Very fast Simulated Annealing for HW-SW partitioning

Sudarshan Banerjee Nikil Dutt
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

banerjee@ics.uci.edu dutt@ics.uci.edu

CECS Technical Report #04-#18

June, 2004

Abstract

Hardware/software (HW-SW) partitioning is a key problem in the codesign of embedded systems and has been studied extensively in the past. With the wide availability of commercial platforms such as the Virtex-II Pro series from Xilinx that integrate processors with reconfigurable logic, one major existing challenge is the lack of efficient algorithms that can generate very high-quality solutions by exploring a huge HW/SW exploration space- the key criterion is to obtain such solutions at a speed suitable for integration into a compiler-based partitioner. In this report, we make two contributions for HW-SW partitioning of applications specified as procedural call-graphs:
1) We prove that during partitioning, the execution time metric for moving a vertex needs to be updated only for the immediate neighbours of the vertex, rather than for all ancestors along paths to the root vertex. This enables move-based partitioning algorithms such as Simulated Annealing (SA) to execute significantly faster, allowing call graphs with thousands of vertices to be processed in less than half a second
2) Additionally, we devise a new cost function for SA that enables searching of spaces overlooked by traditional SA cost functions for HW-SW partitioning, allowing the discovery of additional partitioning solutions in a very efficient manner.
We present experimental evidence on a very large design space with over 12000 problem instances. We generate the problem instances by varying the call-graph sizes from 20 to 1000 vertices, indegree/outdegree of vertices, communication-to-computation ratios, and varying the area constraint on the hardware partition. Thousands of problem instances are explored in a matter of minutes as compared to several hours or days using a traditional SA formulation. Aggregate data collected

over this large set of experiments demonstrates that when compared to a KLFM algorithm starting with all vertices in software, our approach is 1) asymptotically faster, with a run-time around 5 times faster for graphs with 1000 vertices, 2) is frequently able to locate better design points with over 10 % improvement in application execution time, and 3) the average improvement in application execution time is around 5%. We confirmed the solution quality of results generated by our approach by additional comparisons with a) set of KLFM runs starting from different initial configurations for the same problem instance, and b) other cost-functions commonly used in SA-based approaches for HW-SW partitioning. Overall, our approach generates superior results and executes much faster.

Contents

List of Figures

1 Introduction

Partitioning is an important problem in all aspects of design. HW-SW (hardware-software) partitioning, i.e the decision to partition an application onto hardware (HW) and software (SW) execution units, is possibly the most critical decision in HW-SW codesign. The effectiveness of a HW-SW design in terms of system execution time, area, power consumption, etc, are primarily influenced by partitioning decisions.

The partitioning problem has a lot of related variations depending on the objective function being optimized. In this report, we consider the problem of minimizing execution time of an application for a system with hard area constraints. An example of a related problem is the problem of minimizing aggregate energy consumption for a system with hard constraints on execution time.

We consider an application specified as a DAG (directed acyclic graph), extracted in the form of a callgraph from a sequential application written in ’C’. Our target system architecture has one microprocessor (SW) and one area-constrained hardware unit (HW) such as a reconfigurable logic fabric. An example of such an architecture is the Xilinx Virtex-II Pro Platform FPGA XC2VPX20 that integrates one PPC405 (PowerPC) processor with reconfigurable logic. In this work, we assume that the HW and SW units execute in mutual exclusion- this allows us to focus exclusively on the partitioning problem.

For a DAG representing a call graph, the execution time of a vertex needs to be computed from all descendants in the sub graph rooted at the vertex. In HW-SW partitioning, when a vertex is moved from SW to HW or vice-versa, the execution time of the program changes due to the different execution times of a HW versus a SW implementation, along with increased or decreased communication cost across the cut. This change in execution time is represented by the *execution time change* metric.

In this report, we make two contributions to HW-SW partitioning. First we prove that for a callgraph representation, when a vertex is moved to a different partition, it is only necessary to update the *execution time change* metric [8] for its immediate parents and immediate children instead of all ancestors along the path to the root. This in general allows for a more efficient application of move-based algorithms like simulated annealing (SA).

Secondly, we present a cost function for simulated annealing to search regions of the solution space often not thoroughly explored by traditional cost functions. This enables us to frequently generate more efficient design points.

Our two contributions result in a very fast simulated annealing (SA) implementation that generates partitionings such that the execution times are frequently better by over 10% compared to a KLFM (Kernighan-Lin/Fiduccia-Matheyes) algorithm for HW-SW partitioning for graphs ranging from 20 vertices to 1000 vertices. Equally importantly, graphs with a thousand vertices are processed in much less than a second, with the algorithm run-time asymptotically faster than a KLFM implementation by around 5 times for graphs with a 1000 vertices.

Given the known propensity of a KLFM approach to get stuck at local minima, we additionally conducted experiments where the KLFM algorithm was allowed to start from different configurations. Our approach generates better quality results compared to this set of independent KLFM runs on the same problem space. Additional comparisons with commonly used SA cost functions in HW-SW partitioning establish the quality of results generated by our approach.

The rest of this report is organized as follows: in Section 2, we review related research in HW-SW partitioning. In Section 3 we review the problem description. In Section 4, we prove that the *execution time change* metric needs to be updated only for immediate neighbours of a vertex. In Section 5, we discuss the simulated annealing algorithm and present our proposed approach towards a more interesting cost function. In Section 6, we present the experiments conducted. We conclude with Section 7.

2 Related work

Hardware-software partitioning is an extensively studied ”hard” problem with a plethora of approaches- dynamic programming [13], genetic algorithms [4], greedy heuristics [12], to name a few. Most of the initial work on HW-SW partitioning, [13], [14] focussed on the problem of meeting timing constraints with a secondary goal of minimizing the amount of hardware. Subsequently there has been a significant amount of work on optimizing performance under area constraints, [1], [3], [8]. With the goal of searching a larger design space, techniques such as simulated annealing (SA) have been applied to HW-SW partitioning using fairly simple cost functions. While a lot of initial work such as [14] was based exclusively on simulated annealing, recent approaches commonly measure their quality against a SA implementation. For example, [1] compares SA with a knowledge-based approach, and [3] compares SA with tabu search.

It is well-known that SA requires careful attention in formulating a cost function that allows the search to ”hill-climb” over suboptimal solutions. However, much of the published work in HW-SW partitioning have not studied in detail the SA cost functions that permit a wider exploration of the search space. As an example, in [3], [9], the SA formulation considers only valid solutions satisfying constraints, thus restricting the ability of SA to ”hill-climb” over invalid solutions to reach a valid better solution.

The two previous pieces of work in HW-SW partitioning that are most directly related to our work are [8], [5]. Our model for HW-SW partitioning is based on [8], a well-known adaptation of the KL paradigm for HW-SW partitioning; our efforts in improving the quality of the cost function are closely related to [5].

Our partitioning granularity is similar to [8], effectively that of a loop-procedure call-graph; each partitioning object represents a function and the DAG edges are annotated with callcounts. [8] introduced the notion of execution time change metric for a DAG, and updating the metric potentially by evaluation of ancestors along the path to the root. The linear cost function in [8] ignores the effect of HW area as long as the area constraint is satisfied.

[5] provides an in-depth discussion of cost functions and the notion of improving the results obtained from a simple linear cost function by dynamically changing the weights of the variables. We differ from [5] in the following ways: [5] addresses the problem of choosing a suitable granularity for HW-SW partitioning that minimizes area while meeting timing constraints; since we consider the problem of minimizing execution time while satisfying HW area constraints, the proposed cost function in [5] needs significant adaptation for our problem. In [5], the dynamic weighting technique was applied towards the secondary objective of minimizing HW area once the primary

objective, the timing constraint, was almost satisfied. We however, apply a dynamic weighting factor to our cost functions in various regions of the search space to better guide the search. Last but not the least, since their primary focus was on the granularity selection problem, there was no quantitative comparison of their approach with other algorithms- we have compared our approach to the KLFM approach with an extensive set of test cases and demonstrated the effectiveness of our approach.

3 Problem description

3.1 Problem description

The application specification methodology and architectural assumptions in our problem definition are similar to [8]. Here we provide a brief summary of the key aspects of the problem definition.

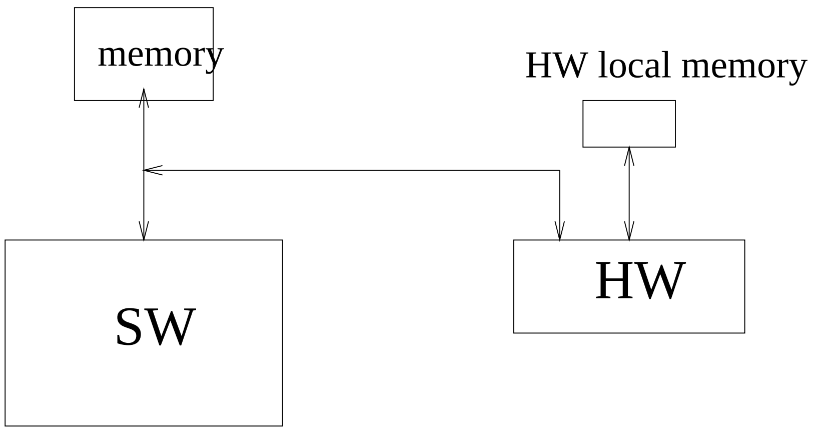

Figure 1. Target architecture

We consider an application specified as a DAG (directed acyclic graph), extracted from a sequential program written in C, or, any other procedural language. The target architecture for this application is a system with a single SW processor and a single HW unit connected by a system bus, as shown in Figure 1. An example of such an architecture is the widely available Xilinx XC2VPX20 with a single PPC405 processor connected to reconfigurable logic by a PLB (processor local bus). In this work, we assume mutually exclusive operation of the two units, i.e the two units may not be computing concurrently. We additionally assume that the HW unit can be configured only once before the application starts execution and the HW functionality does not change once the application starts execution, i.e., we consider that the HW unit does not have dynamic RTR (run-time reconfiguration) capability [1]. The problem considered in this report is to partition the application such that the execution time of the application is minimized while simultaneously satisfying the hard area constraints of the HW unit.

[1]This is a relevant practical assumption in light of the significant reconfiguration penalties incurred in such commonly available single-context RTR architectures

Each partitioning object corresponding to a vertex in the DAG is essentially a function that can be mapped to HW or SW. Each directed edge (x, y) in the DAG represents a call or an access made by the caller function x to the callee function y. The SW execution times and callcounts are obtained from profiling the application on the SW processor. In this model, the HW execution time and the HW area for the functions are estimated from synthesis of the functions on the given HW unit [2]. Communication time estimates are made by simply dividing the volume of data transferred by the bus speed. Since the execution time model is sequential, bus contention is assumed to play an insignificant role.

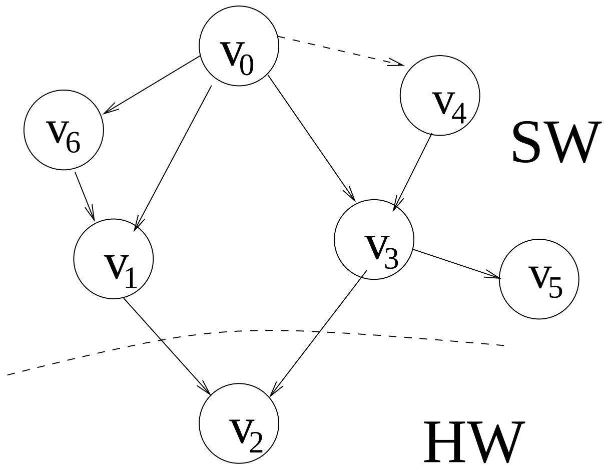

Figure 2. Simple example

We next motivate the first part of our contribution with the simple example shown in Figure 2. For the callgraph in the figure, the execution time of the program (same as the execution time for vertex v_0) obviously depends on the execution time of its descendant v_2. Let us assume all vertices were initially in SW. If we move the vertex v_2 to HW, the execution time changes due to HW-SW communication on the edges (v_3, v_2), (v_1, v_2) and change in execution time for vertex v_2. It would appear that any execution time related metric for the vertices v_0, v_6, v_4, would need to be updated when this move is made. In the next section, we show with a simple example that this is not true for the *execution time change metric* and follow up with a proof.

Before proceeding further, we need to introduce a slightly more formal set of notations required in the rest of this report.

3.2 Notational details

The input to the partitioning algorithm is a directed acyclic graph (DAG) representing a call-graph, CG = (V, E). V is the set of graph vertices where each vertex v_i represents a function. E is the set of graph edges where each edge e_{ij} represents a function call to the child function v_j by

[2]With a large number of objects, fast, good quality estimators are of course more practical than detailed logic synthesis

the parent function v_i. The application representation assumes that there are no recursive function calls.

Each edge is associated with 2 weights (cc_{ij}, ct_{ij}). cc_{ij} represents the call count, i.e, the number of times function v_j is called by its parent v_i. ct_{ij} represents the HW-SW communication time, i.e, if v_i is mapped to SW and its child v_j is mapped to HW (or vice-versa), ct_{ij} represents the time taken to transfer data between the SW and the HW unit for each call. An important assumption is made that vertices mapped onto the same computing unit have negligible communication latency, i.e SW-SW or HW-HW communication time can be considered to be 0 for practical purposes.

Each vertex v_i is associated with 3 weights (t_i^s, t_i^h, h_i). t_i^s represents the software (SW) execution time for a given vertex, i.e, the time required to execute the function corresponding to v_i on the processor. t_i^h represents the hardware (HW) execution time, i.e, the time required to execute the function on the HW unit. The hardware implementation of the function requires area h_w on the HW unit. Note that this definition works off a single Pareto point - in this work, we do not consider compiler (synthesis) optimizations leading to multiple HW implementations with different area and timing characteristics.

A partitioning of the vertices can be represented in the following manner: $P = \{v_0^s, v_1^h, v_2^s.....\}$. This denotes that in partitioning P, vertex v_0 is mapped to SW, v_1 is mapped to HW, v_2 is mapped to SW, etc. Two key attributes of a partitioning are (T^P, H^P). T^P denotes the execution time of the application under the partitioning P, H^P denotes the aggregate area of all components mapped to hardware under partitioning P.

4 Efficient computation of execution time change metric

Given a sequential execution paradigm and a call-graph specification, the execution time of a vertex v_i is computed as the sum of its self-execution time and the execution time of its children. The execution time for v_i additionally includes HW-SW communication time for each child of v_i mapped to a different partition. Thus, if T_i^P denotes the execution time for vertex v_i under a given partitioning P

$$T_i^P = t_i + \sum_{j=1}^{|C_i|}(cc_{ij} * T_j) + \sum_{j=1}^{|C_i^{diff}|}(cc_{ij} * ct_{ij})$$

where t_i is either t_i^h or t_i^s depending on whether v_i is currently mapped to HW or SW in the partitioning P. C_i represents the set of all children of vertex v_i, C_i^{diff} represents the set of all children of v_i mapped to a different partition. Note that if v_0 corresponds to *main* in a 'C' program, T_0^P equals the complete program execution time when partitioning P specifies whether vertices are mapped to HW or SW.

For a partitioning P, the *execution time change* metric Δ_i^P, for a vertex v_i, is defined as the change in execution time when the vertex v_i is moved to a different partition. That is, Δ_i^P denotes the change in T_0^P when vertex v_i is mapped to a different partition.

From the definition of the *execution time change* metric, it would appear that when a vertex is moved to a different partition, the metric values for all its ancestors would need to be updated. Indeed, in previous work, [8], the change equations assumed it was necessary to update ancestors all the way to the root. This, however, is not the case: it is only necessary to update the metric values for immediate neighbours. In the following, we first give an intuition for this through a simple example, and follow up with a proof.

Consider the simple example in Figure 3.

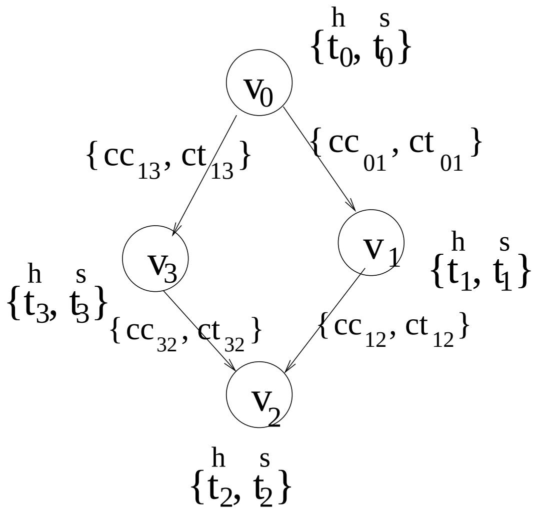

Figure 3. simple call graph

For the graph shown in Figure 3, the program execution time corresponding to a partitioning P = $\{v_0^s, v_1^s, v_2^s, v_3^s\}$ is given by

$$T_0^P = t_0^s + cc_{01} * (t_1^s + cc_{12} * t_2^s) + cc_{03} * (t_3^s + cc_{32} * t_2^s)$$

If the vertex v_0 is moved to HW, i.e we have a new partitioning P_0 = $\{v_0^h, v_1^s, v_2^s, v_3^s\}$,

$$T_0^{P_0} = t_0^h + cc_{01} * (t_1^s + ct_{01} + cc_{12} * t_2^s) + cc_{03} * (t_3^s + ct_{03} + cc_{32} * t_2^s)$$

The *execution time change* metric for vertex v_0 is

$$\Delta_0^P = (T_0^1 - T_0) = (t_0^h - t_0^s) + cc_{01} * ct_{01} + cc_{03} * ct_{03} \text{ -} \qquad \textbf{Equation (A)}$$

We can similarly compute Δ_1^P, Δ_2^P, Δ_3^P.

Next, let us consider a partitioning P_2 = $\{v_0^s, v_1^s, v_2^h, v_3^s\}$ where the vertex v_2^h is mapped to HW.

For this partitioning, execution time for vertex v_0 is given by

$$T_0^{P_2} = t_0^s + cc_{01} * (t_1^s + cc_{12} * (ct_{12} + t_2^h)) + cc_{03} * (t_3^s + cc_{32} * (ct_{32} + t_2^h))$$

If the vertex v_0 is now moved to HW, i.e we have a new partitioning P_{20} = $\{v_0^h, v_1^s, v_2^h, v_3^s\}$,

$$T_0^{P_{20}} = t_0^h + cc_{01} * (t_1^s + ct_{01} + cc_{12} * (ct_{12} + t_2^s)) + cc_{03} * (t_3^s + ct_{03} + cc_{32} * ((ct_{32} + t_2^s))$$

Thus,

$$\Delta_0^{P_2} = (T_0^{P_{20}} - T_0^{P_2}) = (t_0^h - t_0^s) + cc_{01} * ct_{01} + cc_{03} * ct_{03} \text{ -} \qquad \textbf{Equation (B)}$$

Equations (A) and (B) are identical. That is, when vertex v_2 is moved to a different partition, the metric Δ_0^P remains unchanged for a non-immediate ancestor v_0.

In order to prove that the above result holds in the generic case, the simple example gives us the insight that we need to represent the expression T_0^P in a form slightly differently from the original definition. We define the *aggregate call count*, CC_i for a vertex v_i in the following recursive manner: $CC_i = \sum_{j=1}^{|P_i|}(cc_{ji} * CC_j)$. P_i represents the set of all parent vertices (all functions it is called from) for the vertex v_i. CC_0, i.e the *aggregate call count* for the root vertex, is 1.

Armed with the above definition, we can proceed to prove the following lemma:

Lemma: For any two vertices v_x and v_y, if a vertex v_x is moved to a different partition, Δ_y needs to be updated only if there is an edge (x,y) or, (y,x). This update involves changing exactly one term in Δ_y, i.e this update can be done in $O(1)$ time per edge.

Proof:

CC_i obviously represents the exact number of times the function corresponding to vertex v_i is called along all possible paths from the root. Now, if we recursively expand T_0^P, an unrolled representation for T_0^P is:

$$T_0^P = \sum_{i=0}^{|V|}(CC_i * t_i) + \sum^{(i,j)\ \varepsilon\ E}(\delta_{ij}^P * CC_i * ct_{ij}).$$

δ_{ij}^P is the *Kronecker delta* function defined for edge (i,j) in the partitioning P - it takes a value of 1 if the vertex v_i and its child vertex v_j are mapped to different partitions, 0 otherwise. The first expression has exactly one term per vertex and the second expression has exactly one term per edge. If we now evaluate $T_0^{P_x}$, the new execution time when vertex v_x is moved to generate the new partitioning P_x,

$$T_0^{P_x} = \sum_{i=0}^{(V-v_x)}(CC_i * t_i) + t_x^{P_x} * CC_x + \sum^{(i,j)\varepsilon(E-X)}(\delta_{ij}^P * CC_i * ct_{ij}) + \sum^{(i,j)\ \varepsilon\ X}(\delta_{ij}^{P_x} * CC_i * ct_{ij})$$

where X is the set of all edges adjacent to vertex v_x, including its immediate parents and immediate children. The *Kronecker delta* values can change only for edges adjacent to v_x. We can now evaluate $\Delta_x^P = T_0^{P_x} - T_0^P$ corresponding to the change in execution time when vertex v_x is moved.

$$\Delta_x^P = CC_x * (t_x^{P_x} - t_x) + \sum^{(i,j)\ \varepsilon\ X}((\delta_{ij}^{P_x} - \delta_{ij}^P) * CC_i * ct_{ij})$$

We can similarly compute Δ_y^P for any other vertex v_y as

$$\Delta_y^P = CC_y * (t_y^{P_y} - t_y) + \sum^{(i,j)\ \varepsilon\ Y}((\delta_{ij}^{P_y} - \delta_{ij}^P) * CC_i * ct_{ij}) \qquad \textbf{Equation (C)}$$

Next, we evaluate the execution time $T_0^{P_{xy}}$ corresponding to the new partitioning when vertex v_x is moved, followed by vertex v_y being moved.

$$\begin{aligned}T_0^{P_{xy}} = {} & \sum_{i=0}^{(V-v_x-v_y)}(CC_i * t_i) + t_x^{P_x} * CC_x + t_y^{P_y} * CC_y + \\ & \sum^{(i,j)\varepsilon(E-X-Y)}(\delta_{ij}^P * CC_i * ct_{ij}) + \sum^{(i,j)\varepsilon(X-[x,y])}(\delta_{ij}^{P_x} * CC_i * ct_{ij}) + \\ & \sum^{(i,j)\varepsilon Y}(\delta_{ij}^{P_{xy}} * CC_i * ct_{ij})\end{aligned}$$

where $[x,y]$ denotes either (x,y) or (y,x). Note that from our problem definition we can not have both (x,y), and, (y,x).

Thus, we can compute $\Delta_y^{P_x} = T_0^{P_{xy}} - T_0^{P_x}$ as

$$\begin{aligned}\Delta_y^{P_x} &= CC_y * (t_y^{P_y} - t_y) -^{((i,j)=[x,y])} (\delta_{ij}^{P_x} * CC_i * ct_{ij}) - \sum^{(i,j)\varepsilon(Y-[x,y])}(\delta_{ij}^P * CC_i * ct_{ij}) + \\ &\qquad \sum^{(i,j)\ \varepsilon\ Y}(\delta_{ij}^{P_{xy}} * CC_i * ct_{ij}) \\ &= CC_y * (t_y^{P_y} - t_y) + \sum^{(i,j)\varepsilon(Y-[x,y])}((\delta_{ij}^{P_{xy}} - \delta_{ij}^P) * CC_i * ct_{ij}) + \\ &\qquad {}^{((i,j)=[x,y])}((\delta_{ij}^{P_{xy}} - \delta_{ij}^{P_x}) * CC_i * ct_{ij}) \qquad \textbf{Equation (D)}\end{aligned}$$

δ_{ij} depends only on whether vertices v_i and v_j are in the same partition, or in different partitions. That is, $\delta_{ij}^{P_{xy}} = \delta_{ij}^P$ if $(i,j) \neq [x,y]$. So, comparing Equations (C) and (D), we have

$$\Delta_y^{P_x} - \Delta_y^P =^{((i,j)=[x,y])} ((\delta_{ij}^{P_{xy}} - \delta_{ij}^{P_x}) - (\delta_{ij}^{P_y} - \delta_{ij}^P)) * CC_i * ct_{ij}$$

Thus we have proved that when vertex v_x is moved to a different partition, Δ_y needs to be updated only if there is an edge $[x,y]$. Also, the update involves changing exactly one term in Δ_y.

5 Simulated annealing

5.1 Algorithm Description

The concept of simulated annealing originated in theoretical physics where Monte-Carlo methods are employed to simulate phenomena in statistical mechanics. Essentially the simulation method considers a system consisting of a huge number of particles at an initial temperature T: providing a random displacement to a particle causes a change in the system energy and a sufficient number of such displacements causes the system to reach statistical equilibirium at T. In simulated annealing, the physical process of cooling a liquid to its freezing point to obtain an ordered structure is simulated at several such temperature steps by letting the system reach equilibrium at each temperature.

The physical process described above was associated with combinatorial optimization problems in [20]. An objective function is associated with the energy of a physical system and system dynamics are imitated by random local modifications of the current solution- a feasible solution corresponds to a system state and an optimal solution corresponds to a state of minimal energy. Thus, key parameters in any formulation are the initial temperature T, the cooling (annealing) schedule which mandates how the temperature is decremented, and the number of iterations at each step.

```
        **Algorithm SA**
while (NOT_EQUILIBRIUM)
   For $i = 1\ \ to\ \ I_t$     // iterations at current temperature
      P’ = random perturbation of the current configuration, P
      $COST^{\Delta}$ = COST(P’) - COST(P)
      if ($COST^{\Delta} < 0$)
            P = P’
      else
            generate random number x $\varepsilon$ [0,1]
      if ($x < e^{-COST^{\Delta}/T}$)
            P = P’
   endfor
   UPDATE T () (from annealing schedule)
   EVALUATE_EQUILIBRIUM_CONDITION()
endwhile
```

In HW-SW partitioning, perturbation is commonly defined as a move of a single vertex from HW to SW and vice versa, though experiments have been conducted with perturbations involving multiple moves [9]. A typical cost function is a linear combination of normalized metrics [1], [5], [3]. For our problem, the two metrics we need to consider are the execution time of a partitioning and the hardware area.

For a simulated annealing approach to HW-SW partitioning, execution time is primarily driven by the computation cost of the cost function. It is computationally expensive to traverse a graph with 100’s to thousands of vertices for every new configuration. For our given problem, we can simply use the *execution time change* metric defined earlier to update the execution time for a new partitioning. Let us consider a partitioning P with attributes (T^P, H^P). A move of vertex v_i from HW to SW generates a new partitioning P_1 with attributes $(T^P + \Delta_i^P, H^P - h_i)$. Similarly, a move of vertex v_i from SW to HW generates a new partitioning P_2 with attributes $(T^P + \Delta_i^P, H^P + h_i)$.

If we computed the execution time of a partitioning by a simple traversal of the call-graph, each move would have a computation cost of $O(E)$. The *execution time change* metric enables us to update the execution time simply by updating the immediate neighbours of a vertex. Since the average indegree and outdegree of a call graph is expected to be a low number, the average cost of a move is very low and enables the simulated annealing algorithm to do a very rapid evaluation of the search space.

5.2 Cost function for simulated annealing

It is well-known that simulated annealing is a good approach for high quality solutions to problems with a single optimization criterion, but it doesn't behave as well on problems with multiple objectives. In HW-SW partitioning, a significant amount of work considers only valid configurations that satisfy constraints, thus effectively reducing the ability of the algorithm to "hill-climb". Often, a statically weighted linear combination of metrics is used in an attempt to overcome the limitation of simulated annealing in handling a multiobjective problem.

In this section we provide the intuition for developing cost functions that explore points often not considered in traditional cost functions. Simulated annealing inherently uses randomization to overcome local minima- moves that do not improve the optimization objective are accepted with some probability. Our goal is to guide the algorithm towards potentially more interesting design points by explicitly forcing the algorithm to accept apparently bad moves when we are far away from the goal. Simultaneously, we force the algorithm to probabilistically reject some apparently good moves that would always be accepted by most heuristics. As an example, when we are far away from our optimization goal, we would prefer not to always accept a move that improves execution time only slightly at the cost of a significant amount of hardware area.

Our first observation about cost functions is that rather than the absolute value of the cost function, the primary concern of the SA algorithm is to take a decision after every perturbation whether the cost function changed for the better or for the worse. In case the new configuration seems undesirable , we additionally need a quantitative evaluation of the degree of inferiority to a desired solution. So, instead of defining a cost function, we attempt to define a delta function on the key parameters for the given problem- the two parameters, obviously change in execution time, area.

In Figure 4, we try to provide the intuition behind our approach. In this diagram, each possible partitioning P is represented as a point in the two-dimensional plane with x and y co-ordinates. The x-axis represents the execution time corresponding to the given partitioning, while the y-axis represents the aggregate hardware area for components mapped to HW. The vertical lines T_{min} and T_{max} represent the execution times corresponding to an all-hardware and an all-software solution respectively. The horizontal line A_c represents the area constraint. To solve our problem of minimizing execution time under a hard area constraint, we effectively need to search for a point as close as possible to the upper left corner of the bounded rectangular region A. This is of course based on the implicit assumption that all partitioning objects cannot possibly be accomodated in

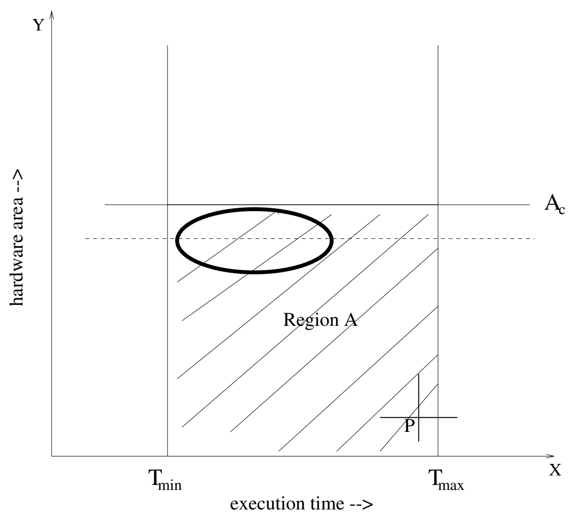

Figure 4. Solution space

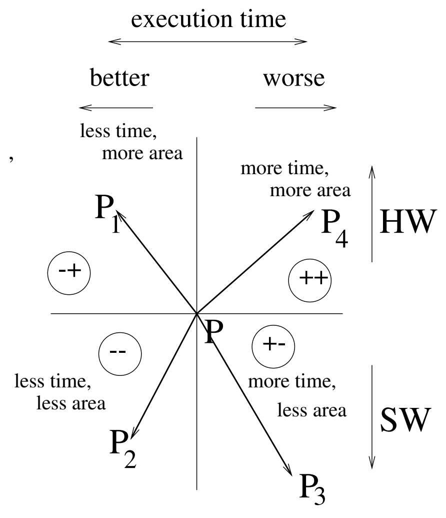

Figure 5. Neighbourhood move

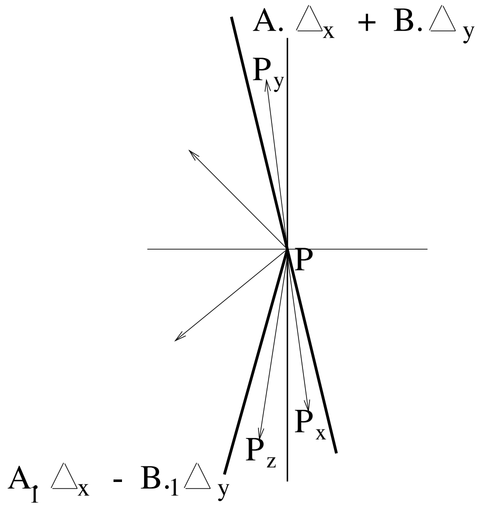

Figure 6. Cost functions

HW- and, mapping as many objects as possible to HW is likely to yield the most improvement in the execution time.

Effect of a move

Let us now consider the implication of moving a single component in the partitioning P to generate a new partitioning, say P_1, as shown in Figure 5. P_1 corresponds to a partitioning with improved (less) execution time, and more HW area, i.e P_1 is generated by some move which a SW component is moved to HW. Similarly the point P_2 corresponds to a move with less hardware area, i.e, a move from HW to SW. More generally, when a single component in partitioning P is moved to generate the new partitioning P_i, the new point P_i lies in one of the four quadrants centred at P. A partitioning P_1 with improved execution time and additional HW area lies in the quadrant $(-t,+h)$, represented in Figure 5 as $(-,+)$. Similarly, a partitioning P_2 with improved execution time and reduced HW area lies in the quadrant $(-t,-h)$, represented as $(-,-)$ and so on for partitionings P_3, P_4.

We next consider the evaluation of a cost function ($A*\Delta_T+B*\Delta_A$) at the point P. This cost function corresponds to a random move generated by the SA algorithm. Δ_T is the difference in execution time, and is the same as the *execution time change* metric for the vertex to be moved. Δ_A is the change in area caused by the move: Δ_A is positive for a SW-HW move and negative for a HW-SW move. A and B are weights that include the normalization factors required to be able to combine the two cost function components which are in completely different units.

This cost function is a simple straight line through P splitting the region around P into 2 equal parts. In simulated annealing, a random number is generated to choose a possible neighbourhood move. Based on the cost function, this move is either accepted blindly or, accepted with some probability dependent on the temperature. In traditional cost functions like [8], where the hardware area component of the cost function is ignored as long as the constraint is satisfied, essentially every random move that improves the execution time component of the cost function is accepted with a probability of 1. In Figure 5, this corresponds to blind acceptance of all moves represented by P_1, P_2, i.e all moves lying in quadrants $(-t,+h)$, $(-t,-h)$ represented in the figure as $(-+)$, $(--)$. The hardware area component of the cost function plays a role only when a move leads to a constraint violation- this component is effectively considered irrelevant in a significant amount of the search space.

Quadrant-based cost function

Now, let us consider P to specifically correspond to a partitioning in which few components have been mapped to hardware and hence the execution time is expected to be relatively close to the software execution time. In terms of the bounded rectangular region A in Figure 4, P is nearer the lower right corner. For such a point P, we would like to bias the move acceptance such that:

(a) we provide additional weightage to certain moves that cause the execution time to deteriorate slightly but in return free up a large amount of HW area. These are represented by P_x in Figure 6.

(b) we reduce the weightage on certain moves that improve execution time slightly but consume an additional large amount of HW area. These are represented by P_y in Figure 6.

(c) we reduce the weightage on certain moves that improve the execution time slightly but free up a large amount of HW area. These are represented by P_z in Figure 6.

Providing additional bias to the set of moves P_x is a non-traditional approach but we expect such moves to open up more combinatorial possibilities. We achieve our objective of providing additional bias to such moves by introducing a cost function $(A*\Delta_T + B*\Delta_A)$, $A >> B$ in the quadrant $(+t,-h)$. In cost functions that ignore the HW area component far from the objective, all moves in this quadrant carry a penalty and are accepted with some probability dependent on temperature. By introducing a cost function in this quadrant, the set of moves P_x are always accepted.

A similar reasoning of enabling the cost function to explore more combinatorial possibilities lies behind our choice of reducing weightage on the set of moves P_y. For these points, we are effectively discouraging moves with a slight gain in execution time, but with significantly increased HW area.

Our reasoning behind reducing weightage for the set of moves P_z is somewhat different. For these moves, we are actually attempting to guide the search away from making moves that do not appear to be making progress towards our desired solution space. Intuitively, for a partitioning where there are relatively few HW components, the HW-SW communication cost can potentially play a dominant role. For moves like P_z, freeing up a large amount of HW could potentially result in a slight improvement in execution time due to significant reduction in HW-SW communication. Blindly accepting such moves translates to attempting to reduce communication cost between some vertex v_x mapped to HW and its neighbours in SW by moving back v_x to SW. When we are far from our desired solution space, we would instead prefer to encourage the algorithm to reduce communication cost by adding more of its neighbours to HW.

Dynamic weighting

We next consider the notion of dynamically weighting the components of the cost function as suggested in [5]. Changing the A and B components dynamically affects the search region by dynamically changing the slope of the line through P. This is a powerful technique that has a big impact on the effectiveness of the search. In [5], this technique was applied only when the current solution was somewhat close to the constraint. The algorithm in [5] essentially focussed on satisfying its primary objective, i.e the timing constraint- once this constraint was satisfied, dynamic factors were used to increase weightage of its secondary objective, minimizing the HW area. We however, apply a dynamic weighting factor to our cost functions in various regions in an attempt to better guide the search. Conceptually we attempt to guide our search more towards the top left corner of the bounded region by dynamically weighting the time component with the distance from the boundary.

Among the other key issues we have considered in formulating our cost function are the impact of boundary violations, i.e when a move leads us to a partitioning with HW area greater than the constraint. We penalize all such moves with a factor proportional to the extent of the boundary violation. We can clearly achieve this with a high weightage on the area component , i.e a function $(A * \Delta_T + B * (Area_{new} - A_c))$, where $B >>> A$. Similarly when a move leads from an invalid partitioning to a valid partitioning, we reward it with a factor proportional to the extent that it is inside the boundary.

Another important aspect of our cost function is the notion of a threshold. When we are very close to the boundary, we need a cost function that has only a slight bias towards the component representing execution time. In our cost function, the time component is dynamically weighted by the distance from the boundary- we have observed experimentally that close to the boundary, desirable weights for the time component in this region are even lower than what our cost function provides. Thus, we needed to add the notion of a threshold region very close to the boundary where we explicitly assign a lower weightage to the time component of the cost function.

Based on the above discussion, our cost function is algorithmically described as:

```
if (current partititioning is a valid solution)
    if (move causes boundary violation)
        Significant penalty proportional to area violation        (i)
    else if (current partitioning is very near to boundary)
        Slightly reduced weightage on time (as compared to (iii))
    else
        if (move in quadrant (–,–))
            $(A_1 * \Delta_T + B_1 * \Delta_A)$, where $A_1 >> B_1$        (iii)
        else
            $(A_2 * \Delta_T + B_2 * \Delta_A)$,        (iv)
else // (current partititioning lies outside boundary)
    a mirror image of the above set of rules.
```

In Equations (iii) and (iv), the terms A_1 and A_2 are dynamically weighted by the distance from the boundary.

An actual code snippet corresponding to (iv):

```
timeComponent = 2.4 * (timeChange/currentExecutionTime) * (areaConstraint/currentArea)
areaComponent = areaChange/areaConstraint
penalty = timeComponent + areaComponent
```

The time change component of the cost function is normalized with the current execution time and dynamically weighted with a number representing the distance from the desired solution region. The HW area component of the cost function is normalized with respect to the areaConstraint.

5.3 Key parameters for Simulated Annealing

A key aspect of any simulated annealing formulation is the multitude of parameters involved. While theoretical analysis with Markov chains in work such as [19] have shown that simulated annealing approaches can generate the global optima provided certain conditions are satisfied, such analysis does not provide much information on actual choice of the various parameters. We next present the parameter settings in our implementation.

Initial temperature T

This is commonly chosen as a large value to ensure that most moves are accepted when execution starts. Our experiments confirmed that above a certain threshold, the correlation between the starting temperature and other problem dimensions (such as number of partitioning entities) is relatively low. So, we kept the initial temperature T fixed at 5000 for our entire set of experiments. All results presented in the following experimental section correspond to T = 5000.

Cooling schedule

This dictates how the temperature parameter is decremented at each step. While different cooling schedules have been proposed in the literature such as the Lundy-Mees schedule [16], we chose the widely-used *geometric* schedule. In this schedule, the new temperature is given by $T_{new} = \alpha * T$ where α is a constant that typically varies between 0.9 - 0.99. After initially conducting a wide range of experiments, we fixed α at 0.96. All our presented experimental results correspond to α = 0.96.

Number of Iterations at current temperature I_t

This is a key aspect that specifies the number of moves at the current temperature before the cooling schedule is applied to decrement the temperature. Recent work in HW-SW partitioning often seem to either eliminate the inner loop completely [3], only relying on a global termination condition, or, do not explicitly mention their choice [1]. Our observations indicate that this criterion plays a significant role in determining solution quality. This parameter is frequently set to be a function of the number of partitioning objects, such as a simple constant multiple in [17], a polynomial (approximately quadratic) in [9]. Based on our extensive set of experiments, we propose I_t = i iterations in the $i'th$ temperature step. This is in keeping with our overall rationale behind developing our cost function- our aggregate strategy is to initially make relatively quick strides towards the desired solution space, and subsequently spend more computational effort searching for a very high quality solution.

Global Equilibrium

The algorithm termination criterion is again a very key issue. In existing work this is often formulated as a small number of constant temperature iterations without any improvement in solution quality– this constant is frequently chosen as 3 in work such as [9], [17]. However with our ”more global” approach to the problem, we choose the stopping criterion as the number of cumulative moves that do not produce any improvement. This naturally has a strong correlation with the problem size in terms of partitioning objects– so, we scale this global stopping criterion from 5000 moves without improvement for graphs with fifty vertices, to 15000 moves without improvement for graphs with a thousand vertices.

6 Experiments

As shown in Figure 7, we explored a very large space of possible designs by generating graphs which varied the following set of parameters: (1) varying indegree and outdegree (2) varying number of vertices (3) varying CCRs (computation-to-communication ratio). (4) varying area constraints.

6.1 Experimental setup

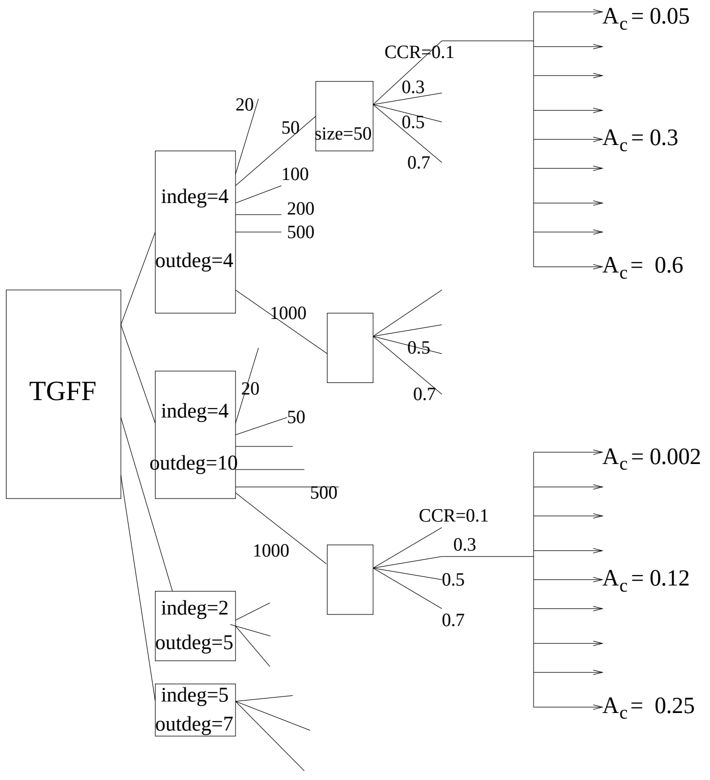

Figure 7. Set of experiments

We used the graph generator TGFF [15] to generate the graphs used in our experiments. TGFF is a parameterizable generator that can accept user specifications like maximum indegree, outdegree of the vertices. We additionally augmented TGFF for our experiments- an example of an augmentation was one that enabled TGFF to generate HW execution times for vertices such that the HW execution time of a vertex was faster than the SW execution time by a number between 3 and 8 times.

Let S = $\{20, 50, 100, 200, 500, 1000\}$ denote the range of graph sizes generated where size corresponds approximately to the number of vertices in the graph. As an example, for a graph size 50, TGFF generates a graph with between 48 to 52 vertices. We chose S to observe how our algorithm worked on a large range of graph sizes. Let CR = $\{0.1, 0.3, 0.5, 0.7\}$ denote the set of CCR's (communication-to-computation ratio). The notion of CCR is very important in partitioning and

scheduling algorithms that consider communication between tasks/functions. A CCR of 0.1 means that on an average, communication between functions/tasks in a call-graph/task graph requires 1/10'th the execution times of the functions/tasks in the graph. As CCR increases, communication starts playing a more important role in coarse-grain partitioning and scheduling algorithms.
Step 1 The maximum indegree and outdegree of a vertex were set to 4 each, which are reasonably representative of realistic call-graphs. Corresponding to these fixed parameters, we generated a set of graphs with the following characteristics. Each run of SA chose a graph size from S = $\{20,50,100,200,500,1000\}$; for each graph size we chose CCR from CR = $\{0.1,0.3,0.5,0.7\}$ Thus, we effectively generated a set of graphs $|S|$ X $|CR|$. Note that in the tables that follow, graphs with size 50 are denoted as $v50$, graphs with size 100 are denoted as $v100$, etc.
Step 2 For each of the graphs generated in Step 1, we varied the area constraint A_c as a percentage of the aggregate area needed to map all the vertices to HW. On one extreme, we set A_c such that very few partitioning objects would fit into HW, while at the other extreme, a significant proportion of the objects would fit into HW.
Step 3 We repeated the above two steps with a maximum indegree and outdegree of (i) 4 and 10 (ii) 2 and 5 (iii) 5 and 7.

As a consequence of our experimental setup, the experimental data presented represents information collected from over 12000 experiments.

To measure the quality of results, we simply record the program execution times computed by the SA algorithm with our new cost function, and the KLFM algorithm as in [8]. In prior work, however, experiments to measure the quality of a partitioning algorithm have often been formulated by forcing constraint violations and attempting to integrate the degree of violations into some unitless number, as in [8], [1].

For a given design configuration, if T_{kl} is the execution time of the partitioned application computed by the KLFM algorithm, and T_{sa} is the execution time of the partitioned application computed by the SA algorithm, our quality measure is the performance difference given by:

$$(T_{kl} - T_{sa})/T_{kl} * 100$$

Thus, a positive number, say, 5%, implies that the KLFM algorithm computed an execution time better than SA by 5%, while a negative number, say -10%, implies that the SA algorithm computed an execution time better than the KLFM algorithm by 10%.

6.2 Experimental results and key observations

The experimental results are classified under two categories. The first category of experimental results confirm the benefits of our approach over the KLFM approach by demonstrating that our approach generates superior quality results **AND** is asymptotically faster. The second category of results quantifies the effect of our dynamic SA cost function and parameter settings by comparing with other SA-based approaches.

6.2.1 Proposed SA (SA-new) Vs KLFM

Figures 8 through 11 represent the complete data for all graphs with 50 vertices. Appendix A contains the rest of the data for graphs of other sizes: Figures 13 through 16 in Appendix A

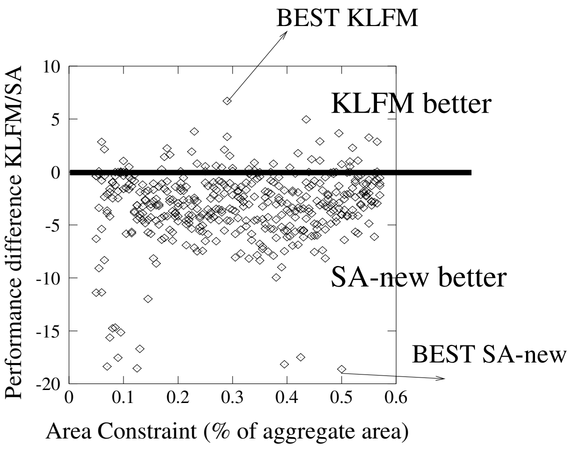

Figure 8. v50, CCR 0.1: Performance Vs constraint

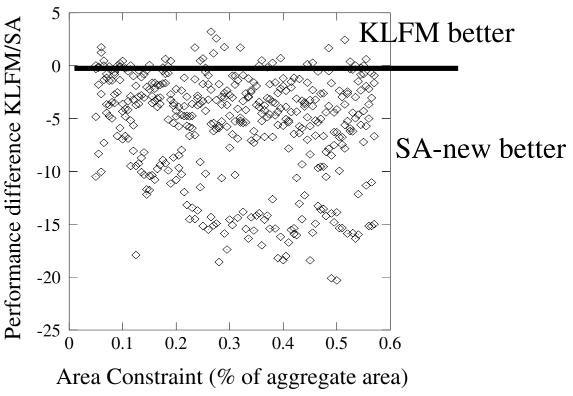

Figure 9. v50, CCR 0.3: Performance Vs constraint

Graph category	BEST	WORST	AVG	BETTER%	BETTER-5%	WORSE-5%
v20	-24.9%	12.3 %	-4.17 %	72%	36.5%	0.7%
v50	-22.9%	6.7 %	-5.75 %	93%	45.4%	0.1%
v100	-18.2%	5.7%	-5.47%	96%	48%	0.05%
v200	-13.9%	4.3%	-3.74%	90%	33%	0%
v500	-16 %	6.8 %	-4.53 %	87%	47.3%	0.9%
v1000	-13.7%	6.4%	-4.17%	81%	43.5%	0.7%

Table 1. Solution quality: KLFM (all Software) Vs SA-new

represent the complete data for all graphs with 20 vertices, Figures 17 through 20 for all graphs with 100 vertices, etc.

Let us consider Figure 8– this represents data collected for graphs with 50 vertices, and a fixed CCR of 0.1 for various area constraints. The *x*-axis corresponds to the various area constraints and the *y*-axis corresponds to the performance difference. In this figure, a significant majority of points show negative performance difference (below 0)– this indicates that our SA formulation mostly generates better results than the KLFM approach. The point $BEST\ SA - new$ represents the best performance improvement by our SA formulation over the KLFM approach, while point $BEST\ KLFM$ represents the best solution computed by the KLFM algorithm. Similarly, Figure 9, Figure 10, Figure 11 represent the data for graphs with 50 vertices and a CCR of 0.3, 0.5, 0.7 respectively.

In Table 1, we summarize the aggregate results from all experiments comparing the SA with our new cost function (SA-new) against the KLFM algorithm starting with an initial configuration of all vertices mapped to SW. The column header $BEST$ represents the best improvement in execution time computed by the SA-new approach compared to the KLFM approach for the set of

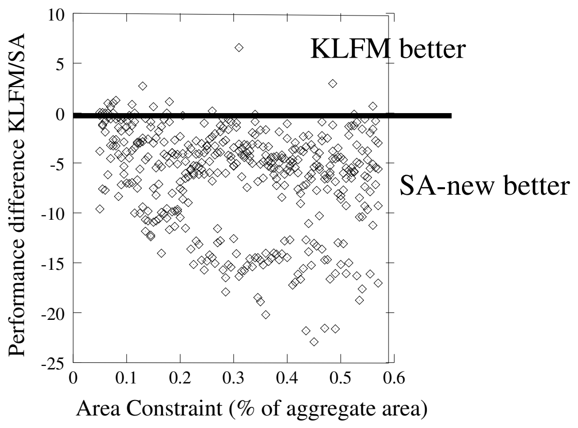

Figure 10. v50, CCR 0.5: Performance Vs constraint

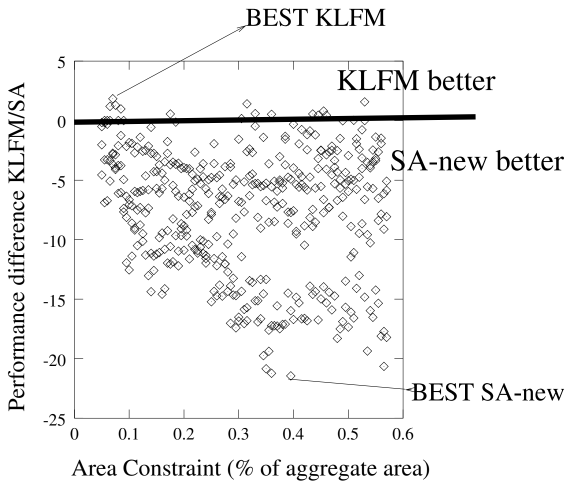

Figure 11. v50, CCR 0.7: Performance Vs constraint

experiments represented by a row like $v50$. Entries that are more negative indicate that SA-new is better- for example, an entry of -16% means the execution time computed by SA-new is better than KLFM by 16%. The column header $WORST$ represents the best results produced by the KLFM algorithm. The column header AVG represents the average deviation between the two algorithms. The column header $BETTER\%$ represents the overall percentage of test cases for which our approach generates better results, i.e lower execution time for the corresponding test case. The column headers $BETTER-5$ and $WORSE-5$ indicate the overall percentage of test cases for which the results generated are better and worse respectively by 5%.

As can be observed from column 4 ($BETTER$), in an overwhelming majority of test cases, SA-new produces results that are better than KLFM– column 3 (AVG) shows that the average improvement over KLFM for all graph sizes is close to 5%. This is confirmed by columns 5 ($BETTER-5$) and column 6 ($WORSE-5$) where we clearly see that the solution quality is better by over 5 % in a significant number of experiments while the number of experiments where KLFM is better by over 5% is negligible. Additionally, for non-trivial graph sizes of 50 vertices or more, column 2 ($WORST$) shows that the best results for KLFM are at most 7% better than the results computed by our new cost function.

Table 2 shows the run-times required by KLFM and SA-new to generate the results in Table 1. Again, the run-time is averaged over the experimental set– as an example, in Table 2 the data corresponding to (row $v500$, column KLFM) was obtained as the average runtime over the set of a couple of thousand experiments with KLFM on graphs with 500 vertices. As can be clearly seen from Table 2 and Figure 12, the run-time of our SA formulation is asymptotically better than the run-time of the KLFM formulation- for graphs with around 1000 vertices, the run-time of our algorithm is almost 5 times faster.

Observations

Overall, Figures Figure 8 through Figure 11 (and Figure 13 through Figure 32 in the Appendix)

Graph category	SA-new runtime(s)	KLFM runtime(s)
v20	.07	.05
v50	.08	.05
v100	.1	.07
v200	.19	.11
v500	.25	.48
v1000	.36	1.6

Table 2. Run-time performance comparison

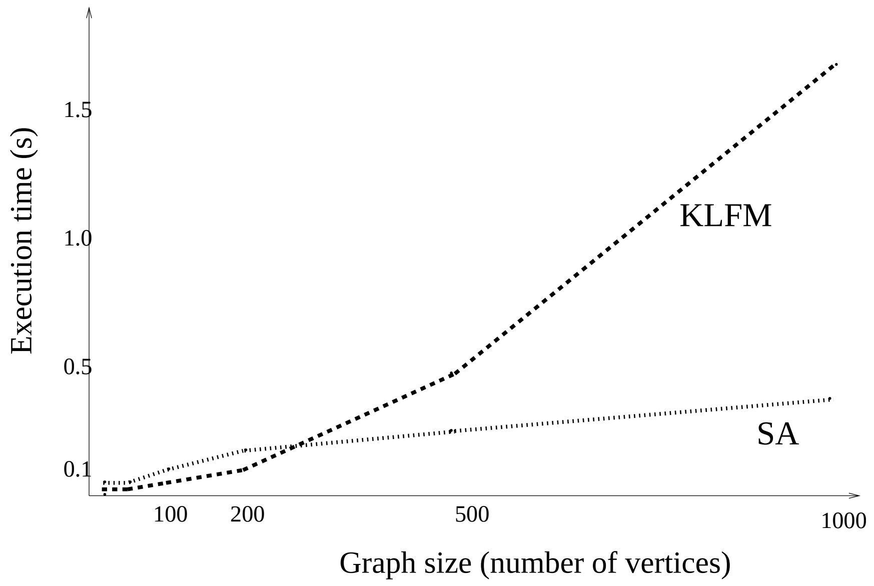

Figure 12. Run-time performance plot for algorithms

Graph category	BEST	WORST	AVG	BETTER%	BETTER-5%	WORSE-5%
v20	-23.8%	12.3 %	-1.89%	52%	15.4%	0.9%
v50	-19.6%	6.7%	-4.1%	88%	27.9%	0.1%
v100	-14%	9.3%	-3.1%	87%	20.7%	0.6%
v200	-10.1%%	6.8 %	-2.51%	81%	15.6%	0.3%
v500	-12.7%	7.1%	-2.84 %	82%	20.7%	0.1%
v1000	-13.3%	6.4%	-2.81 %	77%	26.5%	0.1%

Table 3. Solution quality: KLFM (best) Vs SA-new

show that our approach generates better quality results than the KLFM approach when the design space is larger. We next take a brief look at how key aspects of the design space affect the solution quality.

Effect of area constraint

At lower area constraints where a small percentage of nodes can fit into HW, the exploration space is relatively limited. In such scenarios the KLFM approach generates reasonably good results. This shows up in graphs with 500 and 1000 vertices where the KLFM frequently generates slightly better results at an area constraint of 0.05 (at most 5% of vertices can be mapped to HW). However, as area constraint grows, increasing the scope for exploration, our approach starts generating superior solutions. Also, as Table 1 showed, the best results generated by the KLFM are slightly better, while SA-new often generates much better results.

Effect of CCR

Comparing Figure 8 with Figure 11, we see the effect of communication in the design space. The number of experiments that show an improvement of over 15% is many more in Figure 11 with a higher CCR of 0.7. As CCR increases, the scope of communication-computation tradeoff increases. Our approach, SA-new, is able to do a much better job of exploring this space.

Proposed SA (SA-new) Vs KLFM (best)

KLFM based partitioning approaches are known to be sensitive to the initial partitioning. So, we let the KLFM start from some different initial configurations and obtained the best result for each experiment from this set of independent KLFM runs. In Table 3, we compare the quality of results obtained from SA-new with this *best of KLFM* heuristic. As the results in Table 3 confirm, the quality of results obtained from this set of KLFM runs is superior to that obtained from a KLFM with all vertices initially in software. However, the quality of results generated by SA-new is still superior with a significant percentage of cases showing imporovement by over 5% while the number of test cases where SA-new performs worse by over 5% is still negligible.

Thus, Tables 1, 2, and 3 confirm that our approach indeed generates superior results in general and asymptotically executes much faster than the KLFM-based approach.

Graph category	BEST	WORST	AVG	BETTER%	BETTER-5%	WORSE-5%
v20	-15.2%	13.5 %	0.21%	8%	0.7%	2%
v50	-28.8%	9.4 %	-1.86%	75%	8%	1%
v100	-30%	7.7%	-4.4%	83%	36.3%	0.2%
v200	-18.6 %	3.1	-7.3%	93%	75%	0%
v500	-22.6%	7.9%	-10.1 %	92%	82.6%	0.4%
v1000	-24.8%	7.2%	-10.5 %	89%	77.1%	0.9%

Table 4. Solution quality: SA ($time + \delta(area)$ Vs SA-new

Graph category	BEST	WORST	AVG	BETTER%
v20	-32.8%	9.8 %	-15.5%	99%
v50	-56.6%	-8 %	-32.2%	100%
v100	-58.8%	-7.1%	-34.7%	100%
v200	–57.8%%	-3.6 %	-36%	100%
v500	-54.8%	-1.2%	-35.5 %	100%
v1000	-54.5%	-0.2%	-31.6 %	100%

Table 5. Solution quality: SA-previous Vs SA-new

6.2.2 Proposed SA (SA-new) Vs other SA

We next present results of experiments where we compared our SA with dynamic cost function (SA-new) with other SA-based approaches.

Static Vs dynamic cost function

In Table 4, we quantify the effect of using our dynamic cost function. For this set of experiments, we compare SA-new with a SA cost function ($time + \delta(area)$) as in [8]. The rest of the parameter settings were identical– initial temperature, cooling schedule (α), local and global stopping criterion were same.

From this table we observe that under the given parameter settings, the static SA cost function does well for small graphs with 20 vertices, but as the graph size increases, the dynamic cost function does significantly better.

Effect of parameter settings

In Table 5, we quantify the effect of changing some key parameter settings. We compare SA-new with a SA based approach, SA-previous, with cost function ($time + \delta(area)$). The parameters initial temperature and cooling schedule (α) are same for SA-new and SA-previous. However, for SA-previous,the global equilibrium criterion and local inner loop stopping criterion were set similar to work such as [9], [17]. The global stopping criterion was set to 3 temperature iterations with no improvement, and the number of iterations of the inner loop was set as a polynomial function (approximately quadratic) similar to [9].

The data shows a very significant difference in quality of results between SA-new and SA-previous– the average quality improvement from using SA-new is over 30% with almost all test cases generating better results with SA-new. We acknowledge that SA-previous would have benefited from a significant parameter tuning effort (different starting temperature, and/or different cooling schedule, etc). However, we believe it is still interesting to confirm that the presence of a hard area constraint in the SA objective makes the problem hard enough such that a simple adaptation of parameter settings from existing work is likely to lead to very low-quality results. Additionally, we hope that our extensive quantitative data will spur future researchers to continue work on similar detailed analysis of the effects of such parameter settings in the context of HW-SW partitioning.

7 Conclusion

In this work, we made two contributions. We first proved that for HW-SW partitioning of an application represented as a callgraph, when a vertex is moved between partitions, it is necessary to update the execution time metric only for the immediate neighbours of the vertex. We additionally developed a new cost function for SA that attempts to explore regions of the search space often not considered in other cost functions. Our two contributions result in a SA implementation that generates partitionings such that the execution times are frequently better by 10 % over a KLFM algorithm starting with all vertices in software for graphs ranging from 20 vertices to 1000 vertices, and the average improvement is close to 5% for a set of almost 12000 experiments. Equally importantly, the algorithm execution times are very fast- graphs with 1000 vertices are processed in less than half a second, and the algorithm is asymptotically faster than a KLFM implementation, with execution times faster by 5 times for graphs with a 1000 vertices.

Comparisons with a set of KLFM implementations starting from different initial configurations indicate that the average solution quality of results generated by our approach is still superior to the best results generated by this set of independent KLFM runs. This leads us to believe that such a fast SA formulation makes it feasible to fine-tune the function further in a real design environment to generate partitioning solutions with a quality significantly better than that obtained from a KLFM approach.

One key limitation of our current implementation is that we use a simple additive HW area estimation model that does not consider resource sharing. This limitation can potentially be overcome in a more comprehensive implementation with an approach like [10]. Another important aspect currently missing from our implementation is that it does not consider the existence of multiple area-time Pareto points obtained from different compiler (synthesis) optimizations– however, note that it is very simple to extend our SA-based approach to consider this issue. A move selection instead of being from HW to SW could potentially be simply from any implementation point to another- our neighbourhood update mechanism is still valid and the run-time of our approach in this scenario is exepected to be very similar to run-times with a single implementation point.

In the future, we plan to extend these concepts to systems where HW and SW execute concurrently, i.e, consider scheduling issues as part of the problem formulation. Another interesting direction would be to extend the cost function concepts developed here to algorithms with fewer tunable parameters. While Simulated Annealing is a very powerful vehicle, our learning experience of individually tuning a lot of different parameters in SA confirms a need for heuristics in HW-SW partitioning that are *more deterministic*, yet capable of rapidly generating solutions of a similar high quality as our approach by exploiting the power of random moves in a similar controlled manner. One possible starting point for such explorations could possibly be an investigation of the WalkSAT SAT solver for hypergraph partitioning [2] that essentially does a KLFM with probabilistic move selection.

8 Acknowledgements

This work was partially supported by NSF Grants CCR-0203813 and CCR-0205712.

References

[1] M L Vallejo, J C Lopez, ”On the hardware-software partitioning problem: System Modeling and partitioning techniques”, ACM TODAES, V-8, 2003

[2] A Ramani, I Markov, ”Combining two local search approaches to hypergraph partitioning”, International Joint Conference on Artificial Intelligence, AAAI, 2003.

[3] T Wiangtong, P Cheung, W Luk, ”Comparing three heuristic search methods for functional partitioning in hardware-software codesign”, Jrnl Design Automation for Embedded Systems, V-6, 2002

[4] K Ben Chehida, M Auguin, ”HW/SW partitioning approach for reconfigurable system design”, CASES 2002

[5] J Henkel, R Ernst, ”An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation Techniques”, IEEE Trans.on VLSI, V-9, 2001

[6] K S Chatha, R Vemuri, ”Magellan: Multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs”, CODES 2001

[7] J Henkel, R Ernst, ”A hardware/software partitioner using a dynamically determined granularity”, DAC 1997

[8] F Vahid, T D Le, ”Extending the Kernighan-Lin heuristic for Hardware and Software functional partitioning”, Jrnl Design Automation for Embedded Systems, V-2, 1997

[9] P Eles, Z Peng, K Kuchinski, Doboli, ”System Level Hardware/Software Partitioning based on simulated annealing and Tabu Search”, Jrnl Design Automation for Embedded Systems, V-2, 1997

[10] F Vahid, D Gajski, ”Incremental hardware estimation during hardware/software functional partitioning”, IEEE Trans. VLSI, V-3, 1995

[11] F Vahid, J Gong, D Gajski, ”A binary-constraint search algorithm for minimizing hardware during hardware-software partitioning”, EDAC 1994

[12] A Kalavade, E Lee, ”A global criticality/Local Phase Driven algorithm for the Constrained Hardware/Software partitioning problem”, CODES 1994

[13] R Ernst, J Henkel, T Benner, ”Hardware-software cosynthesis for microcontrollers”, IEEE Design and Test,V-10, Dec 1993

[14] R Gupta, De Micheli, ”System-level synthesis using re-programmable components”, EDAC 92

[15] R P Dick, D L Rhodes, W Wolf, ”TGFF: task graphs for free”, CODES 1998

[16] M Lundy, A Mees, ”Convergence of an annealing algorithm”, Mathematical Programming, V-34, 1986

[17] C Sechen, A Sangiovanni-Vincentelli, ”The Timberwolf Placement and Routing Package”, IEEE Jrnl Solid-State Circuits, V-20, 1985

[18] D G Luenberger, ”Linear and non-Linear programming”, Addison-Wesley, 1984.

[19] F Romeo, A Sangiovanni-Vincentelli, ”Probabilistic hill-climbing algorithms: properties and applications”, ERL, College of Engineering, UC Berkeley, 1984

[20] S Kirkpatrick, C D Gelatt, M P Vechi, ”Optimization by simulated annealing”, Science, V-220, 1983

[21] C M Fiduccia, R M Mattheyes, ”A Linear-time heuristic for improving network partitions”, DAC, 1982

[22] B Kernighan, S Lin, ”An efficient heuristic procedure for partitioning graphs”, The Bell System Technical Journal, V-29, 1970

9 Appendix A: Aggregate data for KLFM Vs SA (proposed cost function)

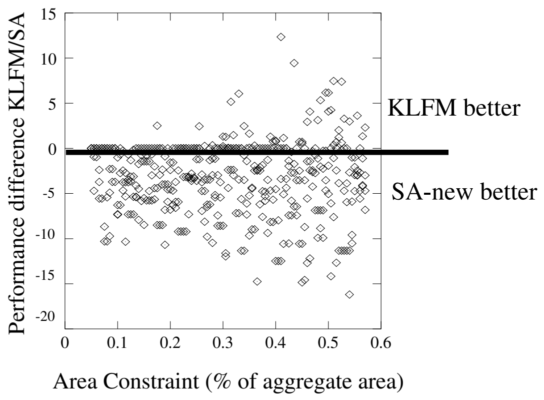

Figure 13. v20, CCR 0.1: Performance Vs constraint

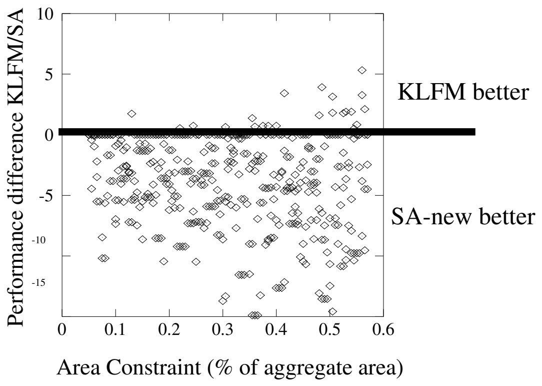

Figure 14. v20, CCR 0.3: Performance Vs constraint

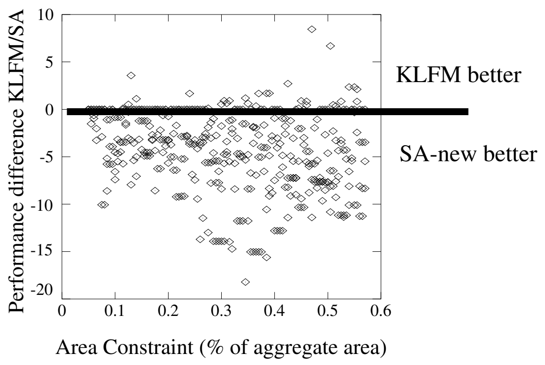

Figure 15. v20, CCR 0.5: Performance Vs constraint

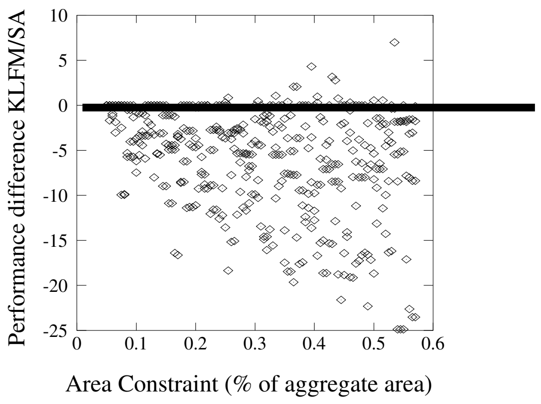

Figure 16. v20, CCR 0.7: Performance Vs constraint

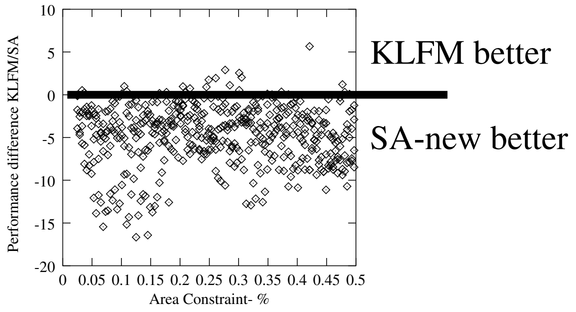

Figure 17. v100, CCR 0.1: Performance Vs constraint

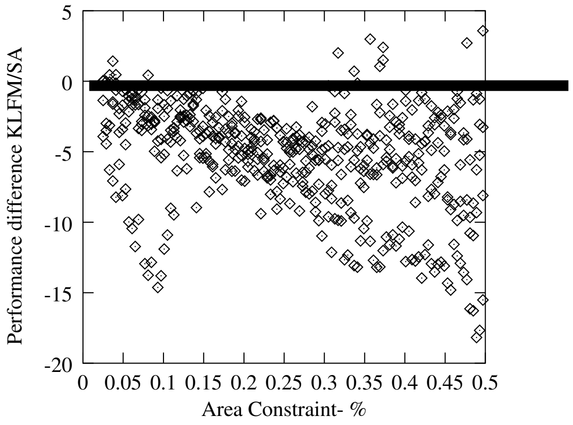

Figure 18. v100, CCR 0.3: Performance Vs constraint

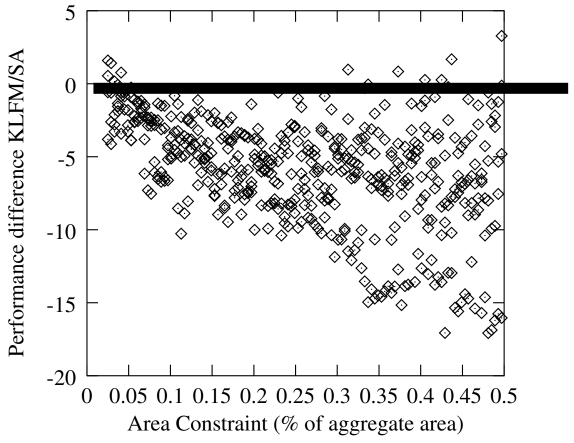

Figure 19. v100, CCR 0.5: Performance Vs constraint

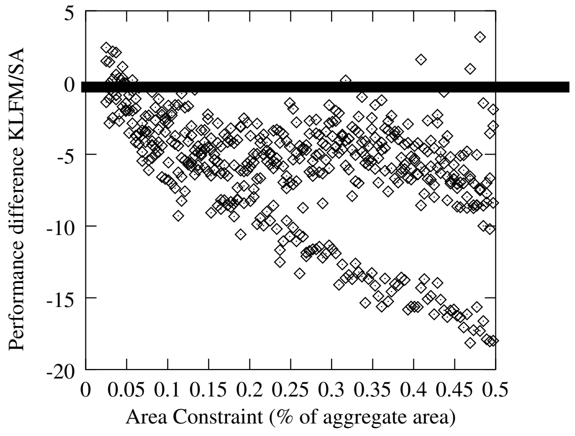

Figure 20. v100, CCR 0.7: Performance Vs constraint

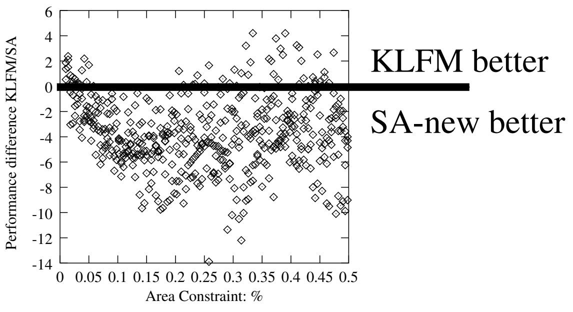

Figure 21. v200, CCR 0.1: Performance Vs constraint

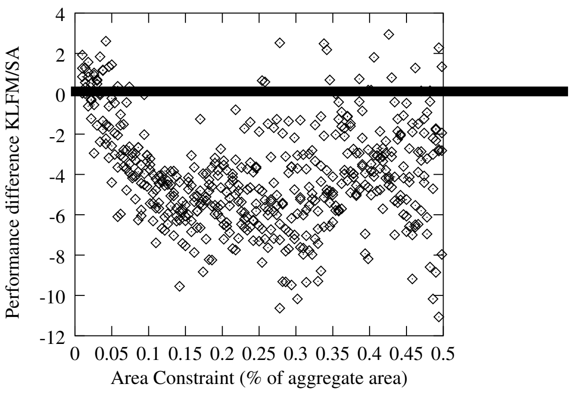

Figure 22. v200, CCR 0.3: Performance Vs constraint

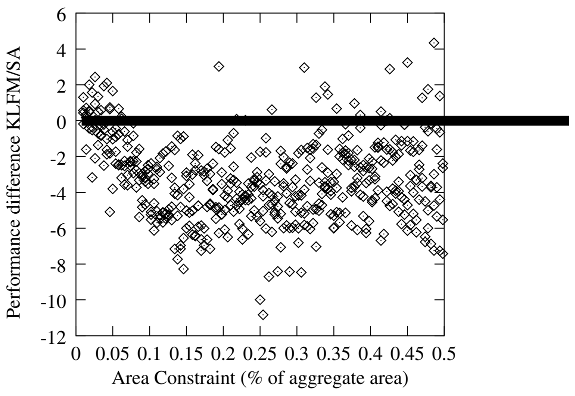

Figure 23. v200, CCR 0.5: Performance Vs constraint

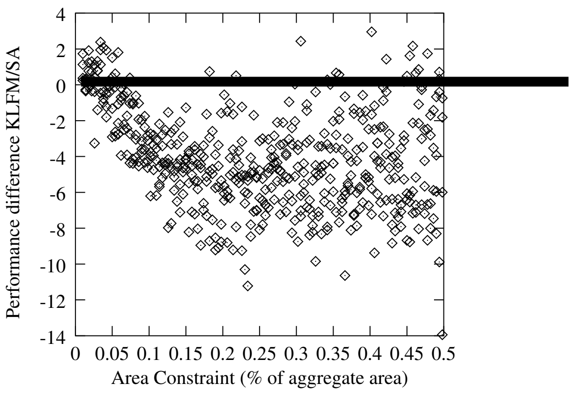

Figure 24. v200, CCR 0.7: Performance Vs constraint

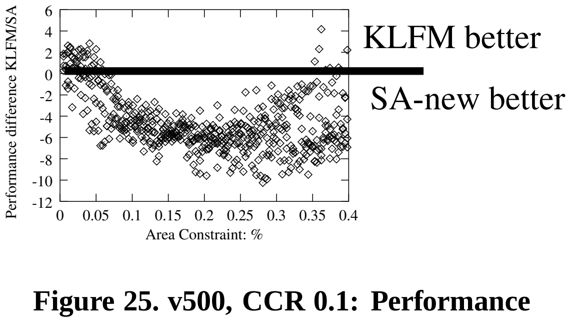

Figure 25. v500, CCR 0.1: Performance Vs constraint

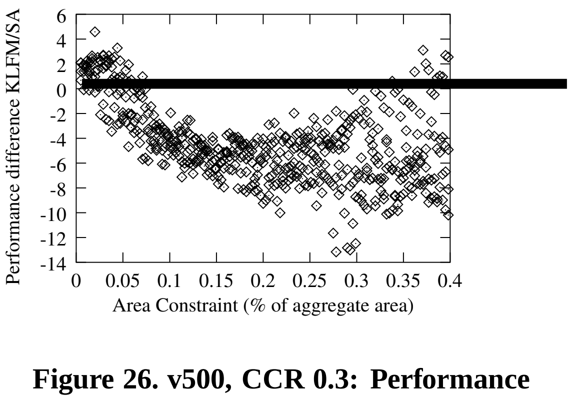

Figure 26. v500, CCR 0.3: Performance Vs constraint

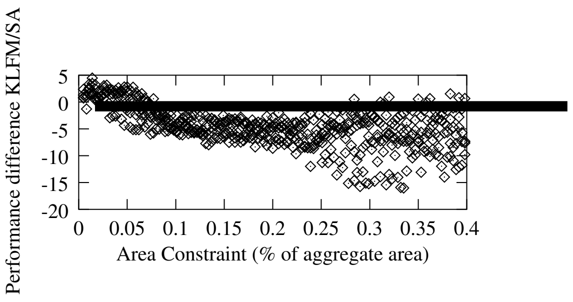

Figure 27. v500, CCR 0.5: Performance Vs constraint

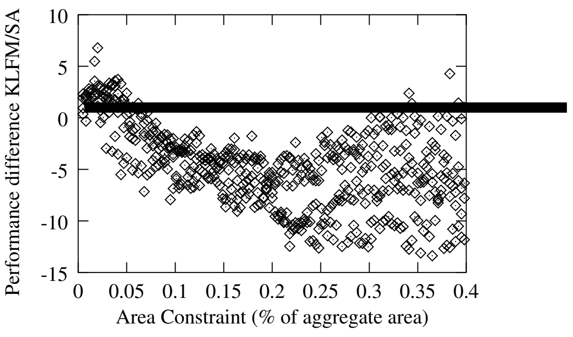

Figure 28. v500, CCR 0.7: Performance Vs constraint

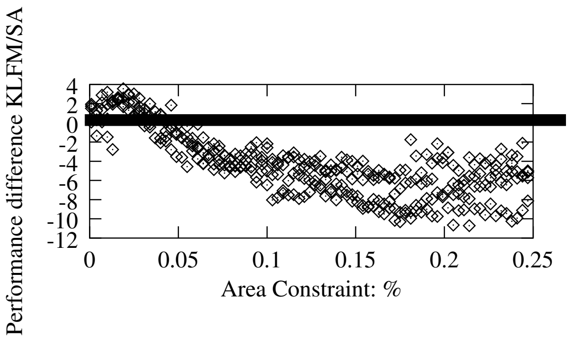

Figure 29. v1000, CCR 0.1: Performance Vs constraint

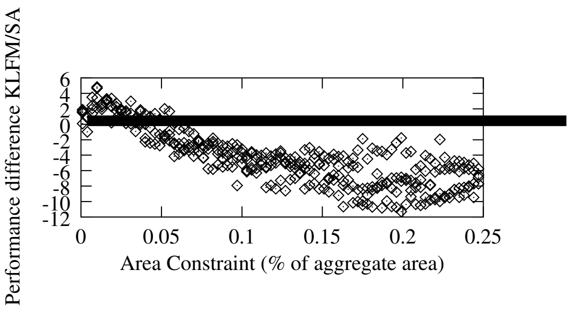

Figure 30. v1000, CCR 0.3: Performance Vs constraint

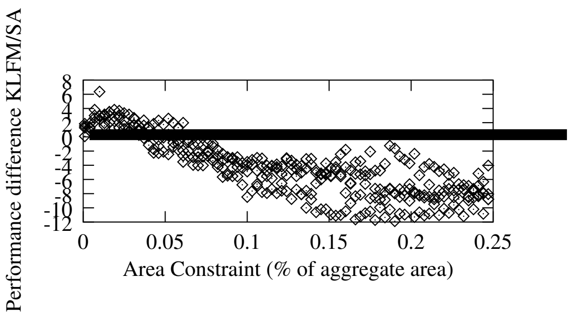

Figure 31. v1000, CCR 0.5: Performance Vs constraint

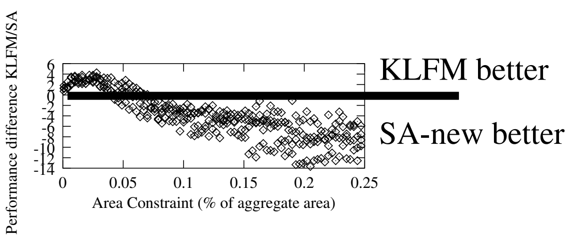

Figure 32. v1000, CCR 0.7: Performance Vs constraint