"Proceedings of Codes/CASHE '94, Third Intl. Workshop on Hardware/Software Codesign, Grenoble, France, Sept. 22-24, 1994, pp 42-48"

A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem

Asawaree Kalavade and Edward A. Lee

Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720, USA
{kalavade,eal}@eecs.berkeley.edu

Abstract

An algorithm for the constrained hardware/software partitioning (assignment and scheduling) problem is presented. The key feature of the algorithm is the adaptive objective mechanism governed by the combination of global and local measures. As hardware area minimization and latency constraints present contradictory objectives, a global time-criticality (GC) measure selects an objective function in accordance with feasibility. In addition to global consideration, local characteristics of the nodes are emphasized by classifying nodes into local phase (LP) types. A local phase 1 node (extremity) has an obvious preference for an implementation on the basis of its area/time requirements. A local phase 2 node (repeller) is a repeller to an implementation on the basis of relative preferences of other nodes. At each iteration, the global and local criteria are superimposed by a threshold mechanism so as to determine the best implementation. The algorithm has quadratic complexity in the number of nodes and has shown promising behavior on the examples tested.

1.0: Introduction

Typical applications of embedded systems include telecommunications, multimedia systems, consumer products, robotics, and automotive control systems. Options to implement such systems include general purpose programmable processors and full-custom ASICs. Application-specific systems are commonly used whenever performance requirements (throughput) cannot be met by general-purpose solutions. However, a completely application-specific solution is often too expensive in terms of the design cost and time. It is hence increasingly common to use a mixed hardware/software implementation for such systems. Typically, custom hardware is used for the performance-intensive portions of the application, combined with a programmable processor to implement the rest. This gives the advantage of meeting performance requirements with a reduced design cost.

The design of such heterogeneous hardware/software systems is often referred to as hardware/software codesign or system-level design. The objective is usually to obtain an implementation that meets the performance requirements at a minimum cost. The finite capacities of the hardware and the software resources constitute additional constraints in the problem. In the context of hardware/software codesign, four key problems emerge [1]: partitioning, synthesis, cosimulation, and design methodology management. In this paper we will focus on the hardware/software partitioning problem. The synthesis of mixed hardware/software components and the generation of a simulation model for the mixed system have been discussed in detail elsewhere [2].

The outline of this paper is as follows: In Section 2.0 the problem is first defined. Related work is discussed in Section 3.0. In Section 4.0 the GCLP algorithm is presented. Analysis of the algorithm and experimental results are presented in Section 5.0 and Section 6.0.

2.0: The hardware/software partitioning problem

The application is described as a DAG ($G = (N,A)$), where nodes represent computations (at a *task* or *process* level of granularity) and arcs describe data and control precedences between nodes. Associated with each node i are four non-negative numbers: ah_i (area required for hardware implementation of i), as_i (code size required for software implementation of i), th_i (execution time for hardware implementation of i), and ts_i (execution time for software implementation of i). Associated with each arc (i,j) is a non-negative integer N_{ij}, indicating the number of samples of data sent from node i to node j. The underlying target architecture is assumed to consist of a programmable processor and custom hardware (motivated by the core-based ASIC technology). Costs associated with the communication of one sample of data across the hardware/software interface are specified by non-negative numbers ah_{comm} (hardware area), as_{comm} (software area), and t_{comm} (time). Design constraints include the latency (T), the capacity of the hardware resource (AH), and the capacity of the software resource (AS: available memory).

The hardware/software partitioning problem is to find a mapping (assignment I_i) of nodes to hardware and software, and the start time of each node (schedule ts_i), subject to the above constraints and taking communication costs into consideration, such that the area occupied by the nodes mapped to hardware is minimum.

3.0: Related work

The heterogeneous hardware/software partitioning problem can be formulated as an ILP for an exact optimal solution. However, this formulation becomes intractable for realistic signal processing problems with multirate operations.

Some heuristics have been reported in the area of hardware/software partitioning. Gupta et. al. [3] present a hardware-oriented approach where all nodes (except the data-dependent tasks) are initially mapped to hardware. The nodes are progressively moved to software (using a greedy approach) according to the timing constraints. The algorithm does not optimize hardware or software utilization. Henkel et. al. [4] present a software-oriented approach where all the tasks are mapped to software at start. The nodes are then moved to hardware (using Simulated Annealing) until timing constraints are met. This requires a long run time and the quality of the solution depends on the cooling schedule. Baros[5] presents a clustering-based approach to the partitioning problem. This formulation ignores precedences and scheduling information; it solves only the assignment problem.

Heterogeneous multiprocessor scheduling formulations in the literature[6][7] cannot be applied directly to the hardware/software partitioning problem as they ignore the area dimension while selecting the mapping.

Considerable attention has been directed towards the hardware partitioning problem in the high-level synthesis community. The goal in most cases is to meet the chip capacity constraints; timing constraints are not considered. Most of these approaches [8][9][10] use a clustering-based approach first presented by Camposano et al. [11].

Force Directed Scheduling, presented by Paulin et al. [12] determines the assignment time step in accordance with a global concurrency mechanism. However, it ignores area heterogeneity and hence it cannot be applied to model the heterogeneous hardware/software partitioning problem, where nodes can be mapped to multiple implementations differing in area and time properties.

4.0: The global-criticality/local-phase driven algorithm (GCLP) for constrained hardware/software partitioning

In order to minimize the hardware area, an intuitive solution is to map as many nodes as possible to software (group migration). This would result in a good utilization of the existing software resource and also reduce the hardware area. However, latency and capacity constraints could restrict this.

Another approach is to serially traverse a node list and map each node to an implementation that minimizes some objective function. Two possible objective functions could be used: minimize the *finish time* of the node (governed by the execution time on the selected implementation and the communication between the node and its predecessors) or minimize the *percentage resource consumed* by the node (hardware area/software size). However, meeting timing constraints and minimizing hardware area are contradictory goals. For example, an objective function that minimizes the finish time drives the solution towards time feasibility. Selecting the faster implementation could, however, drive it away from the optimal solution (minimum area). On the other hand, if a node is assigned to an implementation that minimizes hardware area, it is possible that feasibility will not be met. Another inherent limitation with serial traversal is its greedy approach; allocation is unlikely to be globally optimal.

The GCLP partitioning algorithm traverses a node list, and for each node *i*, it determines the mapping (I_i) and the start time (ts_i).

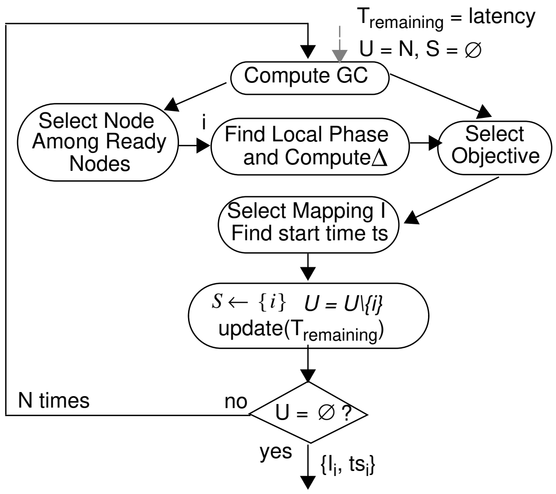

Figure 1: The GCLP Algorithm.

It overcomes the limitations of serial traversal. Instead of using a *hardwired* objective function, the GCLP algorithm *selects an appropriate objective at each step.* The objective function is selected in accordance with the following measures: 1. *Global Criticality* (GC): A global lookahead measure that gives an estimate of time criticality at each step of the algorithm. 2. *Local Phase* (LP): A measure of node heterogeneity characteristics.

The main body of the algorithm is shown in Figure 1. (The size of the graph is N. U is the set of unscheduled nodes, set to N at start. S is the set of scheduled nodes, empty at start.) In each iteration, a node is first selected from among ready nodes (nodes whose predecessors have been scheduled) based on an *urgency* criteria. The selected node is to be assigned an implementation (I_i) and scheduled in a time slot (starting time ts_i). The assignment of the node is based on two factors: global criticality and local phase. In particular, GC is a global measure of time criticality at each step of the algorithm, based on the scheduled and unscheduled nodes and the latency requirements. It directs the selection of the objective function (see Figure 2) used to determine the mapping for the node under consideration. If time is critical, GC favors an objective function that minimizes finish time, otherwise it minimizes area. GC is compared to a threshold to select an objective function. Mapping based on just GC consideration is not likely to be globally optimal because of node heterogeneity and the node-

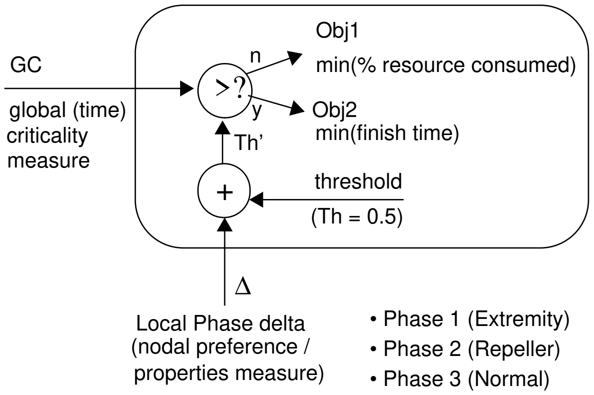

Figure 2: Objective function selection at each iteration

invariant nature of the GC at each step. The GCLP algorithm classifies nodes into three types: local phase 1(extremity), local phase 2(repeller), and local phase 3 (normal) nodes.

As nodes are at a task level of granularity, they could exhibit heterogeneity in area/time consumption on hardware and software implementations. The GCLP algorithm classifies *resource hogs* as *extremities* (local phase 1 nodes). For instance, a hardware extremity requires a large area when implemented in hardware, but could be implemented inexpensively in software. The local preference of such nodes, expressed by a *local phase delta*, modifies the threshold used in GC comparison.

Also, 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. The GCLP algorithm uses the concept of *repellers* (local phase 2 nodes) to perform *on-line swaps* between similar nodes across different implementations. This avoids a post-mapping swap. We identify certain nodal properties (called repeller properties) that are correlated to area/time gain. These repeller properties are ranked by a measure called the effective repeller value (*RV*). Given two nodes with similar software characteristics, and the choice of mapping only one of them to hardware, the node with a higher a software repeller value is selected. The repeller property value is used to modify the threshold used in GC comparison.

In summary, the global lookahead mechanism of GC overcomes the locality problems inherent to serial traversal and allocation of nodes. In addition, the local phase characteristics modify the threshold in accordance with the degree of node heterogeneity and repeller values.

The global criticality and local phase measures are next defined. This is followed by a detailed discussion of the GCLP algorithm. The motivation for using the local phase is further described at the end of this section.

4.1: Global criticality

Let us define state *k* as that when *k* nodes have been mapped. The algorithm begins at state *0* and ends at state *N* for *N* nodes in the graph. For every state *k*, GC(*k*) is the probability that any unscheduled node is implemented in hardware in order to meet overall feasibility. A high value of GC indicates that more nodes need to be mapped to hardware so as to get a feasible solution. GC is thus a measure of time criticality required to maintain time feasibility. GC is computed by the following algorithm:

Algorithm: Compute_GC
Input: Scheduled (*S*) and Unscheduled (*U*) nodes, $T_{remaining}$, ts_i and th_i the software and hardware execution times, $size_i$ size of nodes
Output: GC
Procedure:

1. Find *H* nodes to move to hardware in order to meet latency
   1.1. Estimate *H* using a priority function *Pf*
   1.2. Compute the actual finish time ($T_{finish}(H)$) based on these *H* nodes moved to hardware
   1.3. If not feasible, go to 1.1
2. Compute $GC = \dfrac{\sum_H size_i}{\sum_U size_i}$ and update $T_{remaining}$

In Step 1.1, the *H* nodes to be moved to hardware are first estimated such that the remaining *U*\ *H* nodes can be executed on the software resource within $T_{remaining}$. The nodes to be moved to hardware are selected from an ordered list ranked 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 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 first. We are currently experimenting with different *Pf*s to study the effect of GC on the performance of the algorithm.

Note that the *H* nodes obtained in Step 1.1 is an estimate (lower bound) for the nodes to be moved to hardware to meet latency constraints, since precedences are ignored in this calculation. Step 1.2 determines if these *H* nodes meet feasibility by actually computing the finish time (O(*A*) algorithm). If the result is infeasible, additional nodes are moved by repeating step 1.1. GC is computed in Step 2. The size of a node is the number of elementary operations (atomic at the level of the resource: add, multiply, etc.) in the node.

GC is thus the proportion of unscheduled nodes that need to be mapped to hardware based on the state of the system and latency requirements. Simplistically, GC(k) can be thought of as the node-invariant *probability* that any unscheduled node, in state *k*, is mapped to hardware. It may change at each iteration of the algorithm. A high GC indicates a global time criticality.

4.2: Local phase

All nodes in a graph are classified into three disjoint sets: phase1 (extremity), phase 2 (repeller), and phase 3 (normal node).

4.2.1: Extremity nodes (Phase 1)

Define *I* and $\bar{I}$ to be complementary implementations. A node is an *extremity* to an implementation *I*, if it consumes a large amount of the precious resource on *I* (*time* for the software resource, and *area* for the hardware resource) and a relatively small amount of the precious resource on $\bar{I}$ — the rationale in moving this node to $\bar{I}$ is obvious. We define such a node to be a *local phase 1* node. For example, a hardware extremity node is a node that consumes large area hardware but is not too intensive in software. Figure 3 describes a mechanism to identify the software and

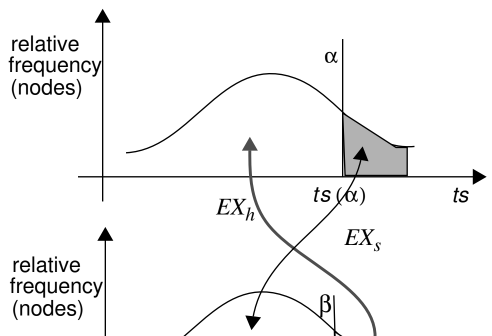

Figure 3: Hardware (EX_h) and software (EX_s) extremities.

hardware extremity sets EX_s and EX_h respectively.

Algorithm: Compute_Extremity_Sets
Input: ts_i, ah_i the software execution time and hardware area for each node i
Output: EX_s and EX_h
Procedure:

1. Compute the histograms for the software execution times and hardware areas, given the ts_i and ah_i values respectively.
2. Choose top percentiles α and β that determine the lower cut-offs $ts(\alpha)$ and $ah(\beta)$ for the ts and ah ranking of nodes respectively.
3. If $(ts_i \geq ts(\alpha)$ and $ah_i < ah(\beta))$, $i \in EX_s$
   If $(ah_i \geq ah(\beta)$ and $ts_i < ts(\alpha))$, $i \in EX_h$
4. Define extremity measure x:
   If $i \in EX_s$, $x = \dfrac{ts_i / ts_{max}}{ah_i / ah_{max}}$, else $x = \dfrac{ah_i / ah_{max}}{ts_i / ts_{max}}$, $0 \leq x \leq 1$
   where $ts_{max} = max_i\{ts_i\}$ and $ah_{max} = max_i\{ah_i\}$.
5. Order the extremity set members by x.

4.2.2: Repellers (Phase 2)

Nodes in a graph are at a *task* level of granularity; i.e. an instruction-level subgraph is associated with each node (Figure 4). Several repeller properties can be identified for each subgraph. For instance, bit-level instruction mix and precision level are software repeller properties; memory-intensive instruction mix and table-lookup instruction mix are hardware repeller properties.

Let us consider the bit-level instruction mix repeller property in some detail. Bit-level Instruction Mix (*BLIM*) is a software repeller property; higher the BLIM, higher the repeller value. It is defined as the fractional contribution of bit-level instructions to the total instructions in a node ($BLIM_i$: $0 \leq BLIM_i \leq 1$). Consider two non-extremity nodes n_1 and n_2, with software and hardware areas as_1, as_2, ah_1, ah_2 respectively. Suppose $BLIM_1 > BLIM_2$. Now, if $as_1 \sim as_2$, then $ah_1 < ah_2$ (because bit-level operations can be 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.

Other software(S) and hardware(H) repeller properties have been similarly quantified. The effective repeller value of a node is computed as a convex combination of the values of the various repeller properties. The algorithm outlined below describes the computation of the effective repeller value (RV_i) for each node.

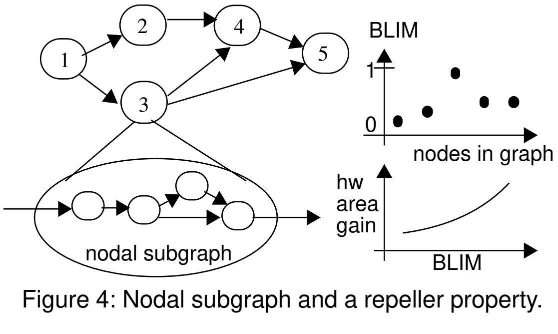

Figure 4: Nodal subgraph and a repeller property.

Algorithm: Compute_EffectiveRepellerValue
Input: For node i in N, for repeller property p in P ($P=S \cup H$), the value of the property $v_{i,p}$.
Output: For each i, the effective repeller value RV_i

1. Compute for each property p:
   $\sigma(v_{i,p})$ = variance, over i, of $v_{i,p}$
   $min(v_{i,p})$ = minimum, over i, of $v_{i,p}$
   $max(v_{i,p})$ = maximum, over i, of $v_{i,p}$
   $a_p = \dfrac{\sigma(v_{i,p})}{\sum_{p \in P} \sigma(v_{i,p})}$ = weight of repeller property p
2. Compute the normalized property value ($nv_{i,p}$) for property p, for each node i
   $nv_{i,p} = \dfrac{v_{i,p} - min(v_{i,p})}{max(v_{i,p}) - min(v_{i,p})}$, where $0 \leq nv_{i,p} \leq 1$.
3. Compute the effective repeller value (RV_i) for each i
   $RV_i = \frac{1}{2} \cdot \left(\sum_{p \in H} a_p \cdot nv_{i,p} - \sum_{p \in S} a_p \cdot nv_{i,p} \right)$, $-0.5 \leq RV_i < 0.5$.

The weight a_p of a property is proportional to its property value variance, since repeller values are computed relative to other nodes. In Step 3 above, we currently assume that the hardware area gain is linearly correlated to the software repeller value. The exact correlation is being investigated and would modify the weights of the individual properties, a_p.

In summary, a node is a *repeller* to an implementation if it is not an extremity, and if it possesses a non-zero effective repeller value. Repellers are classified as local phase 2 nodes. The *effective repeller value* of a node is a measure of the resource utilization gain through swapping of two similar phase 2 nodes between complementary implementations.

A node that is not an extremity or a repeller is defined to be a *phase 3* node *(normal node)*.

The GCLP algorithm is presented next. The motivation for using extremities and repellers is discussed in Section 4.4.

4.3: The GCLP algorithm

Algorithm: GCLP
Input: $G = (N,A)$; $\forall i \in N$: ah_i (hardware area), as_i (software code size), th_i (execution time in hardware), ts_i (execution time in software), and RV_i, (effective repeller value)
$\forall (i,j) \in A$: Nij: number of data samples sent from i to j, ah_{comm} (hardware area), as_{comm} (software size), and t_{comm} (execution time) per data sample transferred between hardware and software.
AH, (hardware capacity), AS, (software capacity), and T (latency) constraints.
Extremity sets EX_s and EX_h,
Output: $\forall i \in N$, implementation I_i (0:hardware, 1:software) and start time ts_i.
Initialize: U = {unscheduled nodes} = N, S = {scheduled nodes} = ϕ.
Procedure:
while {$|U|$ >0}
{
1. Compute GC using Compute_GC (Section 4.1)
2. Determine the set of *ready* nodes R
3. Compute the effective execution time $t_{exec}(i)$ for each node i
   If $i \in U$ $t_{exec}(i) = GC.th_i + (1-GC).ts_i$
   else if $i \in S$ $t_{exec}(i) = t_{Ii}$
4. Compute the longest path (*longestPath(i)*), $\forall i \in R$ using $t_{exec}(i)$

5. Select node i, $i \in R$, for assignment: max(*longestPath(i)*)

6. Select implementation I_i for i:

6.1. if $i \in \left(EX_s \cup EX_h\right)$ $\quad \Delta = sign \cdot \frac{x - x_l}{x_h - x_l} \cdot 0.5$ (phase 1)

else if($RV_i \neq 0$) $\quad \Delta = RV_i$ (phase 2)

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

where

$sign$ = -1 if $i \in EX_s$, else $sign$ = 1

x = extremity measure for i (Section 4.2.1)

x_l (x_h) is the minimum(maximum) x over all i

RV_i = effective repeller value for i (Section 4.4.2)

$-0.5 \leq \Delta \leq 0.5$

6.2. Threshold $= 0.5 + \Delta$, $0 \leq$ Threshold ≤ 1

6.3. If ($GC \geq$ Threshold) p: min(Obj2);
else p: min(Obj1);

6.4. I_i = p; Set(ts_i); U = U\{i}; $S \leftarrow \{i\}$,
Update($T_{remaining}$, $AH_{remaining}$, $AS_{remaining}$);

}

Obj1: $\left(\frac{as_i^c}{AS}, \frac{ah_i^c}{AH_{remaining}}\right)$

Obj2: $t_{fin}(i, I)$, where $I \in (0, 1)$

$t_{fin}(i,I) = max((t_{fin}(b_i,I_b) + t_c(b_i,i)), t_{fin}(i_{last},I)) + t(i,I)$

where

b_i = predecessor of i,

I_b = mapping of b_i,

$t_{fin}(b_i,I_b)$ = finish time of b_i on I_b,

$t_c(b_i,i)$ = communication time between b_i and i,

i_{last} = last node mapped to implementation I,

$t_{fin}(i_{last},I)$ = finish time of i_{last} on I

$t(i,I)$ = execution time of i on implementation I.

Comments:

Obj1 uses a "percentage resource consumption" measure. Note that for hardware, the area required by a node is computed as a fraction of the *remaining* hardware area ($AH_{remaining}$). Obj1 thus favors software allocation as the algorithm proceeds. The resource area required (as_i^c, ah_i^c) incorporates communication components (ah_{comm}, as_{comm}, N_{ij}) between the node i and its predecessor.

Obj2 selects an implementation that minimizes the finish time of a node; its formulation is self-explanatory.

4.4: Threshold modification in the GCLP algorithm

In this section we describe the rationale for modifying the threshold using local phase nodes.

4.4.1: Use of extremities (Phase 1 Nodes)

Let us assume that GC is computed using ts_{max} ranking for moving nodes from software to hardware. Suppose the extremity effect on threshold is ignored. At any state k, GC(k) can be misleading (for assignment) on two counts because of its node invariant interpretation over *all unscheduled* nodes.

1. Its direct use could lead to an obvious (irreparable) local suboptimality. At state k, let the ready node i be a hardware extremity. It is likely that in the GC(k) computation, this node i is retained in software due to the ts_{max} rank-based swapping. If the resultant $GC(k) \geq 0.5$, i could get mapped to hardware based on time-criticality. However, i is a hardware extremity and mapping it to hardware is undesirable from the area minimization objective. Hence, to avoid suboptimality, a mechanism to move *up* the threshold by atleast |GC(k) - 0.5| is needed.
2. It could direct towards infeasibility. At state k, let the ready node i being mapped be a software extremity. It is likely that in GC(k) computation, this node i is moved to hardware. If the resulting $GC(k) < 0.5$, this node could be retained in software, thus changing the mapping assumed in GC(k) computation. However, i is a software extremity, and mapping it to software could cause infeasibility based on execution time. Hence, to avoid infeasibility, a mechanism to move *down* the threshold by atleast |0.5 - GC(k)| is needed.

As discussed above, the threshold needs to be modified (Threshold $= 0.5 + \Delta$) to reflect the effect of extremities. A simple approach is to assume a constant slope for Δ over the minimum to maximum range of the extremity measure (Section 4.2.1).

4.4.2: Use of repellers (Phase 2 Nodes)

Once a feasible solution is obtained, the overall hardware area can be further reduced by swapping nodes between hardware and software. Repellers constitute a *virtual on-line swap* mechanism to reduce the overall hardware area — a post-mapping swap is avoided. Recall that the repeller value is a measure of the swapping gain of two similar phase 2 nodes. Given a choice of mapping just one of two nodes to hardware, the node with a higher software repeller value is chosen. Given a ready phase 2 node i with a sufficiently high repeller value in implementation I, the algorithm tries to achieve a relative shift of i out of I. Specifically, it modifies the threshold such that the objective selected will favor the complementary implementation $\bar{I}$. This swap frees up I for an as-yet unscheduled node with a lower repeller property, thus reducing the overall allocated hardware area. The swap is termed v*irtual* because it does not involve two concurrent nodes as in a traditional swap.

The threshold is modified to reflect the repeller value: Threshold $= 0.5 + \Delta$, where Δ = RV (Section 4.2.2).

5.0: Experimental results

We present two sets of experimental results. The first experiment is a comparison of the solutions obtained from the GCLP algorithm and the optimal ILP formulation. In the second experiment, the GCLP algorithm is applied to three scenarios inorder to study the effectiveness of the local phase nodes.

5.1: Examples

The first example is a 32KHz 2-PSK modem. The nodes are at a *task* level of granularity (individual nodes include carrier recovery, timing recovery, equalizer, scrambler, etc.).

The second example is a 8KHz bidirectional telephone channel simulator (TCS). The task-level nodes include a linear distortion filter, a Gaussian noise generator etc.

A multirate version of the first example (85 nodes) has also been tested.

5.2: Estimation of area/time properties

The Motorola DSP 56000 was used as the target software processor, and hardware was implemented using custom hardware synthesized by Hyper[13]. Memory-mapped I/O was used for the hardware/software communication. The application was developed in the SDF domain of Ptolemy[14]. An acyclic precedence graph was generated for the application automatically. The graph was then retargeted to the 56000 assembly code generation domain of Ptolemy (CG56) and assembly code for the Motorola 56000 was generated[15]. Estimates of the software area (as_i) and execution time on software (ts_i) were obtained from this. The graph was also retargeted to the Silage domain and Silage code was generated[2]. Area and time estimates for hardware implementation were obtained by running the generated silage code through Hyper. The hardware execution time (th_i) was computed as the best-case execution time (critical path). Hardware area estimates (ah_i) corresponding to this execution time were computed.

5.3: Experiment 1

The examples are partitioned using the GCLP algorithm. The results (Table 1) are compared to those obtained by solving the corresponding ILP formulation. The quantities *HA*, *SA*, *Util.*, in Table 1 correspond to the total hardware area (as a fraction of the capacity), the total software size (as a fraction of the memory capacity), and the DSP utilization respectively, in the resulting solution. A *good* solution corresponds to a low hardware area and a high DSP utilization.

example	size (nodes)	ILP HA	ILP SA	ILP Util.	GCLP HA	GCLP SA	GCLP Util.
modem	27	62%	98%	93.8%	64%	84%	84.8%
TCS	15	89%	53%	73.5%	89%	53%	73.5%

Table 1: Results from ILP and GCLP algorithm.

1. The solutions compare favorably. For the first example, the GCLP results are reasonably close to the optimal solution (all but one node had identical mappings). The GCLP solution for the second example is identical to the ILP solution.
2. The ILP ran for several hours, whereas the GCLP algorithm finished in the order of seconds.

5.4: Experiment 2

The GCLP algorithm is run on example 1 under three scenarios. In the first case, the local phase node classification is not considered ($\Delta = 0$). In the second case, the effect of using only the extremities is considered (no repellers). In the third case, Δ for both extremities and repellers is considered. Table 2 summarizes the results.

Algorithm Scenario	hardware area	software area	DSP utilization
GC only, no local phase delta	64%	38.8%	88.5%
GC, and delta for extremities only (no repellers)	61%	48.6%	85%
GC, with delta for extremities and repellers	51.66%	68.5%	86%

Table 2: Comparison of results from different scenarios of GCLP algorithm for example 1.

1. Local phase nodes significantly improve the solution (64% hardware area required without local phase nodes vs. 51.66% hardware area required when local phase nodes are considered).
2. Repellers reduce the hardware area through their virtual-swap mechanism (61% hardware area required when repeller nodes are not considered vs. 51.66% hardware area required when repeller effects are considered).
3. The extremities are seen to match their expected implementations (ex: *Pulse Shaper,* a hardware extremity node, is mapped to software. *Carrier Recovery,* a software extremity node, is mapped to hardware).
4. Repeller nodes are also mapped to their desired implementations (ex: *Scrambler*, a high BLIM software repeller, is mapped to hardware).
5. Also, ranking using ts_{max} and $(ts - th)$ gives better GC estimates (better solution) than the ah_{min} ranking.

6.0: Algorithm analysis

1. The GCLP algorithm has a worst-case complexity of O(*NA*), where *N* is number of nodes and *A* is number of arcs. For graphs representing signal processing systems, $O(A) \approx O(N)$: the algorithm has quadratic complexity.
2. The algorithm detects violation of the timing constraint during GC calculation. A violation of the timing constraint implies one of the following: GC pointed to an implementation, but the node could not get its preferred implementation because the resource was exhausted, or GC pointed to an implementation, but the local phase effect swamped the GC. In either case, a simple swap algorithm (O(*N*) per swap) could be used to maintain feasibility.
3. The algorithm generates a solution that will never violate capacity constraints by construction (an implementation is selected only if a resource exists to implement it).
4. Algorithm boundedness is not proved yet. We are working on the formulation of a *goodness measure* to evaluate the generated solution. Experimental results show that the generated solution is close to optimal.

7.0: Conclusions

The GCLP algorithm for hardware/software partitioning has been presented. The algorithm maps the selected node to an implementation (hardware or software) in each iteration. The key features of the algorithm are summarized:

1. Hardware area minimization and meeting latency constraints present contradictory objectives. The GCLP algorithm uses a global time-criticality measure (defined at each step by the state of scheduled and unscheduled nodes) to determine the objective function at each step.
2. In addition to global consideration, local characteristics of the nodes are emphasized by classifying nodes as extremities, repellers, or normal nodes. The global criticality (GC) maintains global feasibility, while the local phase (LP) accounts for local optimality (extremities) and relative preferences (repellers).
3. At each step, the global and local criteria are superimposed by a threshold mechanism so as to determine the best objective. The combination of GC and LP gives a feasible solution that is close to the optimal.
4. The GCLP algorithm is inherently robust due to the negative feedback that stabilizes the value of GC.

The algorithm shows promising behavior. For the selected set of examples, the results compared favorably with the optimal solution.

Future work includes further tuning of the algorithm parameters and extending the algorithm to partition among multiple hardware implementations (I>2).

8.0: Acknowledgements

This research was supported by a grant from the Semiconductor Research Corporation (SRC 94-DC-008). Pratyush Moghe and Prof. Jan Rabaey are gratefully acknowledged for helpful discussions.

9.0: References

[1] Asawaree Kalavade, Edward A. Lee, “Manifestations of Heterogeneity in Hardware/Software Codesign”, *Proceedings of the 31st Design Automation Conference*, San Diego, CA, June 1994, pp 437-438.

[2] Asawaree Kalavade, Edward A. Lee, “A Hardware/Software Codesign Methodology for DSP applications”, *IEEE Design and Test of Computers*, Sept. 1993, pp 16-28.

[3] R. Gupta, Giovanni DeMicheli, “System-level Synthesis Using Re-programmable Components”, *Proceedings of the European Conference on Design Automation*, Brussels, Belgium, Feb.1992, pp 2-7.

[4] Ernst R., Henkel J., “Hardware/software Codesign of Embedded Controllers based on Hardware Extraction”, *Handouts of the 1st Intl. Workshop on Hardware/Software Codesign*, Estes Park, Colorado, Sept. 1992.

[5] Edna Baros, Wolfgang Rosential, “A Method for Hardware/Software Partitioning”, *Proceedings of COMPEURO’92, IEEE Intl. Conference on Computer and Software Engineering,* May 4-8, 1992, The Hague, The Netherlands, pp 580-585.

[6] G. Sih, E. A. Lee, “A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 2, Feb. 1993, pp 175-187.

[7] Hamada, T. Banerjee, S. Chau, P.M. Fellman, R.D, “Macropipelining based heterogeneous multiprocessor scheduling”, *Proceedings of ICASSP-92*: 1992 IEEE International Conference on Acoustics, Speech and Signal Processing, San Francisco, CA, USA, 23-26 March 1992, New York, NY, USA: IEEE, 1992, p. 597-600 vol. 5.

[8] E. D. Lagnese, D. E. Thomas, “Architectural Partitioning for System-level Synthesis of ICs”, *IEEE Transactions on Computer Aided Design*, Vol. 10, no. 7, July 1991, pp 847-860.

[9] F. Vahid, D. Gajski, “Specification Partitioning for System Design”, *Proceedings of the 29th Design Automation Conference,* June 1992, Anaheim, CA, pp 219-224.

[10] M. C. McFarland, T. J. Kowalski, “Incorporating Bottom-up Design into Hardware Synthesis”, *IEEE Transactions on Computer Aided Design,* Vol. 9, no. 9, Sept. 1990, pp 938-950.

[11] R. Camposano, R. K. Brayton, “Partitioning before Logic Synthesis”, *Proc. of the Intl. Conference on Computer Aided Design (ICCAD)*, 1987, pp 324-326.

[12] Pierre G. Paulin, John P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASICs”, *IEEE Transactions on CAD*, Vol. 8, no. 6, June 89, pp 661-679.

[13] J. M. Rabaey et. al. “Fast Prototyping of datapath-intensive Architectures”, *IEEE Design and Test of Computers*, pp. 40-51, June 1991.

[14] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, “Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems”, *International Journal of Computer Simulation*, special issue on “Simulation Software Development,” January, 1994.

[15] J. Pino, S. Ha, E. Lee, J. Buck, “Software Synthesis for DSP Using Ptolemy”, invited paper in the *Journal on VLSI Signal Processing*, special issue on “Synthesis for DSP”, to appear: 1994.