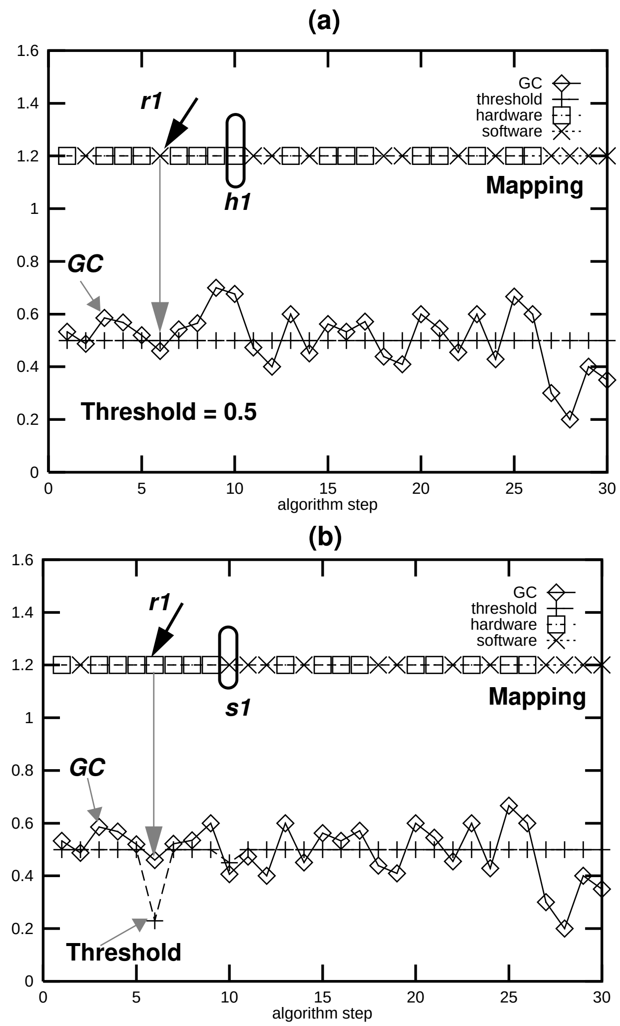

Figure 3.16. GC and mapping: (a) without repeller measures (b) with repeller measures to change the threshold.

case, node mapped in step 6 is mapped to software and the node mapped in step 10 is mapped to hardware. Clearly, this mapping can be improved. In Figure 3.16-b, repeller measures are taken into consideration; they modify the default threshold and hence the mapping. The repeller measure for the node mapped in step 6 lowers the threshold at the point marked *r1* in Figure 3.16-b. This forces the node to get mapped to hardware. The threshold for the node in step 10 is not lowered as much and it gets mapped to software. The mapping of nodes in steps 6 and 10 is thus exactly opposite to that in Figure 3.16-a. Thus nodes in steps 6 and 10 were in effect *swapped on-line* in Figure 3.16-b — the node with a larger repeller measure displaced the one with a smaller measure and this reduced the overall area (1803 in Figure 3.16-a vs. 1677 in Figure 3.16-b, 17 nodes are mapped to hardware in both cases).

3.4.5 Summary of the GCLP algorithm

We have so far discussed the Global Criticality/Local Phase algorithm to solve the binary partitioning problem (*P1*). The key features of the algorithm can be summarized as follows.

Global criticality is a global lookahead measure that quantifies the time criticality at each step of the algorithm, taking into account the currently unmapped nodes and the desired throughput requirements. *GC* is compared with a threshold to select a mapping objective at each step of the algorithm. Nodes that consume disproportionate amounts of resources in hardware and software mappings are classified as extremities. The threshold used for mapping such nodes is modified to account for their mapping preference. The total hardware area is further reduced by using the concept of on-line swaps between repeller nodes. Repeller nodes are classified and quantified on the basis of intrinsic nodal properties.

The threshold used for mapping repeller nodes is modified in accordance with their relative "repulsion" for a mapping. The GCLP algorithm is computationally efficient ($O(|N|^2)$) with a solution quality comparable to the optimal.

3.5 Algorithm for Extended Partitioning: Design Objectives

The GCLP algorithm described so far solves the binary partitioning problem *P1*. The extended partitioning problem *P2* is to jointly optimize the mapping as well as implementation bin for each node. Consider the implementation-bin curve of a node as shown in Figure 3.4. Denote L to be the fastest (left-most) implementation bin, and H to be the slowest (right-most) implementation bin. As the implementation-bin curve is traversed from bins L to H, the hardware area required to implement the node decreases. From the viewpoint of minimizing hardware area, each node mapped to hardware can be set at its H bin (lowest area). This might, however, be infeasible since the H bins correspond to the slowest implementations. The extended partitioning problem is to select an "appropriate" implementation bin and mapping for each node such that the total hardware is minimized, subject to a deadline and resource constraints. This problem is obviously far more complex than the mapping (binary partitioning) problem. Our goal is to design an efficient algorithm to solve the extended partitioning problem. There are two guiding objectives used in the design of this algorithm.

1. <u>Design Objective 1: Complexity that scales reasonably</u>: The binary partitioning problem has $2^{|N|}$ mapping possibilities for $|N|$ nodes in the graph. Given B implementation bins within a mapping, the extended partitioning problem has $(2B)^{|N|}$ possibilities in the worst-case. The algorithm complexity should not scale with the dimensionality (number of design alternatives

per node) of the partitioning process, i.e., if a binary partitioning algorithm has complexity O($|N|^2$), the extended partitioning algorithm should not have complexity O($|N|^{2B}$), since B is typically in the range 5 to 10. Obviously the binary partitioning algorithm cannot be extended directly to solve the extended partitioning problem, since the implementation possibilities explode.

2. Design Objective 2: Reuse of GCLP: Since we already have an efficient algorithm for binary partitioning, the algorithm for extended partitioning should reuse it. This suggests that extended partitioning can be decomposed into two blocks: mapping and implementation-bin selection. GCLP can be used for mapping.

It is not enough, however, to decompose the extended partitioning problem into two isolated steps, namely that of mapping followed by implementation-bin selection. The serial traversal of nodes in a graph means that the implementation bin of a particular node affects the mapping of as-yet unmapped nodes. Since there is a correlation between mapping and implementation-bin selection, they cannot be optimized in isolation. This dependence has to be captured in the algorithm.

Our approach to solving the extended partitioning problem is summarized in Figure 3.17. The heuristic is called MIBS. In the final solution, each node in the graph is characterized by three attributes: mapping, implementation bin, and schedule. As the algorithm progresses, depending on the extent of information that has been generated, each node in the DAG passes through a sequence of three states: (1) free, (2) tagged, and (3) fixed. Before the algorithm begins, all three attributes are unknown. Such nodes are called *free* nodes. Assuming median area and time values, GCLP is first applied to get a mapping and schedule for all the free nodes in the graph. A particular free node (called a *tagged* node) is then

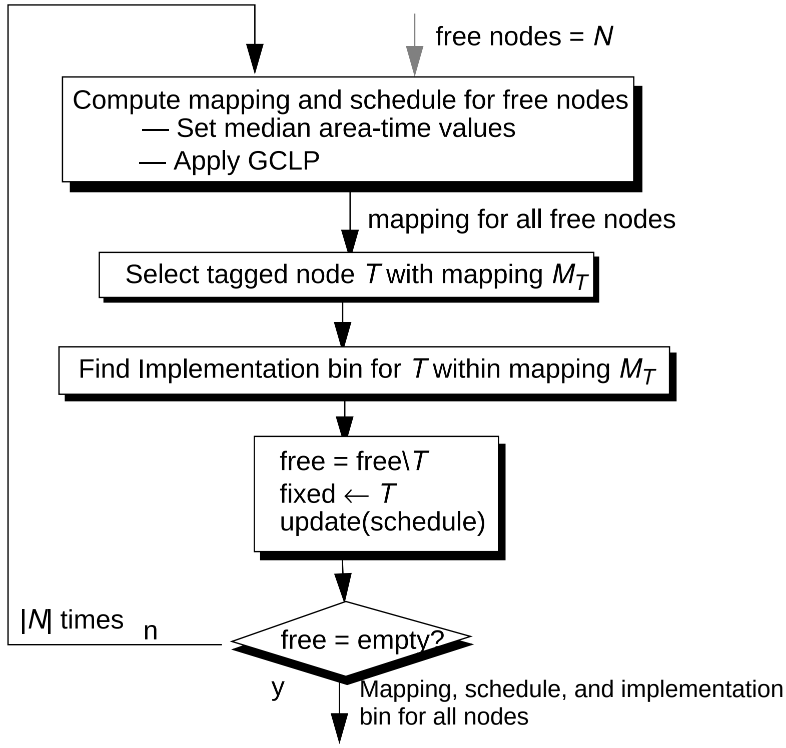

Figure 3.17. MIBS approach to solving extended partitioning

selected. Assuming its mapping to be that determined by GCLP, an appropriate implementation bin is then chosen for the tagged node. In the following section, we describe a bin selection procedure that determines the implementation bin for the tagged node. Once the mapping and implementation bin are known, the tagged node becomes a *fixed* node. GCLP is the applied on the remaining nodes and this process is repeated until all nodes in the DAG become fixed; the MIBS algorithm has $|N|$ steps[13] for $|N|$ nodes in the DAG.

The MIBS approach subscribes closely to the design objectives outlined. GCLP is used for mapping (according to Design Objective 2). Since GCLP and bin selection are applied alternately within each step of the MIBS algorithm, there is continuous feedback between the mapping and implementation-bin selection

13. Each *step* of the MIBS algorithm constitutes the determination of the mapping, implementation bin, and schedule of a node.

stages. The MIBS algorithm will be shown to be reasonably efficient ($O(|N|^3 + B \cdot |N|^2)$, where B is the number of implementation bins per mapping. Thus it scales polynomially with the dimensionality of the problem (Design Objective 1).

In the next section, we describe the bin selection procedure to solve the implementation-bin selection problem.

3.6 Implementation-bin Selection

3.6.1 Overview

In the following, we restrict ourselves to the problem of selecting the implementation bin for hardware-mapped nodes only. The concepts introduced here can be extended to software implementation-bin selection as well.

Recall from Figure 3.17 that, in each step of the MIBS algorithm, GCLP is first applied to determine the revised mapping of free nodes. Let the free nodes mapped to hardware at the current step be called $free^h$ nodes. A tagged node is selected from the set of free nodes. Assuming its mapping to be that determined by GCLP, the bin selection procedure is applied to select an implementation bin for the tagged node.

Figure 3.18 shows the flow of the bin selection procedure. The key idea is to use a lookahead measure to correlate the implementation bin of the tagged node with the hardware area required for the $free^h$ nodes. It selects the most *responsive* bin in this respect as the implementation bin for the tagged node.

Computing the lookahead measure can be very complex since the final implementation bins of the $free^h$ nodes are not known at this step. To simplify matters, we assume that $free^h$ nodes can be in either L or H bins[14]. All $free^h$ nodes are

---

14. A $free^h$ node loses this restriction when it becomes tagged later on.

assumed to be in their H bins initially. The lookahead measure (called *bin fraction* BF_T^j) computes, for each bin j of the tagged node T, the fraction of $free^h$ nodes that need to be moved from H bins to L bins in order to meet timing constraints. A high value of BF_T^j indicates that if the tagged node T were to be implemented in bin j, a large fraction of $free^h$ nodes would likely get mapped to their fast implementations (L bins), hence increasing the overall area. The *bin fraction curve* (BFC_T) is the collection of the all bin fraction values of the tagged node T.

Bin sensitivity is the gradient of BFC_T. It reflects the responsiveness of the bin fraction to the bin motion of node T. Suppose that the maximum slope of the bin fraction curve is between bins k-1 and k (Figure 3.18). Moving the tagged node from bin k-1 to k shifts the largest fraction of $free^h$ nodes to their L bins. Equivalently, the k to k-1 motion for the tagged node results in the largest reduction of the area of $free^h$ nodes. Hence the (k-1)th bin is selected as the implementation bin for the tagged node (B_T*). The computation of the BFC and bin sensitivity is described next.

The notation used in the bin selection procedure is summarized Table 3.

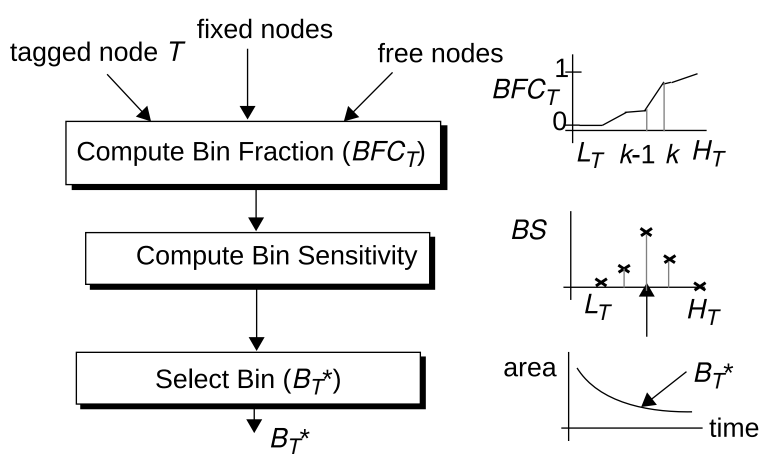

Figure 3.18. The bin selection procedure

Notation	Interpretation
T	Tagged node
fixed nodes	Nodes whose mapping as well as implementation bin has been fixed
free nodes	Nodes whose mapping and implementation bin have not been fixed
$free^h$ nodes	Free nodes mapped to hardware by GCLP at any step in the MIBS algorithm
CH_T (CS_T)	Hardware (software) implementation curve for node T
NH_T (NS_T)	Set of hardware (software) implementation bins for node T
L_T (H_T)	L (H) bin for node T. L (H) is the fastest (slowest) bin.
B_T*	Implementation bin finally selected for node T
BF_T^j	Bin fraction computed when node T is implemented in bin j
BFC_T	Bin fraction curve for node T
BS_{max}	Maximum value of bin sensitivity

Table 3. Summary of notation used in bin selection procedure.

3.6.2 Bin Fraction Curve (BFC)

Assuming node T is implemented in bin j, BF_T^j is computed as the fraction of $free^h$ nodes that have to be moved from their H bins to their L bins in order to meet feasibility. The bin fraction curve BFC_T is the plot of the bin fraction BF_T^j for each bin j of the tagged node T. The procedure to compute the *BFC* is described next. The underlying concept is similar to that used in *GC* calculation (Section 3.3.2). For simplicity, we apply the bin selection procedure only for a tagged node mapped to hardware by GCLP. A single implementation bin is assumed when the tagged node is mapped to software.

Procedure: **Compute_BFC**

Input: N_{fixed} = {fixed nodes}, $N_{free}{}^h$ = {$free^h$ nodes},
T = tagged node, with mapping M_T (assumed hardware),
hardware implementation curve CH_T

Output: $BFC_T = \{(BF_T^j, j),\ \forall j \in NH_T\}$

Initialize: $N_{H \to L} = \phi$, $t_{exec}(p)$ known for all fixed nodes p, $p \in N_{fixed}$.

for ($j = 1$; $j \le |NH_T|$, j++) {

S1. Set $t_{exec}(T) = th_T^j$

S2. For all $k \in N_{free}^h$, set $t_{exec}(k) = th_k^H$ (all $free^h$ nodes at H bins)

S3. Compute T_{finish}, given the mapping and t_{exec} for all nodes

S4. Find the set $N_{H \to L}$ of $free^h$ nodes that need to be moved to their L bins in order to meet deadline

S4.1. $N_{H \to L} \leftarrow next(N_{free}^h)$

S4.2. $t_{exec}(f) = th_f^L$, $\forall f \in N_{H \to L}$ (set to L bins)

S4.3. Update(T_{finish})

S4.4. If $T_{finish} > D$ go to S4.1

S5. $$BF_T^j = \frac{\sum_{i \in N_{H \to L}} size_i}{\sum_{i \in N_{free}^h} size_i}, \quad 0 \le BF_T^j \le 1 .$$

}

The sequence S1 to S5 outlines the procedure used to compute the bin fraction BF_T^j for a particular bin j. N_{fixed} is the set of fixed nodes and N_{free}^h is the set of free nodes that has been mapped to hardware by GCLP at the current step of the MIBS algorithm. In S1, the execution time of the tagged node is set to the execution time for the jth bin. In S2, the execution times for all the $free^h$ nodes are set corresponding to their respective H bins. The finish time for the DAG (T_{finish}) is computed in S3. In S4, we compute $N_{H \to L}$, the set of $free^h$ nodes that need to be moved to their L bins in order to meet the timing constraints. Various ranking functions can be used to order the $free^h$ nodes. One obvious choice is to rank the nodes in the order of decreasing H bin execution times th_i^H. A second possibility is to use (th_i^H / th_i^L) as the function to rank the nodes. This has the effect of moving nodes with the greatest relative gain in time when moved from H to L bin.

$BF_T{}^j$ is computed in S5 as a ratio of the sum of the sizes of the nodes in $N_{H \to L}$ to the sum of the sizes of the nodes in $N_{free}{}^h$. Recall that the *size* of a node is the number of elementary operations (add, multiply, etc.) in the node.

In summary, a high value of the bin fraction for node T indicates that selecting the jth implementation bin is likely to result in a large fraction of $free^h$ nodes being subsequently assigned to their L bins.

3.6.3 Implementation-bin Selection

Figure 3.19-a plots a typical bin fraction curve for a tagged node T. Let $L_T(H_T)$ denote the $L(H)$ bin for node T. How is the desired bin $B_T{}^*$ to be selected for this node? An intuitive choice is to set $B_T{}^* = H_T$, since this corresponds to the smallest hardware area for node T. At H_T, however, $BF_T{}^H$ is high, i.e., a large fraction of the $free^h$ nodes are in L bins so that the total hardware area might be unnecessarily large. As the tagged node shifts from bin H_T downwards, the resulting decrease in BF implies that the fraction of $free^h$ nodes at their L bin decreases, and consequently the allocated hardware area of $free^h$ nodes reduces. The slope of BFC_T represents how fast the $free^h$ node area reduces with the (leftward) bin motion of node T. This slope is called bin sensitivity BS; it reflects the correlation between bin motion of the tagged node and the overall area reduction of the $free^h$ nodes. That is, $BS_T{}^j = BF_T{}^{(j+1)} - BF_T{}^j$, $L \le j \le H - 1$, where $BS_T{}^H = 0$.

Let the maximum bin sensitivity be BS_{max} (Figure 3.19-b). The implementation bin ($B_T{}^*$) for the tagged node T is selected to be the bin with bin sensitivity equal to BS_{max}, if $BS_{max} > 0$. If BFC_T is constant (Figure 3.19-c), then $BS_{max} = 0$, and the tagged node is mapped to its H bin, since moving it from its slowest to fastest implementations does not affect the $free^h$ nodes.

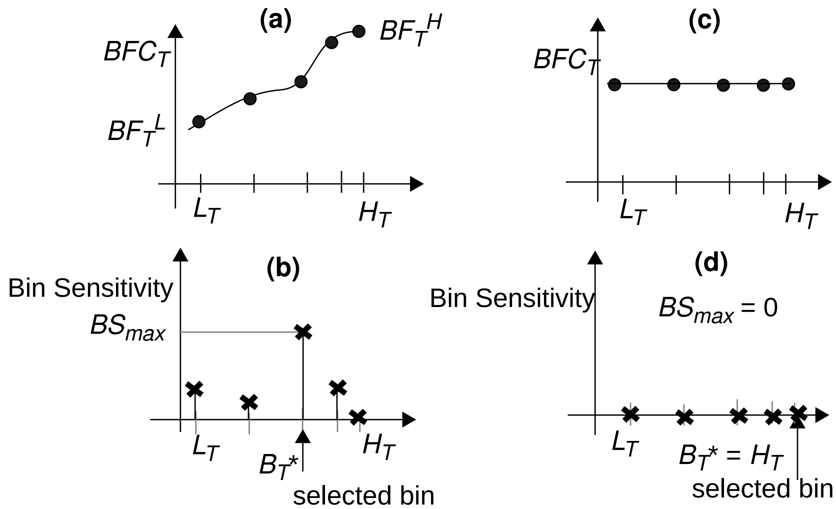

Figure 3.19. Bin fraction and bin sensitivity for implementation-bin selection

Consider the plot of bin sensitivity in Figure 3.20-a, where the regions marked S1 and S2 have identical slopes, i.e., same bin sensitivity. In this case, bin B1 which is closer to the H_T bin is preferred over bin B2 since it corresponds to a smaller area of node *T*. To incorporate this effect in general, the bin sensitivity values are weighted by the area of node *T*. In particular, the weighted bin sensitivity is plotted by multiplying the bin sensitivity at each bin *j* by ah_T^H/ah_T^j (Figure 3.20-c). B_T* is then selected to be the bin with the maximum weighted bin sensitivity. In case of a tie for the maximum weighted bin sensitivity, the bin closer to H_T bin is

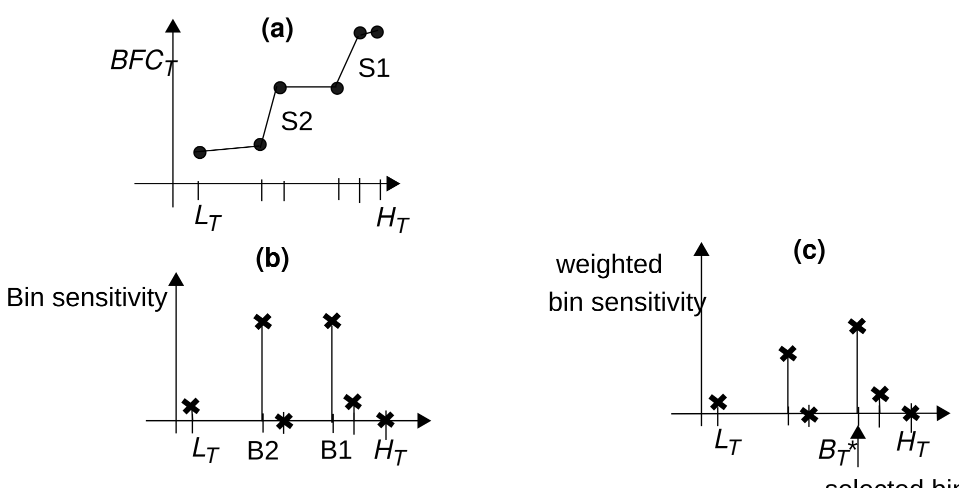

Figure 3.20. Weighted Bin Sensitivity.

selected.

In summary, the strategy for implementation-bin selection is to plot the weighted bin sensitivity and set B_T* to be the bin with the maximum bin sensitivity that is closest to the H_T bin. The bin selection procedure has complexity $O(B \cdot (|N| + |A|))$, as shown in Appendix A5. The procedure is outlined below:

Procedure	**bin_selection**
Input	N_{fixed} = {fixed nodes}, $N_{free}{}^h$ = {$free^h$ nodes} T = tagged node, with mapping M_T (assumed hardware), hardware implementation curve CH_T
Output	B_T*

S1. Compute BFC_T (Section 3.6.2)

S2. Compute bin sensitivity

S3. Compute weighted bin sensitivity

S4. Determine bin B_T* corresponding to the bin with the maximum weighted bin sensitivity

---

In the next section, we present the Mapping and Implementation Bin Selection (MIBS) algorithm to solve the extended partitioning problem *P2*.

3.7 The Extended Partitioning Problem: MIBS Algorithm

Algorithm:	**MIBS**
Input:	$\forall i \in N$: CH_i, CS_i, E_i (extremity measure), and R_i (repeller measure). Software-hardware interface communication costs: ah_{comm}, as_{comm}, and t_{comm}. Constraints: *AH*, *AS*, and *D*.
Output	$\forall i \in N$: mapping M_i ($M_i \in$ {hardware, software}), implementation bin B_i*, and start time t_i.
Initialization	N_{fixed} = {fixed nodes} = ϕ, N_{free} = {free nodes} = {N}. Compute median area and time values for all nodes in software and hardware.
Procedure	

while $\{|N_{free}| > 0\}$ {

S1. Determine M_i and t_i for all $i \in N_{free}$

S1.1. For all $i \in N_{free}$, set area and time values to their median values

S1.2. Use GCLP to compute M_i and t_i for $i \in N_{free}$. (Section 3.3)

S2. Determine the set of ready nodes N_R

S3. Select tagged node T ($T \in N_R$) using urgency measures

S4. Determine the implementation bin B_T^* for node T assuming mapping M_T

S4.1. Use the bin selection procedure to determine bin B_T^* (Section 3.6)

S5. $N_{free} = N_{free} \backslash \{T\}$; $N_{fixed} \leftarrow \{T\}$, Update t_T based on the selected implementation bin B_T^*.

}

N represents the set of nodes in the graph. N_{free} is the set of free nodes; it is initialized to N. N_{fixed} is the set of fixed nodes and is empty at start. The median values of the area and time on hardware and software mappings are computed in the initialization phase. For each step, the MIBS algorithm computes the mapping, the implementation bin, and the schedule of one node. In S1 of each step, the mapping and schedule for all the free nodes is first computed. This is done by applying GCLP over the set of free nodes assuming median area and time values. The set of ready nodes is determined in S2. This represents the set of nodes whose predecessors are fixed nodes. One of these ready nodes is selected as a tagged node in S3. In particular, we select a ready node on the critical path. In S4, the bin selection procedure is applied to determine the implementation bin for this tagged node. Finally, in S5, the schedule of the tagged node is updated depending on the implementation bin selected. The tagged node then becomes fixed. The sequence S1-S5 is repeated $|N|$ times until all the nodes in the graph become fixed.

Note that the mapping of all the nodes is not finalized at one shot in MIBS; future mappings of the remaining free nodes are allowed to change depending on the implementation bin selected for a tagged node. At any step, the known mappings and implementation bins of the fixed nodes affect the mappings of the free nodes. The complexity of the MIBS algorithm is $O(|N|^3 + B \cdot |N|^2)$, where B is the number of implementation bins per mapping (Appendix A6).

3.8 Performance of the MIBS Algorithm

The performance of the MIBS algorithm is examined in this section. As in Section 3.4, we will use both practical examples (the modem and TCS) as well as random graphs to evaluate the performance.

The procedure used to generate the hardware implementation curve for each node in the DAG is described in Section 3.8.1. In Section 3.8.2, the solutions obtained with the MIBS algorithm are compared to the optimal solutions obtained with the ILP formulation. In Section 3.8.3, we demonstrate the effectiveness of the MIBS algorithm in reducing the hardware area relative to the GCLP algorithm.

3.8.1 Estimation of the Hardware Implementation Curve

Silage code is generated for all the nodes in the DAG using the Silage code generation feature of Ptolemy. The Hyper environment is used to generate the hardware implementation curves as follows (see Figure 3.21). For each node, the critical path T_c associated with its control-dataflow graph is first computed. The hardware area required to implement the node at a sample period d equal to the critical path is estimated (Appendix A8.1 discusses the estimation techniques in detail). This sample period corresponds to the L implementation bin for the node. The sample period is then increased in multiples of the sample period until the required hardware reduces to just one resource of each type. This sample period corresponds to the H implementation bin for the node; the hardware area cannot

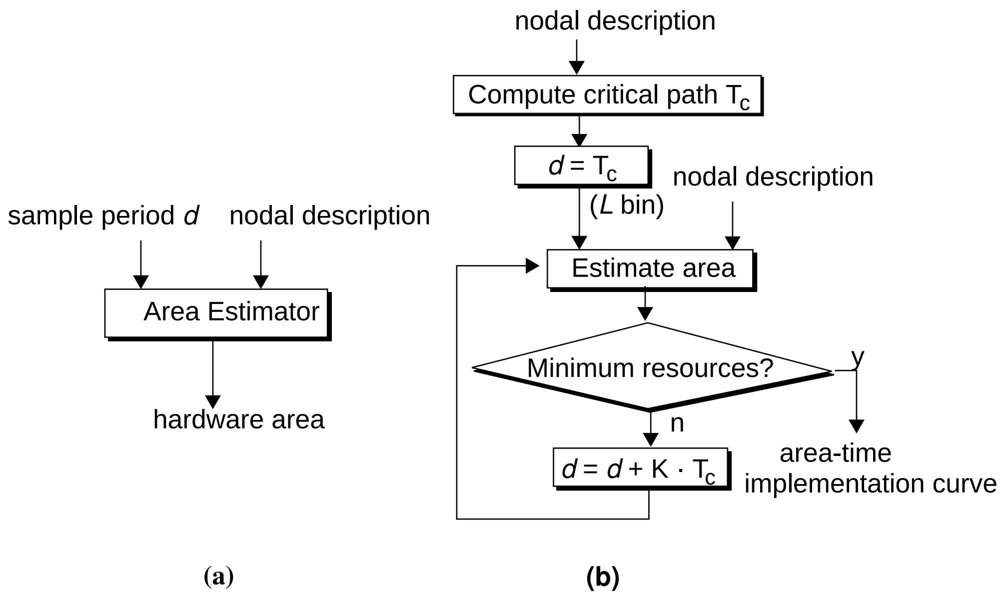

Figure 3.21. (a) Estimation of the hardware area for a node, for a given sample period (b) Computation of the hardware implementation curve for a node.

get any smaller using the particular synthesis mechanism. Sample periods between *L* and *H* bins correspond to the remaining implementation bins for the node.

The generation of the implementation curve for the random graphs is discussed in Appendix A7.3.

3.8.2 Experiment 1: MIBS vs. ILP

ILP formulations of the modem and TCS examples become impossible to solve in a reasonable time. A simplified version of the modem example with 15 nodes and 5 hardware implementation bins per node is considered here. The ILP formulation for this example requires 718 constraints and 396 variables. Table 4 summarizes the solutions obtained with ILP and with MIBS algorithm. The closeness of the solutions is encouraging, especially since ILP becomes formidable for even slightly larger problems.

Figure 3.22 plots the MIBS and ILP hardware areas for a number of random examples. For the examples tested, the MIBS solution is within 18% of the

Scenario	hardware area	solution time
ILP	158	3.5 hours
MIBS	181	3 minutes
Comparison	1.1456 times bigger	70 times faster

Table 4. Comparison of ILP and MIBS solutions.

optimal solution obtained by ILP. Larger examples could not be solved by ILP in reasonable time. In these examples, ILP failed to give even a single feasible integer solution.

3.8.3 Experiment 2: Binary Partitioning vs. Extended Partitioning

Our next objective is to evaluate the effectiveness of the extended partitioning approach in reducing the total hardware area compared with binary partitioning. Three cases are considered. In the first case, mapping is done based on GCLP, assuming that the execution times and areas for the nodes mapped to hardware are set to the values corresponding to their *L* bins. In the second case, this mapping is recomputed, now with the area and execution time values corresponding to the median implementation bins. In the third case, extended partitioning is done based on the MIBS algorithm. Table 5 shows the results for the three cases

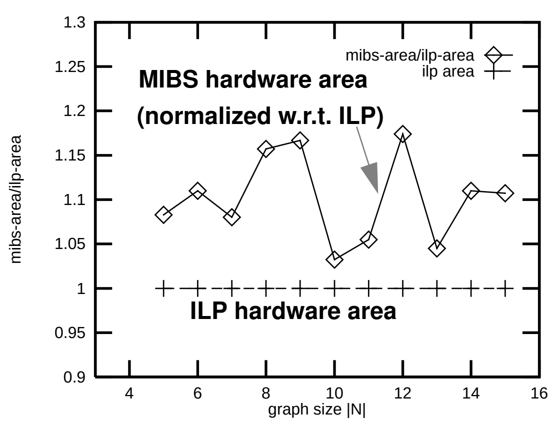

Figure 3.22. Comparison of MIBS and ILP solutions.

case	Scenario	hardware area	area reduction normalized with respect to case 1	solution time
1	GCLP, *L* implementation bin	736	1.0	0.0525s
2	GCLP, median implementation bin	530	0.7201	0.0525s
3	MIBS	362	0.4918	0.7974s

Table 5. Area improvement using MIBS vs. GCLP

applied to the modem example. The MIBS solution is observed to be much superior to both the GCLP solutions (50% less hardware compared to case 1, and 32% less than case 2). This strengthens our premise that implementation flexibility can be used at the partitioning level to reduce the overall hardware area.

In Figure 3.23, we compare, for random graphs, the hardware area obtained with MIBS to that obtained with GCLP (median area and time values). On an average, the area generated by MIBS is 26.4% smaller than that generated by GCLP.

Figure 3.24-a shows the distribution of the nodes among the implementation bins selected by the MIBS algorithm. This distribution is averaged over a number of random examples for a fixed graph size of 25 nodes. The bins are classified into 5 categories: *L* bin, *L* to median bin, median bin, median to *H* bin, and the *H* bin. It is seen that the nodes in hardware are distributed among all the implementation bins. This flexibility reduces time criticality at every mapping decision

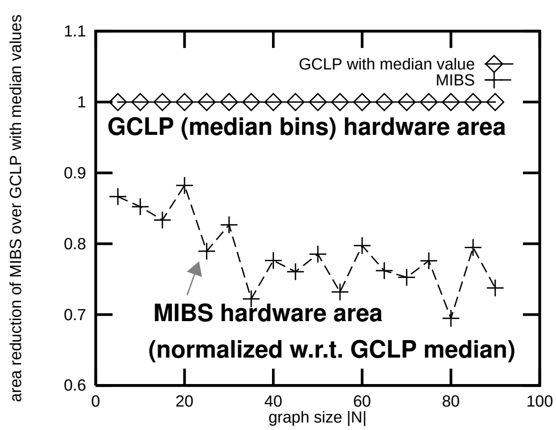

Figure 3.23. MIBS vs. GCLP (Extended Partitioning vs. Binary Partitioning)

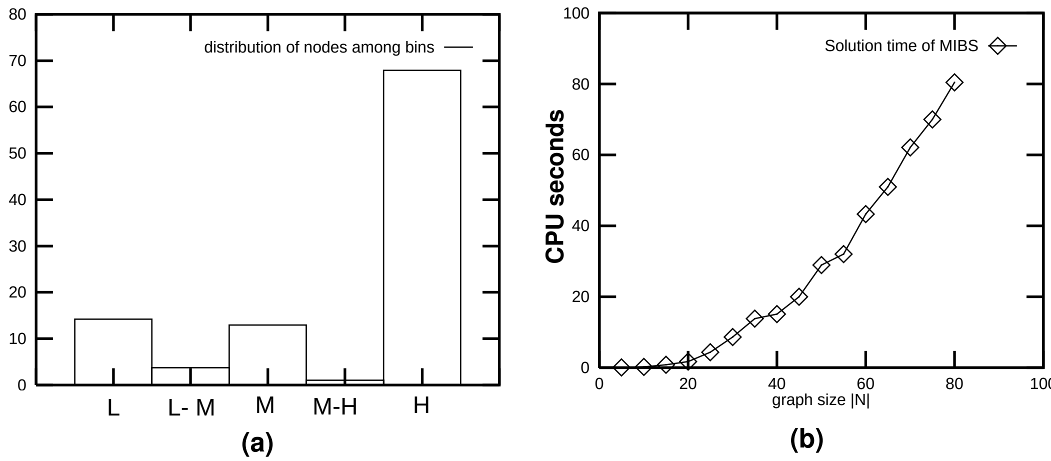

Figure 3.24. (a) Node distribution among implementation bins. (b) Solution time of MIBS algorithm averaged over random graphs per graph size.

and improves DSP utilization, i.e., the number of nodes mapped to software increases. This combined effect (reduced number of hardware nodes and their distribution over several bins) reduces the total hardware area. The MIBS solution time is plotted as a function of the graph size in Figure 3.24-b. The algorithm has complexity of $O(|N|^3 + B.\ N|^2)$.

3.8.4 Parameter Tuning

Several user-settable parameters come into play in the MIBS algorithm. These include: (1) the cut-off percentiles (α, β) used for classifying extremities in GCLP, (2) the extremity measure weight (γ) and the repeller measure weight (ν) in GCLP, (3) the ranking function for *GC* calculation (*ts*, *ts/th*, or *ah*), and (4) the ranking function for *BF* calculation (th^H, th^H/th^L, or ah^L).

Parameters α, β, γ, and ν are tuned by a simple binary search between 0 and 1. We have incorporated this automated search mechanism in our algorithm implementation. Since the MIBS algorithm is extremely fast, such an exploration is computationally viable. The *ts/th* and th^H/th^L ranking functions have been found to perform best for *GC* and *BF* calculations respectively.

3.9 Summary

At the system-level, designs are typically represented modularly, with moderate to large granularity. Each node can be implemented using a variety of algorithms and/or synthesis mechanisms in hardware or software. These implementations typically differ in area and execution time. We define *extended partitioning* as the joint problem of mapping nodes in a precedence graph to hardware or software, scheduling, and selecting a particular implementation (called implementation bin) for each node. The end-objective is to minimize the total hardware area subject to throughput and resource constraints. The extended partitioning problem is NP-hard; we proposed an efficient heuristic called MIBS to approximately solve it. The MIBS algorithm has a complexity of $O(|N|^3 + B \cdot |N|^2)$, where $|N|$ is the number of nodes, and B is the number of implementation options per node, per mapping.

In this chapter, we first presented the GCLP algorithm to solve the *binary partitioning* (mapping and scheduling) problem. It uses a global time criticality measure to adaptively select a mapping objective at each step — if time is critical, it selects a mapping that minimizes the finish time of the node, otherwise it minimizes the resource consumption. This time criticality measure overcomes the inherent drawback with serial traversal. In addition to global consideration, local optimality is sought by taking into account the preferences of nodes that consume disproportionate amounts of resources in hardware and software mappings. This effect is quantified by classifying nodes as extremities. The hardware area is further reduced by using a concept of repellers to effect on-line swaps between nodes. Repellers take into account the relative preferences of nodes, based on intrinsic algorithmic properties that dictate a preferred hardware or software mapping. The

GCLP algorithm is computationally efficient (O($|N|^2$)). For the examples tested, the GCLP solution was found to be no more than 30% larger than the optimal solution. The effectiveness of local phase nodes (extremities and repellers) in reducing the overall hardware area was experimentally verified. On an average, the use of local phase nodes reduces the hardware area by 17%, relative to solutions obtained without using local phase classification of nodes.

The philosophy of the MIBS algorithm is to extend the GCLP heuristic for extended partitioning without the associated complexity buildup. The strategy is to classify nodes in the graph as free, tagged, and fixed. Initially all nodes in the graph are free — their mappings and implementation bins are unknown. GCLP is applied over the set of free nodes. A tagged node is then selected from this set; its mapping is assumed to be that determined by GCLP. A bin selection procedure is used to compute an appropriate implementation bin for the tagged node. The procedure uses a lookahead measure, called bin fraction, which estimates for each bin of the node, the fraction of unmapped nodes that need to move to their fastest implementations so that timing constraints are met. The bin fraction is used to compute a bin sensitivity measure that correlates the implementation bin with the overall hardware area reduction. The procedure selects the bin with maximum bin sensitivity. The procedure simplifies this computation by assuming that the remaining free nodes are either in their slowest or fastest implementations. The tagged node becomes a fixed node once its implementation bin is determined. GCLP is then applied over the remaining free nodes and the sequence is repeated until all nodes in the graph become fixed. In the examples tested, the MIBS solution is found to be within 18% of the optimal solution. Experimental results also indicate that implementation bins can be used effectively to reduce the overall area by as much as 27% over solutions generated using binary partitioning.