Computing and Informatics, Vol. 23, 2004, 557–569

KNAPSACK MODEL AND ALGORITHM FOR HARDWARE/SOFTWARE PARTITIONING PROBLEM

Abhijit Ray, Wu Jigang, Thambipillai Srikanthan

Centre for High Performance Embedded Systems
School of Computer Engineering
Nanyang Technological University
Singapore,639798
Republic of Singapore
e-mail: {PA8760452, asjgwu, astsrikan}@ntu.edu.sg

Manuscript received 12 October 2004; revised 17 December 2004

Communicated by Hong Zhu

Abstract. Efficient hardware/software partitioning is crucial towards realizing optimal solutions for constraint driven embedded systems. The size of the total solution space is typically quite large for this problem. In this paper, we show that the knapsack model could be employed for the rapid identification of hardware components that provide for time efficient implementations. In particular, we propose a method to split the problem into standard 0-1 knapsack problems in order to leverage on the classical approaches. The proposed method relies on the tight lower and upper bounds for each of these knapsack problems for the rapid elimination of the sub-problems, which are guaranteed not to give optimal results. Experimental results show that, for problem sizes ranging from 30 to 3000, the optimal solution of the whole problem can be obtained by solving only 1 sub-problem except for one case where it required the solution of 3 sub-problems.

Keywords: Hardware/software partitioning, embedded systems, algorithm, knapsack problem

1 INTRODUCTION

Hardware/software partitioning (HSP) problem is the problem of deciding for each subsystem, whether the required functionality is to be implemented in hardware or software to get the desired performance in terms of running time and power while maintaining least cost. HSP is one of the crucial steps in embedded system design. Satisfaction of performance requirements for embedded systems can frequently be achieved only by hardware implementation of some parts of the application. Selection of the appropriate parts of the system for hardware and software implementation has a crucial impact both on the cost and overall performance of the final product. At the same time, hardware area minimization and latency constraints present contradictory objectives to be achieved through hardware/software partitioning.

Most of the existing approaches to HSP are based on either hardware-oriented partitioning or software-oriented partitioning. A *software-oriented* approach means that initially the whole application is allotted to software and during partitioning system parts are moved to hardware until constraints are met. On the other hand, in a *hardware-oriented* approach the whole application is implemented in hardware and during partitioning the parts are moved to software until constraints are violated. A software-oriented approach has been proposed by Ernst et al. [1], Vahid et al. [2]. Hardware-oriented approach has been proposed in Gupta et al. [3], Niemann et al. [4]. In [5], the authors proposed a flexible granularity approach for hardware software partitioning. Karam et al. [6], propose partitioning schemes for transformative applications, i.e. multimedia and digital signal processing applications. The authors try to optimize the number of pipeline stages and memory required for pipelining. The partitioning is done in an iterative manner. Rakhmatov et al. [7] modeled the hardware/software partitioning as a unconstrained bipartitioning problem. Cost functions were used to model the computation and the communication costs. The disadvantage in a hardware oriented partitioning is that the partitioning process is stopped as soon as constraints are met. This can easily result in a non-optimal solution. Similarly for software oriented partitioning the algorithm can stop in a local minimum.

Our proposed method is different in the sense that we do not start with a all hardware or all software solution; we rather model the partitioning problem as some standard knapsack problem, which can be solved independently to arrive at the solution. Also for every subproblem we calculate the lower and upper bounds, this helps in rejecting subproblems, which are not expected to give optimal results. Hence not all subproblems need to be solved. This paper is the first work to solve hardware/software partitioning problem using the knapsack problem. The advantage of our work is that it provides the optimal solution. Moreover, many problems are rejected based on their lower bound and upper bound, and this reduces the number of subproblems that need to be solved; hence the algorithm is quite fast.

The outline of this paper is as follows: In Section 2 we give the mathematical model of the hardware/software partitioning problem. In Section 3 we show how to split the partitioning problem into some standard independent 0-1 knapsack

problems, and then we describe the proposed algorithm. In Section 4 we give our proposed algorithm. In Section 5 we give our experimental results. In Section 6 we give the conclusion.

2 MODEL OF THE PHYSICAL PROBLEM

We consider a basic case which can be later extended. In our case the application can be broken down into parts such that each of them can be run simultaneously or in other words the parts do not have any sort of data dependency between them. So we have a set of items $S = \{p_1, p_2, \ldots, p_n\}$ to be partitioned into hardware and software. Let h_i and s_i be the time required for the part p_i to be run in hardware and software, respectively. Also let a_i be the area required for hardware implementation of part p_i, and let A be the total area available for hardware implementation. Our goal is to allot each part into hardware and software so that the combined running time of the whole application is minimized while the area constraint is satisfied. Let us denote the solution of the problems as a vector $X = [x_1, x_2, \ldots, x_n]$ such that $x_i \in \{0, 1\}$, where $x_i = 0\,(1)$ implies that the part p_i is implemented in software (hardware). Since the hardware and software can be run in parallel, the total running time of the application is given by

$$T(X) = max\{H(X), S(X)\} \tag{1}$$

where $H(X)$ is the total running time of the parts running in hardware and $S(X)$ is the total running time of the parts in software. Since all the parts that are implemented in hardware can be run in parallel to each other and all the software parts has to be run in serial, we have

$$H(X) = \max_{1\le i\le n}\{x_i \cdot h_i\} \quad \text{and} \quad S(X) = \sum_{i=1}^{n}(1 - x_i)\cdot s_i.$$

Hence, the problem discussed in this paper can be modeled into

$$\mathcal{P} \left\{ \begin{array}{ll} \text{minimize} & T(X) \\ \text{subject to} & \sum\limits_{i=1}^{n} x_i \cdot a_i \le A \end{array} \right. \tag{2}$$

3 PROBLEM SPLITTING

First of all, we review the knapsack problem. Given a knapsack capacity C and set of items $S = \{1, \ldots, n\}$, where each item has a weight w_i and a benefit b_i. The problem is to find a subset $S' \subset S$, that maximizes the total profit $\sum_{i\in S'} b_i$ under the constraint that $\sum_{i\in S'} w_i \le C$, i.e., all the items fit in a knapsack of carrying capacity C. This problem is called the knapsack problem. The 0-1 knapsack problem is a special case of the general knapsack problem defined above, where each

item can either be selected or not selected, but cannot be selected fractionally. Mathematically, it can be described as follows:

$$\text{0-1 KP}\begin{cases} \text{maximize} & \sum_{i=1}^{n} p_i \cdot x_i \\ \text{subject to} & \sum_{i=1}^{n} w_i \cdot x_i \leq C, \\ & x_i \in \{0,1\}, i = 1, \ldots n \end{cases} \tag{3}$$

where x_i is a binary variable equalling 1 if item i should be included in the knapsack and 0 otherwise. It is well known that this problem is NP-complete[8]. However, several large scaled instances could be solved optimally in fractions of a second in spite of the exponential worst-case solution time of all knapsack algorithms [8, 9, 10].

Let us assume that the items are ordered by their efficiencies in a non-increasing manner. We define the efficiency as

$$e_j = \frac{b_j}{w_j} \tag{4}$$

thus we have $e_i \geq e_j$ for all i, j such that $i < j$.

Let

$$\overline{b_j} = \sum_{i=1}^{j} b_i \quad j = 1, 2, \ldots, n \tag{5}$$

$$\overline{w_j} = \sum_{i=1}^{j} w_i \quad j = 1, 2, \ldots, n. \tag{6}$$

Packing a knapsack in a greedy way means to put the items in the decreasing order of their efficiency as long as $w_j \leq C - \overline{w}_{j-1}$ i.e., as long as the next item fits in the unused capacity of the knapsack. According to the definition given in [8], the break item is the first item, which cannot be included in the knapsack. Thus the break item t satisfies

$$\overline{w}_{t-1} \leq C \leq \overline{w}_t \tag{7}$$

Let the residual capacity r be defined as

$$r = C - \overline{w}_{t-1}. \tag{8}$$

By linear relaxation, [8] showed that an upper bound on the total benefit of 0-1 KP is

$$u = \lceil \overline{b}_{t-1} + r \cdot \frac{b_t}{w_t} \rceil \tag{9}$$

and the lower bound is given by,

$$l = \overline{b}_{t-1}. \tag{10}$$

In HSP problem, sort all the items $p_1, p_2, \ldots, p_n$ in decreasing order of their hardware running time. We are sorting so that if we allocate the item with the highest hardware running time, the running time of the whole hardware part is

fixed to its hardware running time. This is because all the hardware parts can be run in parallel, so that after sorting we have the items ordered as $p'_1, p'_1, \ldots, p'_n$ and let $h'_i(s'_i)$ be the time required for the part p'_i to be run in hardware(software). Thus, after sorting the following condition is satisfied

$$h'_i \geq h'_j \qquad \text{for all} \quad i \leq j \quad \text{and} \quad 0 \leq i, j \leq n. \tag{11}$$

Let us define $S_T = \sum_{i=1}^{n} s'_i$ and $R_i = S_T - s'_i$. Now we split the problem $\mathcal{P}$ into the following n subproblems $\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n$.

3.1 Subproblem $\mathcal{P}_1$

Let p'_1 be implemented in hardware, i.e., $x_1 = 1$, so the total time T that we have to minimize becomes

$$\begin{aligned} T(X) &= max\{h'_1, S(X)\} \\ &= max\left\{h'_1, \sum_{i=1}^{n}(1 - x_i)\cdot s'_i\right\} \\ &= max\left\{h'_1, S_T - \sum_{i=1}^{n} x_i \cdot s'_i\right\} \\ &= max\left\{h'_1, S_T - x_1 \cdot s'_1 - \sum_{i=2}^{n} x_i \cdot s'_i\right\} \\ &= max\left\{h'_1, S_T - s'_1 - \sum_{i=2}^{n} x_i \cdot s'_i\right\} \\ &= max\left\{h'_1, R_1 - \sum_{i=2}^{n} x_i \cdot s'_i\right\}. \end{aligned}$$

Our goal is to minimize the total running time $T(X)$, i.e.,

$$\begin{aligned} minimize \quad T(X) \quad &\Longleftrightarrow \quad minimize\left\{R_1 - \sum_{i=2}^{n} x_i \cdot s'_i\right\} \\ &\Longleftrightarrow \quad maximize\left\{\sum_{i=2}^{n} x_i \cdot s'_i\right\} \end{aligned}$$

subject to the constraint $x_i \in \{0, 1\}$ and the area constraint

$$\sum_{i=2}^{n} x_i \cdot a_i \leq A - a_1. \tag{12}$$

Formally, the subproblem $\mathcal{P}_1$ is described as

$$\mathcal{P}_1 \left\{ \begin{array}{ll} \text{maximize} & \sum_{i=2}^{n} x_i \cdot s'_i \\ \text{subject to} & \sum_{i=2}^{n} x_i \cdot a_i \leq A - a_1. \end{array} \right. \tag{13}$$

It is clear that $\mathcal{P}_1$ is the standard 0-1 knapsack problem, and the solution of $\mathcal{P}_1$ is a feasible solution of the problem $\mathcal{P}$. Let

$$L_1 = \max\{h'_1, R_1 - u_1\},$$

$$U_1 = \max\{h'_1, R_1 - l_1\},$$

where l_1 and u_1 are the lower bound and the upper bound on $\mathcal{P}_1$, respectively. We call $[L_1, U_1]$ the bounded interval of $\mathcal{P}_1$ in the sense that the optimal solution of $\mathcal{P}_1$ would lie in the range $[L_1, U_1]$.

Similarly we have subproblem $\mathcal{P}_k$ for $k > 1$.

3.2 Subproblem $\mathcal{P}_k$

We fix p'_k to be implemented in hardware i.e., $x_k = 1$, and all the items $1, 2, \ldots, k-1$ are in software, because if any of them, say j, is in hardware then any subproblem $\mathcal{P}_l$ such that $l > j$ is a subset of subproblem $\mathcal{P}_j$. That is, we have $x_1 = 0, x_2 = 0, \ldots, x_{k-1} = 0$. The total time is

$$T(X) \quad = \quad max\left\{h'_k, R_k - \sum_{i=k+1}^{n} x_i \cdot s'_i\right\}.$$

We have to minimize the total running time $T(X)$, i.e.,

$$\begin{aligned} minimize \quad T(X) \quad &\Longleftrightarrow \quad minimize\left\{R_k - \sum_{i=k+1}^{n} x_i \cdot s'_i\right\} \\ &\Longleftrightarrow \quad maximize\left\{\sum_{i=k+1}^{n} x_i \cdot s'_i\right\} \end{aligned}$$

subject to the constraint $x_i \in \{0, 1\}$ and the area constraint

$$\sum_{i=k+1}^{n} x_i \cdot a_i \leq A - a_k. \tag{14}$$

Formally, the subproblem $\mathcal{P}_k$ is described as

$$\mathcal{P}_k \left\{ \begin{array}{ll} \text{maximize} & \sum_{i=k+1}^{n} x_i \cdot s'_i \\ \text{subject to} & \sum_{i=k+1}^{n} x_i \cdot a_i \leq A - a_k. \end{array} \right. \tag{15}$$

The bounded intervals of subproblem $\mathcal{P}_k$ are

$$L_k = \max\{h'_k, R_k - u_k\},$$

$$U_k = \max\{h'_k, R_k - l_k\},$$

and the optimal solution of $\mathcal{P}_k$ would lie in the range $[L_k, U_k]$ where l_k and u_k are the lower bound and the upper bound of total benefit of $\mathcal{P}_k$, respectively.

Theorem 1. The optimal solution of $\mathcal{P}$ is the solution X_k of $\mathcal{P}_k$, which gives the maximum benefit amongst all the solutions of subproblems $\mathcal{P}_i$, $1 \le i \le n$.

To prove the above, let X_i be the solution of the subproblem $\mathcal{P}_i$, $i = 1, 2, \ldots, n$. We should prove that

1. Any X_i is a feasible solution of the problem.
2. Any optimal solution of the problem $\mathcal{P}$ belongs to $\{X_1, X_2, \ldots, X_n\}$.

Proof.

1. Each of $\mathcal{P}_i$, $i = 1, 2, \ldots, n$ is formed from the original problem $\mathcal{P}$ by fixing the part i as being included in the knapsack; for each $\mathcal{P}_i$, the capacity is $A_i = A - a_i$. Hence every optimal solution of $\mathcal{P}_i$ is a feasible solution of $\mathcal{P}$ with part i in the knapsack.
2. Let X be any feasible solution of $\mathcal{P}$, and let h_i be the part with the highest hardware running amongst the parts included in the knapsack. Since part i is included in the knapsack, it is one of the solutions of the subproblem $\mathcal{P}_i$.

□

Theoretically, we can create as many subproblems as there are items to be partitioned. However, a closer look at the derivations indicates that we need not create n subproblems if we have n items to be partitioned, for example, if after creating subproblem i, we find that

$$\sum_{j=i+1}^{n} a_j \le A - a_i. \tag{16}$$

This means that after we have fixed items a_i to be implemented into hardware and $a_1, a_2, \ldots, a_{i-1}$ into software, the rest of the items left to be partitioned can be implemented easily in hardware as there is enough hardware space left. In most of the cases, hardware runs faster than software and also the items in hardware can run in parallel. So we want to implement as much as possible into hardware. Thus, since enough space is available, we need not solve the subproblem to find a partition as all the remaining items can now be implemented into hardware. Hence, we can stop creating more subproblems as soon as equation (16) is satisfied.

A point to be noted is that all subproblems are not of the same size. This implies from the fact that for the first problem we have fixed item 1 to be implemented in

hardware and the rest of the items are not known whether they are in hardware or software. However, in the second problem we have fixed item 2 to be implemented in hardware and item 1 is no longer implemented in hardware. This is because if in the second problem item 1 is implemented in hardware then it becomes a subset of subproblem one. Similarly for subproblem i all the items $1, 2, \ldots, i-1$ are automatically fixed to be in software. Hence, the size of the problem decreases from subproblem 1 to n.

Similarly we can create subproblems for each of the items $p_1, p_2, \ldots, p_n$. The optimal solutions obtained for each of the subproblems will be different from each other, but the optimal solution for the whole problem is the optimal solution from only one of the subproblems. The best solution of all the subproblems in this case will be the optimal solution for the original problem.

4 ALGORITHM DESCRIPTION

Following is a brief description of the proposed algorithm. The first step is to sort all the items that are to be partitioned in decreasing order of their hardware running time. Then, create a subproblem is corresponding to each of the items to be partitioned, i.e., there are n subproblems to be solved. For each of the subproblems calculate the upper bound and lower bound on the benefits. Find the subproblem with the highest lower bound, $lb_{\max}$. All subproblems whose upper bound is smaller than $lb_{\max}$ is guaranteed to give a solution whose benefit is less than $lb_{\max}$. Hence all such subproblems can be ignored and need not be solved. The subproblem with the maximum lower bound should be solved first. Then the subproblems whose upper bound is smaller than the solution should be rejected. These steps should be repeated until all subproblems have been solved or rejected. The outline of the algorithm for solving the HSP problem is given below:

1: $BOUND := 0$;
2: sort all the items to be partitioned in decreasing order of
  their hardware running time;
3: form the subproblems $\mathcal{P}(i)$, $i = 1, 2, \ldots, n$;
4: **for** $(i := 1;\ i \leq n;\ i{+}{+}\ )\{$
5:   calculate the upper bound $U(i)$ and the lower bound $L(i)$ for $\mathcal{P}(i)$;
6:   **if** $(L(i) > BOUND)$ {
7: $BOUND := L(i)$;
8:   }
9: }
10: **while** (there are subproblems left to be solved) {
11:   select the subproblem with the highest lower bound;
12:   **if** $(U(i) < BOUND)$ {
13:   reject this subproblem;
14:   } **else** {
15:   solve this subproblem;

```
16:         B(i) :=benifit of the above solution;
17:         if (B(i) > BOUND){
18:             BOUND := B(i);
19:         }
20:     }
21: }
```

5 EXPERIMENTAL WORKS

The proposed algorithm was implemented on a Pentium, 500MHz system, running on Linux. We used random data for partitioning. For solving the individual 0-1 knapsack problems we used the algorithm for 0-1 knapsack problem, given in [9]. All results are the average to 1000 runs of the algorithm. The experiment was performed for different problem sizes and area constraints.

Table 1 shows the execution time of our algorithm for different problem sizes. In this table, the columns denote the different values for the area constraint. Specifically, we have used values 10, 20, ..., 90 percent of the total area required if all the items were to be implemented in hardware. The different rows are for different problem sizes. For example, for a problem size of 60 and an area constraint of 30 percent of total area required if all the 60 parts are to be implemented in hardware, we have a running time of 0.195 ms (from Table 1).

size	fraction of area put as constraint								
n	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
30	0.156	0.461	0.180	0.180	0.180	0.102	0.117	0.102	0.117
60	0.203	0.211	0.195	0.195	0.188	0.117	0.195	0.164	0.109
90	0.125	0.195	0.211	0.188	0.203	0.203	0.188	0.203	0.180
300	0.312	0.336	0.375	0.281	0.352	0.383	0.344	0.359	0.289
500	0.555	0.578	0.594	0.547	0.617	0.508	0.594	0.516	0.516
700	0.906	0.906	0.891	0.945	0.906	0.930	0.859	0.852	0.883
1000	1.688	1.852	1.695	1.633	1.625	1.688	1.758	1.680	1.672
2000	8.078	8.008	8.086	8.078	8.133	7.969	7.977	8.000	7.914
3000	22.102	22.172	22.406	22.141	22.688	22.203	22.195	22.281	22.242

Table 1. Running time (ms) of the algorithm for different problem sizes

Table 2 gives a count of the number of subproblems that needed to be solved to arrive at the optimal solution for the whole problem. For example, for a problem size of 30 and area constraint of 20 percent of total area required if all the 30 parts were to be implemented in hardware, the algorithm needed to solve 3 subproblems to arrive at the optimal solution. The result shows that we need to solve very few subproblems to arrive at the solution. In fact only for one instance we had to solve three subproblems to arrive at the solution, while the rest needed only one subproblem to be solved.

size	fraction of area put as constraint								
n	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
30	1	3	1	1	1	1	1	1	1
60	1	1	1	1	1	1	1	1	1
90	1	1	1	1	1	1	1	1	1
300	1	1	1	1	1	1	1	1	1
500	1	1	1	1	1	1	1	1	1
700	1	1	1	1	1	1	1	1	1
1000	1	1	1	1	1	1	1	1	1
2000	1	1	1	1	1	1	1	1	1
3000	1	1	1	1	1	1	1	1	1

Table 2. The number of subproblems solved for different problem size and area constraints

We also provide a plot of running time of the algorithm for the different problem sizes (Figure 1). We used the area constraint as 50 percent of the area required if all parts were to be implemented in hardware. Y-axis denotes the running time in milliseconds. X-axis denotes the problem size.

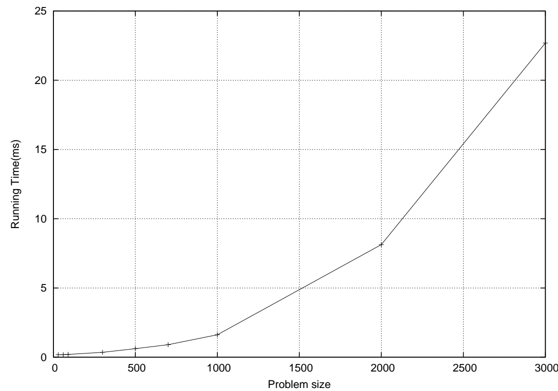

Fig. 1. Running time vs. Problem size

From the results, we can see that, for small problem size (e.g. 30, 60, 90), the execution time fluctuates. With increase in size the running time increases. This correctly reflects the property of the knapsack problem. This can also be seen from the graph (Figure 1). Also for the same problem size the execution time is stable for different area constraint. This is because the number of subproblems solved is 1 for almost all the cases (see Table 2). Hence, since the same numbers of same sized subproblems are being solved for all the cases, the execution time is similar.

6 CONCLUSION

We have proposed an efficient hardware/software partitioning technique based on the knapsack model. It was shown that by examining the upper and lower bounds of the subproblems, we could rapidly eliminate the large number of subproblems that do not contribute to optimal solutions. Our investigations demonstrate that a substantial reduction in the number of subproblems that require processing is possible, thereby providing for an efficient means to partitioning of hardware and software. This is of particular significance when the problem size is large. Our simulations based on problems size ranging from 30 to 3000 confirm that the number of subproblems require solution is one except for one case where it was increased to just 3 in order to compute the optimal solution. We are currently extending our knapsack model based approach to include communication overheads so as to represent a more accurate partitioning scheme.

Acknowledgement

A part of the work titled "Knapsack Model and Algorithm for HW/SW Partitioning Problem" appeared in the 4th *International Conference in Computational Sciences(ICCS), 2004, Lecture Notes in Conputer Science 3036, pp. 200–205.*

REFERENCES

[1] Ernst, R.—Henkel, J.—Benner, T.: Hardware-Software Cosynthesis for Microcontrollers. IEEE Design and Test of Computers, 1993, pp. 64–75.

[2] Vahid, F.—Gajski, D. D.—Jong, J.: A Binary-Constraint Search Algorithm for Minimizing Hardware During Hardware/Software Partitioning. IEEE/ACM Proceedings European Conference on Design Automation (EuroDAC), 1994, pp. 214–219.

[3] Gupta, R. K.—Micheli, G. D.: System-Level Synthesis Using Reprogrammable Components. Proceedings. [3rd] European Conference on Design Automation, 1992, pp. 2–7.

[4] Niemann, R.—Marwedel, P.: Hardware/Software Partitioning Using Integer Programming. Proceedings European Design and Test Conference, 1996. ED & TC 96, pp. 473–479.

[5] Henkel, J.—Ernst, R.: An Approach to Automated Hardware/Software Partitioning Using a Flexible Granularity That Is Driven by High-Level Estimation Technique. Very Large Scale Integration (VLSI) Systems, IEEE Transactions, Vol. 9, 2001, No. 2, pp. 273–289.

[6] Chatha, K. S.—Vemuri, R.: Hardware-Software Partitioning and Pipelined Scheduling of Transformative Applications. Very Large Scale Integration (VLSI) Systems, IEEE Transactions, Vol. 10, 2002, No. 3, pp. 193–208.

[7] Rakhmatov, D. N.—Vrudhula, S. B. K.: Hardware-Software Bipartitioning for Dynamically Reconfigurable Systems. Hardware/Software Codesign, 2002. Codes 2002. Proceedings of the Tenth International Symposium on, pp. 145–150.

[8] Pisinger, D.: Algorithms for Knapsack Problems. Ph.D. Thesis, 1995, pp. 1–200.

[9] Pisinger, D.: A Minimal Algorithm for the 0-1 Knapsack Problem. Operations Research, 1997, pp. 758–767.

[10] Jigang, W.—Yunfei, L.—Schroeder, H.: A Minimal Reduction Approach for the Collapsing Knapsack Problem. Computing and Informatics, Vol. 20, 2001, pp. 359–369.

Abhijit Ray received his B. Tech (Bachelor of Technology) degree in Electrical Engineering from Regional Engineering College, Kurukshetra, India (now National Institute of Technology, Kurkshetra) in 1998. He is currently pursuing his PhD in Computer Engineering at Nanyang Technological University (Singapore). His research interests are in processor selection, hardware/software co-design.

Wu Jigang received his B.S. degree in computational mathematics from Lanzhou University (China) in 1983. He received his PhD in computer software and theory from University of Science and Technology of China. He was successively an assistant professor, lecturer in Lanzhou University from 1983 to 1993. He was an associate professor in Yantai University (China) from 1993 to 2000. He has been with Nanyang Technological University (NTU), Singapore, since 2000. He is currently a research fellow in Centre for High Performance Embedded Systems, NTU. Dr. Wu has published more than 70 technical papers. His research interests include hardware/software co-design, reconfigurable computing and parallel computing.

Thambipillai Srikanthan has been with Nanyang Technological University (NTU), Singapore, since 1991, where he holds a joint appointment as Associate Professor and Director of the Centre for High Performance Embedded Systems. He received his B.Sc. (Hons) in Computer and Control Systems and PhD in System Modelling and Information Systems Engineering from Coventry University, United Kingdom. His research interests include system integration methodologies, architectural translations of compute intensive algorithms, high-speed techniques for image processing and dynamic routing. Dr. Srikanthan has published more than 180 technical papers and has served a number of administrative roles during his academic career. He is the founder and Director of the Centre for High Performance Embedded Systems, which is now a University level research centre at NTU. He is a corporate member of the IEE and a senior member of the IEEE.