A Hardware-Software Partitioning Algorithm for Minimizing Hardware

Frank Vahid
Jie Gong
Daniel D. Gajski

Technical Report ICS-93-38
September 18, 1993

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 856-8059

vahid@ics.uci.edu
jgong@ics.uci.edu

Abstract

Partitioning a system's functionality among interacting hardware and software components is an important part of system design. We introduce a new partitioning algorithm that caters to the main objective of the hardware/software partitioning problem, i.e. minimizing hardware for given performance constraints. We demonstrate results superior to those of previously published algorithms intended for hardware/software partitioning.

Contents

1 Introduction **2**

2 Problem Definition **4**

3 Solving the hardware/software partitioning problem **5**
3.1 Greedy algorithms 5
3.2 Hill-climbing algorithms 6
3.3 A new algorithm based on constraint-search 8
3.3.1 Foundation 8
3.3.2 Algorithm 10
3.3.3 Reducing runtime in practice 10
3.3.4 Complexity 11

4 Experimental results **12**

5 Conclusion **16**

List of Figures

1 Basics parts of a hw/sw partitioning system 3
2 An example cost sequence 9
3 Performance constraints imposed on different examples 13
4 Hardware required for given performance constraints 14
5 Results of Hardware-software Partition 15

1 Introduction

Combined hardware/software implementations are common in embedded systems. Software running on an existing processor is cheaper, more modifiable, and more quickly designable than an equivalent application-specific hardware implementation. However, hardware may provide better performance. A system designer's goal is to implement a system using the minimal amount of application-specific hardware, if any at all, to achieve the performance required by the system's environment. In other words, the designer attempts to implement as much functionality as possible in software.

Deficiencies of the much practiced ad hoc approach to partitioning have led to research into more formal, algorithmic approaches. In the ad hoc approach, a designer starts with an informal functional description of the desired system, such as an English description. Based on previous experience and mental estimations, the designer partitions the functionality among hardware and software components, and the components are then designed and integrated. This approach has two key limitations. First, due to limited time, the designer can only consider a small number of possible partitionings, so many good solutions will never be considered. Second, the effects that partitioning has on performance are far too complex for a designer to accurately estimate mentally. As a result of these limitations, designers often use more hardware than necessary to ensure performance constraints are met.

In formal approaches, one starts with a *functional description* of the system in a machine-readable language, such as VHDL. After verifying, usually through simulation, that the description is correct, the functionality is decomposed into functional portions of some granularity. These portions, along with additional information such as data shared between portions, make up an internal *model* of the system. Each portion is mapped to either hardware or software by *partitioning algorithms* that search large numbers of solutions. Such algorithms are guided by automated *estimators* that evaluate *cost functions* for each partitioning. The output is a set of functional portions to be mapped to software and another set to be mapped to hardware. Simulation of the designed hardware and compiled software can then be performed to observe the effects of partitioning. Figure 1 shows a typical configuration of a hardware/software partitioning system.

The partitioning algorithm is a crucial part of the formal approach because it is the algorithm that actually minimizes the expensive hardware. However, current research into algorithms for hardware/software partitioning is at an early stage. In [1], the essential criteria to consider during partitioning are described, but no particular algorithm is given.

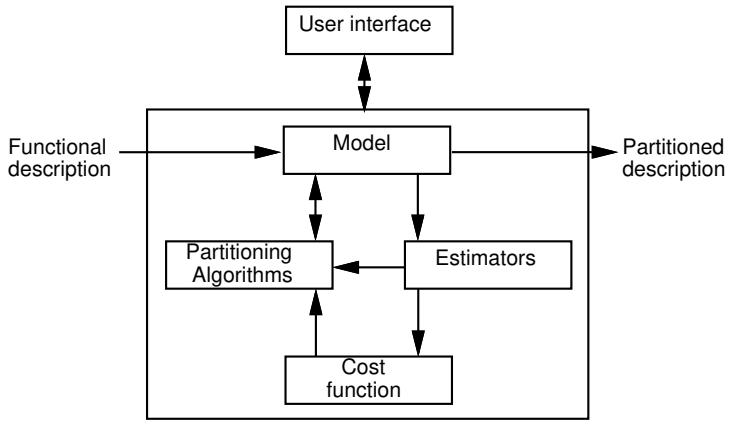

Figure 1: Basics parts of a hw/sw partitioning system

In [2], certain partitioning issues are also described but no algorithm is given. In [3], an algorithm is described which starts with most functionality in hardware, and which then moves portions into software as long as such a move improves the design. The algorithm uses a greedy control strategy which, while simple to implement and fast, is unable to escape local minimums so will likely result in more hardware than necessary. In [4], an approach is described which starts with all functionality in software, and which then moves portions into hardware using a hill-climbing algorithm such as simulated annealing. The authors hypothesize that starting with an all-software partitioning will result in less final hardware than starting with all-hardware partitioning (our results support this hypothesis). However, the partitioning algorithm does not actually attempt to minimize hardware; in fact, the designer must preselect a set of hardware functional units. In [5], simulated annealing is also used. The focus is on placing highly utilized functional portions in hardware. However, there is no direct performance measurement and no specific techniques described for hardware minimization.

In summary, the common shortcoming of previous approaches is the lack of advanced methods to minimize hardware. We propose a new algorithm which specifically addresses the unique characteristics of hardware/software partitioning, namely that hardware size should be minimized while performance constraints are met. The algorithm separates to a degree the problem of selecting a hardware size from the problem of partitioning to meet size and performance constraints. By incorporating existing hill-climbing algorithms rather than developing a custom control strategy, we are able to build on the sophisticated control techniques already developed to escape local minimums.

The paper is organized as follows. Section 2 gives a precise definition of the hardware/software partitioning problem. Section 3 describes previous hardware/software partitioning algorithms, an extension we have made to one previous algorithm to reduce hardware, and our new hardware-minimizing algorithm based on constraint-search. Section 4 summarizes our experimental results.

2 Problem Definition

While the partitioning subproblem interacts with other subproblems in hardware/software codesign, it is distinct, i.e. it is orthogonal to the choice of specification language, the level of granularity of functional decomposition, and the specific estimation models employed.

We are given a set of functions $F = f_1, f_2, ..., f_n$ which compose the functionality of the system under design. The functions may be at any of various levels of granularity, such as tasks (e.g. processes, procedures or code groupings) or arithmetic operations. We are also given a set of performance constraints $C = \{C_1, C_2, ..., C_m\}$, where $C_i = \{G_i, V_i\}, G_i \subset F, V_i \in Real$. V_i is a constraint on the maximum execution-time of the all functions in group G_i. It is simple to extend the problem to allow other performance constraints such as those on bitrates or inter-operation delays, but we have not included such constraints in order to simplify the notation.

A **hardware/software partitioning** is defined as a partition $P = \{H, S\}$, $H \subset F$, $S \subset F$, $H \bigcup S = F$ and $H \bigcap S = \emptyset$. This definition does not prevent further partition of hardware or software. Hardware can be partitioned into several chips while software can be executed on more than one processor. The **hardware size** of H is defined as the size (e.g. transistors) of the hardware needed to implement the functions in H. The **performance** of G_i is defined as the total execution time for the group of functions in G_i for a given partitioning P. A **performance satisfying** partitioning is one for which $Performance(G_i) \leq V_i$ for all $i = 1 \ldots m$.

Definition 1: Given F and C, the **Hardware/Software Partitioning Problem** is to find a performance satisfying partitioning P such that $HardwareSize(H)$ is minimal. In other words, the problem is to map all the functions to either hardware or software in such a way that we find the minimal hardware for which all performance constraints can still be met. Note that the hardware/software partitioning problem, like other partitioning problems, is NP-complete.

The **all-hardware size** of F is defined as the size of an all-hardware partitioning, i.e. $HardwareSize(F)$. Note that if an all-hardware partitioning does not satisfy performance constraints, no solution exists.

To compare any two partitionings, a cost function is required. A **cost function** is a function $Cost(P, C, I)$ which returns a natural number that summarizes the overall goodness of a given partitioning, the smaller the better. I contains any additional information that is not contained in P or C. We define an **iterative improvement partitioning algorithm** as a procedure $PartAlg(P, C, I, Cost())$ which returns a partitioning P' such that $Cost(P', C, I) \leq Cost(P, C, I)$. Examples of such algorithms include group migration [6] and simulated annealing [7].

Since it is not feasible to implement the hardware and software components in order to determine a cost for each possible partitioning generated by an algorithm, we assume that fast estimators are available [8, 9, 10].

3 Solving the hardware/software partitioning problem

3.1 Greedy algorithms

One simple and fast algorithm starts with an initial partition, and moves objects as long as improvement occurs. Such an algorithm is shown below. It uses a procedure $Move(P, f_i)$ which returns a new partitioning P' obtained by moving f_i to S if it is currently in H, or to H if it is currently in S.

Algorithm 3.1 Greedy1

```
Create initial partitioning P
repeat
    for each f_i ∈ F
        if Cost(Move(P, f_i), C, I) < Cost(P, C, I)
            P = Move(P, f_i)
until no moves improve cost
```

The greedy algorithm presented in [3], due to Gupta and DeMicheli, can be viewed as an extension of this algorithm which ensures performance constraints are met. The algorithm uses a procedure $Successors(f_i)$ which returns a set of functions that succeed f_i in the

internal model of the system's functionality.

Algorithm 3.2 Gupta/DeMicheli algorithm

Create initial partitioning $P = \{F, \emptyset\}$
repeat
  for each $f_i \in H$
    $AttemptMove(P, f_i)$.
until no moves improve cost

procedure $AttemptMove(P, f_i)$
  if $Move(P, f_i)$ is performance satisfying and $Cost(Move(P, f_i), C, I) < Cost(P, C, I)$
    $P = Move(P, f_i)$
    for each $f_j \in Successors(f_i)$
      $AttemptMove(P, f_j)$

The algorithm starts by creating an all-hardware partitioning, thus guaranteeing that a performance satisfying partitioning is found if it exists (actually, certain functions which are considered unconstrainable are initially placed in software). To move a function requires not only cost improvement but also that all performance constraints still be satisfied (actually they require that maximum interfacing constraints between partitions be satisfied). Once a function is moved, the algorithm tries to move closely related functions before trying others.

Both of the greedy algorithms suffer from the limitation that they are easily trapped in a local minimum. As a simple example, consider an initial partitioning that is performance satisfying, in which two heavily communicating functions f_1 and f_2 are initially in hardware. Suppose that moving either f_1 or f_2 to software results in performance violations, but moving both f_1 and f_2 results in a performance satisfying partitioning. Neither of the above algorithms can find the latter solution because doing so requires accepting an intermediate, seemingly negative move of a single function.

3.2 Hill-climbing algorithms

To overcome the limitation of greedy algorithms, others have proposed using an existing hill-climbing algorithm such as simulated annealing. Such an algorithm accepts some number of negative moves in a manner that overcomes many local minimums. One simply creates an initial partitioning and applies the algorithm.

In [4], an approach is described by Ernst and Henkel that uses an all-software solution for the initial partitioning. A hill-climbing partitioning algorithm is then used to extract functions from software to hardware in order to meet performance. The authors reason that such extraction should result in less hardware than the Gupta/DeMicheli approach where functions are extracted in the other direction, i.e. from hardware to software.

Cost function

We now consider devising a cost function to be used by the hill-climbing partitioning algorithm. The difficultly lies in trying to balance the performance satisfiability and hardware minimization goals. The Gupta/DeMicheli approach avoids the problem by removing performance satisfiability from the cost function. The cost function is only used to evaluate partitionings that already satisfy the performance constraints. The algorithm simply rejects all partitionings that are not performance satisfying. We saw that this approach is easily trapped in a local minimum. The Ernst/Henkel approach avoids the problem by removing hardware size from the cost function, fixing it apriori. This approach has the limitation of requiring the designer to manually try numerous hardware sizes, reapplying partitioning for each, to try to find the smallest hardware size that yields a performance satisfying partition. (In fact, in their approach the designer actually must select the functional units apriori, which is very difficult without knowing which functions are to be implemented).

We propose a third solution. We use a cost function with two terms, one indicating the sum of all performance violations, the other the hardware size. The performance term is weighed very heavily to ensure that a performance satisfying solution is found, i.e. minimizing hardware is a secondary consideration. The cost function is:

$$Cost(P,C) = k_{perf} \times \sum_{i=1}^{m} Violation(C_i) + k_{area} \times HardwareSize(H)$$

where

$$Violation(C_i) = Performance(G_i) - V_i$$

if the difference is greater than 0, else $Violation(C_i) = 0$. Also, $k_{perf} >> k_{area}$, but k_{perf} should *not* be infinity, since then the algorithm could not distinguish a partitioning which almost meets constraints from one which greatly violates those constraints.

We refer to this solution as the PWHC (performance-weighted hill-climbing) algorithm. We shall see that it gives excellent results as compared to the Gupta/DeMicheli algorithm, but there is still room for improvement.

3.3 A new algorithm based on constraint-search

While incorporating performance and hardware size considerations in the same cost function, as in PWHC, tends to gives much better results than previous approaches, we have determined a superior approach for minimizing hardware. Our approach involves decoupling to a degree the problem of satisfying performance from the problem of minimizing hardware.

3.3.1 Foundation

The first step is to relax the cost function goal. Rather than minimizing size, we just wish to find any size below a given constraint C_{size}.

$$Cost(P, C, C_{size}) = k_{perf} \times \sum_{i=1}^{m} Violation(C_i) + k_{area} \times Violation(HardwareSize(H), C_{size})$$

It is no longer required that $k_{perf} >> k_{area}$. We set $k_{perf} = k_{area} = 1$.

Hardware minimization can thus be stated as a problem which is distinct from the partitioning problem.

Definition 2: Given F, C, $PartAlg()$ and $Cost()$, the **Minimal Hardware-Constraint Problem** is to determine the smallest C_{size} such that $Cost(PartAlg(P, C, C_{size}, Cost()), C, C_{size}) = 0$.

In other words, we must choose the smallest size constraint for which a performance satisfiable solution can be found.

Theorem 1: Let $PartAlg()$ be such that it always finds a zero-cost solution if one exists. Then $Cost(PartAlg(P, C, C_{size}, Cost()), C, C_{size}) = 0$ implies that $Cost(PartAlg(P, C, C_{size} + 1, Cost()), C, C_{size} + 1) = 0$.

Proof: we can create a (hypothetical) algorithm which subtracts 1 from its hardware-size constraint if a zero-cost solution is not found. Given $C_{size} + 1$ as the constraint, then if a zero-cost is not found, the algorithm will try C_{size} as the constraint. Thus the algorithm can always find a zero-cost solution for $C_{size} + 1$ if one exists for C_{size}.

The above theorem states that if a zero-cost solution is found for a given C_{size}, then zero-cost solutions will be found for all larger values of C_{size} also. From this theorem we see that the sequence of cost numbers obtained for $C_{size} = 0, 1, ..., AllHardwareSize$ consists of x non-zero numbers followed by $C_{size} - x$ zero's, where $x \in \{0..AllHardwareSize\}$. Let $CostSequence$ equal this sequence of cost numbers. Figure 2 depicts an example of a

$CostSequence$ graphically. We can now restate the minimal hardware-constraint problem as a search problem:

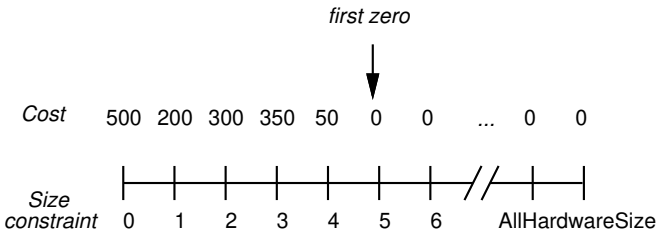

Figure 2: An example cost sequence

Definition 3: Given F, C, $PartAlg()$ and $Cost()$ the **Minimal Hardware-Constraint Search Problem** is to find the first zero in $CostSequence$.

Given this definition, we see that the problem can be easily mapped to the well-known problem of *Sorted-array search*, i.e. finding the first occurrence of a key in an ordered array of items. The main difference between the two problems is that whereas in sorted-array search the items exist in the array beforehand, in our constraint-search problem an item is added (i.e. partitioning applied and a cost determined) only if the array location is visited during search. In either case, we wish to visit as few array items as possible, so the difference does not affect the solution. A second difference is that the first x items in $CostSequence$ are not necessarily in increasing or decreasing order. Since we are looking for a zero cost solution, we don't care what those non-zero values are, so we can convert $CostSequence$ to an ordered sequence by mapping each non-zero cost to 1.

The constraint corresponding to the first zero cost represents the minimal hardware, or optimal solution to the partitioning problem. Due to the NP-completeness of partitioning, it should come as no surprise that we can not actually guarantee an optimal solution. Note that we assumed in the above theorem that $PartAlg()$ finds a zero-cost solution if one exists for the given size constraint. Since partitioning is NP-complete, such an algorithm is impractical. Thus $PartAlg()$ may not find a zero-cost solution although one may exist for a given size constraint. The result is that the first zero in $CostSequence$ may be for a constraint which is larger than the optimal, or that non-zero costs may appear for constraints larger than that yielding the first zero cost, i.e. the sequence of zeros contains spikes. However, the first zero cost should correspond to a constraint *near* the optimal if a good algorithm is used. In addition, any spikes that occur should also only appear near the optimal. Thus the algorithm should yield near optimal results, but of course this can not

be guaranteed.

It is well-known that binary-search is a good solution to the sorted-array search problem, since it's worst case behavior is $log(N)$ for an array of N items. We therefore incorporate binary-search into our algorithm.

3.3.2 Algorithm

We now describe our hardware-minimizing partitioning algorithm based on binary-search of the sequence of costs for the range of possible hardware constraints, which we refer to as the BCS (binary constraint-search) algorithm. The algorithm uses variables low and $high$ which indicate the current window of possible constraints in which a zero-cost constraint lies, and variable mid which represents the middle of that window. Another variable, P_{zero}, stores the zero-cost partitioning which has the smallest hardware constraint so far encountered.

Algorithm 3.3 Binary constraint-search (BCS) hw/sw partitioning

$low = 0, high = AllHardwareSize$
while $low < high$
  $mid = \frac{low+high+1}{2}$
  $P' = PartAlg(P, C, mid, Cost())$
  if $Cost(P', C, mid) = 0$
    $high = mid - 1$
    $P_{zero} = P'$
  else
    $low = mid$
return P_{zero}

The algorithm performs a binary search through the range of possible constraints, applying partitioning and then the cost function as each constraint is "visited". The algorithm looks very much like a standard binary-search algorithm with two modifications. First, mid is used as a hardware constraint for partitioning whose result is then used to determine a cost, in contrast to using mid as an index to an array item. Second, the cost is compared to 0, in contrast to an array item being compared to a key.

3.3.3 Reducing runtime in practice

After experimentation, we developed a simple modification of the constraint-search algorithm to reduce its runtime in practice. Let $size_{best}$ be the smallest zero-cost hardware

constraint. If a C_{size} constraint much larger than $size_{best}$ is provided to $PartAlg()$, the algorithm usually finds a solution very quickly. The functions causing a performance violation are simply moved to hardware. If a C_{size} constraint much smaller than $size_{best}$ is provided, the algorithm also stops fairly quickly, since it is unable to find a sequence of moves that improves the cost. However, if C_{size} is slightly smaller or larger than $size_{best}$, the algorithm usually makes a large number of moves, gradually inching its way towards a cost of zero. This situation is very different from traditional binary-search where a comparison of the key with an item takes the same time for any item. Near the end of binary-search the window of possible constraint values is very small, with $size_{best}$ somewhere inside this window. Much of the constraint-search algorithm's runtime is spent reducing the window size by minute amounts and reapplying lengthy partitioning.

In practice, we need not find the smallest hardware size to such a degree of accuracy. We thus terminate the binary-search when the window size (i.e. $high - low$) is less than a certain percentage of AllHardwareSize. This percentage is called an **accuracy factor**. We have found that an accuracy factor of 1% achieves a speedup of roughly 2.5. In our implementation, the user can select a larger or smaller accuracy factor to tradeoff runtime with accuracy.

3.3.4 Complexity

The worst-case runtime complexity of the constraint-search algorithm equals the complexity of the chosen partitioning algorithm $PartAlg()$ multiplied by the complexity of our binary constraint-search. While the complexity of the binary search of a sequence with $AllHardwareSize$ items is $log_2(AllHardwareSize)$, the accuracy factor reduces this to a constant.

Theorem 2: The complexity of the binary search of an N element $CostSequence$ with an accuracy factor a is $log_2(\frac{1}{a})$.

Proof: We start with a window size of N, and repeatedly divide the window size by 2 until the window size equals $a \times N$. Let w be the number of windows generated; w will thus give us the complexity. An equivalent value for w is obtained by starting with a window size of $a \times N$, and multiplying the size by 2 until the size is N. Hence we obtain the following equation: $(a \times N) \times 2^w = N$. Solving for w yields $w = log_2(\frac{N}{a \times N}) = log_2(\frac{1}{a})$. The complexity is therefore $log_2(\frac{1}{a})$.

We see that binary constraint-search partitioning with an accuracy factor has the same theoretical complexity as the partitioning algorithm $PartAlg()$. In practice, the binary

constraint-search contributes a small constant factor. For example, an accuracy factor of 1% results in a constant factor of $log_2(100) = 7$.

4 Experimental results

We briefly describe the environment in which we compared the various algorithms. It is important to note that most environment issues are orthogonal to the issue of algorithm design. Our algorithms should perform well in any of the environments discussed in other work such as [1, 2, 3, 4]. It should also be noted that any partitioning algorithm can be used within the BCS algorithm, not just simulated annealing.

We take a VHDL behavioral description as input. The description is decomposed to the granularity of tasks, i.e. processes, procedures, and optionally to statement blocks such as loops. Large data items (variables) are also treated as functions. Estimators of hardware size and behavior execution-time for both hardware and software are available [8, 9]. These estimators are especially designed to be used in conjunction with partitioning. In particular, very fast and accurate estimations are made available through special techniques to incrementally modify an estimate when a function is moved, rather than reestimating entirely for the new partitioning. The briefness of our discussion on estimation does not imply that it is trivial or simple, but instead that it is a different issue not discussed in this paper.

We implemented three partitioning algorithms: the Gupta/DeMicheli algorithm in Section 3.1 (abbreviated as Gupta), the PWHC algorithm in Section 3.2, and the BCS algorithm in Section 3.3. The $PartAlg()$ used in PWHC and BCS is group migration [6]. The accuracy factor used in BCS is 1%. We applied each algorithm to several examples: a real-time medical system (Volume) for measuring the patient's urinary bladder volume [8], and three industrial designs including a beam former system (Beam), a fuzzy logic control system (Fuzzy), and a microwave transmitter system (Microwave). For each example, a variety of performance constraints were input. Some examples have performance constraints on one group of tasks in the system. Others have performance constraints on two groups of tasks in the system. We tested each example using a set of performance constraints that reside between time required for all-hardware partitioning and time required for all-software partitioning.

The hardware area required for the four examples under different performance constraints by the three partitioning algorithms is shown in Figure 4. The horizontal axis in

Example \ Constraint	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Beam	2000	3000	4000	5000	6000	8000	10000	12000	14000	16000
Fuzzy	3000	5000	7000	9000	11000	13000	15000	30000	60000	120000
Microwave	90, 500	100, 500	200, 1000	1000, 500	2000, 1000	90, 3000	100, 4000	1000, 3000	200, 7000	2000, 7000
Volume	30, 100	90, 100	70, 300	10, 300	50, 500	70, 500	30, 700	10, 900	50, 900	90, 700

Figure 3: Performance constraints imposed on different examples

each graph represents different performance constraints (i.e. $C1 \ldots C10$). The details of the performance constraints imposed on each example are shown in Figure 3. The unit used for performance is clock cycle. The vertical axis represents the hardware area required for given performance constraints in the unit of $micron^2$. The initial partitioning used in each trial by PWHC and BCS is an all-software partitioning. We also ran PWHC and BCS starting with an all-hardware partitioning, but found that the results are inferior to those starting with an all-software partitioning.

Figure 5 summarizes the experimental results. The *run time* includes the algorithm's computation time and time for building the estimation information. The *percent hardware excess* represents the difference in hardware obtained by an algorithm as compared to the minimum hardware found for the given performance constraint. Results show that the Gupta algorithm is fast. The run time required for the BCS algorithm, in the worst case, is about eight times longer than the Gupta algorithm. However, with a longer but still practical run time, the BCS algorithm improves the partitioning results tremendously. For certain performance constraints, the partitions found by the Gupta algorithm requires $129.10 - 2739.45\%$ more hardware than those found by the BCS algorithm. And the partitions found by the PWHC algorithm requires $0.0 - 1474.80\%$ more hardware than those found by the BCS algorithm. For all performance constraints examined in each example, the average hardware required by the partitions found by Gupta algorithm is $30.16 - 170.21\%$ more than that required by BCS algorithm. And the average hardware required by the partitions found by PWHC algorithm is $0.0 - 38.74\%$ more than that required by BCS algorithm.

The most important fact to note is that the BCS algorithm always found the smallest hardware size among the three algorithms.

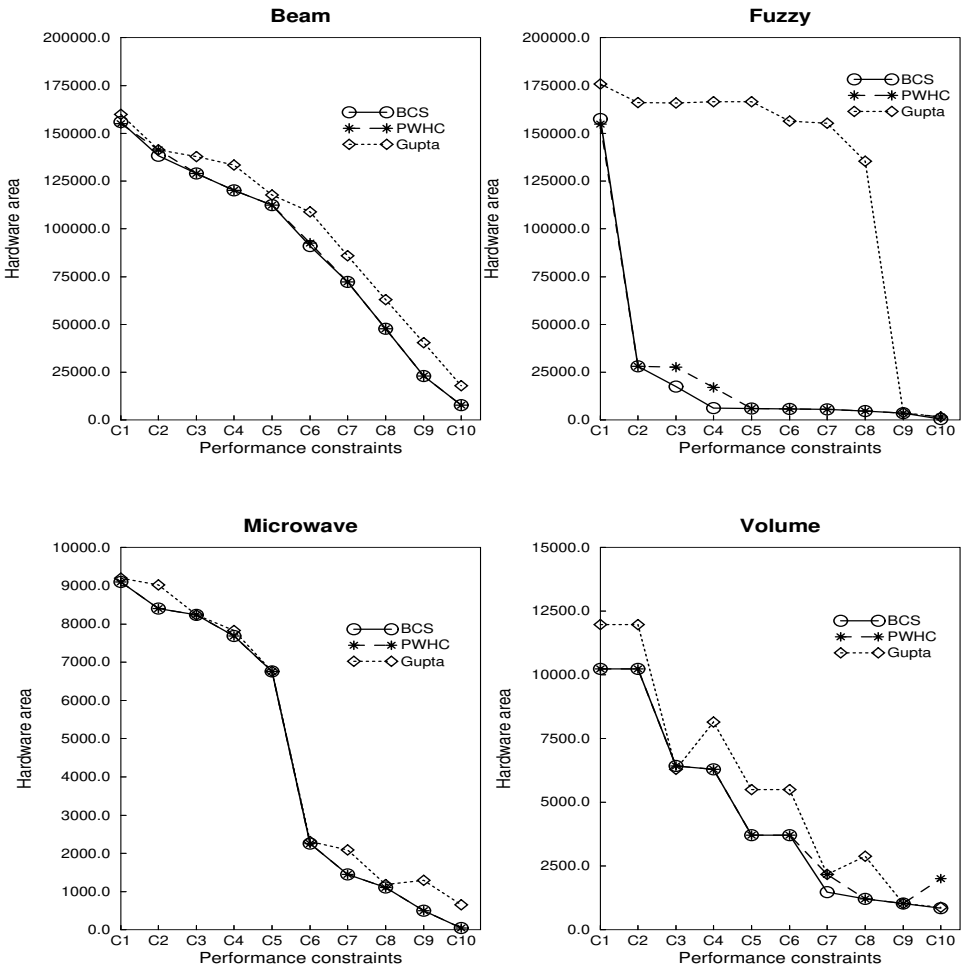

Figure 4: Hardware required for given performance constraints

Example	Number of spec. lines, tasks, and perf. constr.	Algorithms	Average run time (in seconds)	Average percent hw. excess	Maximum percent hw. excess
Beam	492, 49, 1	BCS	440.4	**0.0%**	**0.0%**
		PWHC	209.6	0.35%	2.23%
		Gupta	52.3	30.16%	129.10%
Fuzzy	292, 70, 1	BCS	453.2	**0.0%**	**0.0%**
		PWHC	264.8	38.74%	1474.80%
		Gupta	56.6	170.21%	2739.45%
Microwave	736, 29, 2	BCS	109.5	**0.0%**	**0.0%**
		PWHC	46.0	0.0%	0.0%
		Gupta	29.7	147.98%	1258.33%
Volume	208, 35, 2	BCS	114.8	**0.0%**	**0.0%**
		PWHC	44.5	18.41%	137.71%
		Gupta	13.1	34.67%	137.21%

Figure 5: Results of Hardware-software Partition

5 Conclusion

The constraint-search algorithm excels over previous hardware/software partitioning algorithms in its ability to find a minimal-hardware solution that satisfies performance constraints. The computation time required by the algorithm is practical, and can be greatly reduced if slightly less accuracy is required. The reduced hardware determined by our algorithm has the important benefits of reduced system cost, lower design time, and more easily modifiable final designs.

References

[1] D. Thomas, J. Adams, and H. Schmit, "A Model and Methodology for Hardware/Software Codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[2] A. Kalavade and E. Lee, "A Hardware/Software Codesign Methodology for DSP Applications," in *IEEE Design & Test of Computers*, 1993.

[3] R. Gupta and G. D. Micheli, "System-level Synthesis using Re-programmable Components," in *Proceedings of the European Conference on Design Automation*, pp. 2–7, 1992.

[4] R. Ernst and J. Henkel, "Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction," in *International Workshop on Hardware-Software Co-Design*, 1992.

[5] Z. Peng and K. Kuchcinski, "An Algorithm for Partitioning of Application Specific Systems," in *Proceedings of the European Conference on Design Automation*, pp. 316–321, 1993.

[6] B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems*. California: Benjamin/Cummings, 1988.

[7] S. Kirkpatrick, C. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[8] J. Gong, D. Gajski, and S. Narayan, "Software Estimation from Executable Specifications," in *Journal of Computer and Software Engineering*, to appear.

[9] S. Narayan and D. Gajski, "Area and Performance Estimation from System-Level Specifications." UC Irvine, Dept. of ICS, Technical Report 92-16,1992.

[10] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast Timing Analysis for Hardware-Software Co-Synthesis," in *Proceedings of the International Conference on Computer Design*, pp. 452–457, 1993.