Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

Information Processing Letters 98 (2006) 41–46

Information Processing Letters

www.elsevier.com/locate/ipl

Low-complex dynamic programming algorithm for hardware/software partitioning

Wu Jigang *, Thambipillai Srikanthan

School of Computer Engineering, Nanyang Technological University, Singapore 639798

Received 17 February 2005; received in revised form 1 December 2005; accepted 7 December 2005

Available online 17 January 2006

Communicated by F.Y.L. Chin

Abstract

A low-complex algorithm is proposed for the hardware/software partitioning. The proposed algorithm employs dynamic programming principles while accounting for communication delays. It is shown that the time complexity of the latest algorithm has been reduced from $O(n^2 \cdot \mathcal{A})$ to $O(n \cdot \mathcal{A})$, without increase in space complexity, for n code fragments and hardware area $\mathcal{A}$.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Dynamic programming; Algorithms; Complexity; Hardware/software partitioning

1. Introduction

Most modern electronic systems are composed by both hardware and software. In the design of such mixed hardware/software (Hw/Sw) systems, co-design techniques play more important role. It dominantly affects to overall system performance [1–5]. Hw/Sw partitioning has been proposed over the last decade. It transforms an application specification into communicating hardware and software components of an embedded system that exhibit the desired behavior and satisfy the performance constraints. Software is more flexible and cheaper, but hardware is faster. Thus, efficient techniques for Hw/Sw partitioning can achieve results superior to software-only solution.

Earlier approaches in [6–8] are hardware-oriented. They start with a complete hardware solution and iteratively move parts of the system to the software as long as the performance constraints are fulfilled. On the other hand, [2,9,10] are software-oriented, because they start with a software program moving pieces to hardware to improve speed until the time constraint is satisfied. In these approaches performance satisfiability is not part of the cost function. For this reason, the algorithms can easily be trapped in a local minimum.

Many approaches emphasis the algorithmic aspects, e.g., evolution algorithm [11], integer programming [12, 13], simulated annealing algorithm [2,14] and ant system algorithm [15]. These approaches are applied to different architectures and cost functions to provided sub-optimal solution minimizing the application execution time. It is difficult to name a clear winner because there have been no widely accepted benchmarks. Generally, they require more iterations resulting in longer design cycle times as the partitioning problem is NP-complete [16–18].

Despite many heuristics and approaches above, developing exact algorithms to find an optimal solution

* Corresponding author.
E-mail addresses: asjgwu@ntu.edu.sg (J. Wu), astsrikan@ntu.edu.sg (T. Srikanthan).

is still very important. Knudsen and Madsen proposed an algorithm called PACE employed in the LYCOS co-synthesis system for partitioning control data flow graphs (CDFG) into hardware and software parts [19, 20]. PACE is the latest dynamic programming approach. Its time complexity is $O(n^2 \cdot \mathcal{A})$ and the space complexity is $O(n \cdot \mathcal{A})$ for n code fragments and the available hardware area $\mathcal{A}$.

Unlike most of the previous work, in this paper, we take a theoretical approach focusing only on the algorithmic properties of hardware/software partitioning. In particular, we do not aim at partitioning for a given architecture, nor do we present a complete co-design environment. Rather, we restrict ourselves to the problem of deciding, based on given cost values, which components of the system to implement in hardware and which ones in software. Our contribution is reducing the time complexity of PACE from $O(n^2 \cdot \mathcal{A})$ to $O(n \cdot \mathcal{A})$ without increasing the space complexity.

2. Preliminaries

All assumptions in this paper are the same as those given in [19,20]. In detail, an application corresponds to a CDFG which is divided into basic scheduling code fragments/blocks (called blocks in short), that may be moved between hardware and software. The application is modeled as a sequence of blocks $B_1, B_2, \ldots, B_n$. The corresponding hardware area, hardware execution time, software execution time and intercommunication delays for each block are provided in advance by a synthesis system, e.g., LYCOS [20]. Fig. 1, cited from [19,20], shows the computational model for Hw/Sw partitioning, in which hardware blocks and software blocks cannot execute in parallel. It is assumed that the adjacent hardware blocks are able to communicate the read/write variables they have in common directly between them without involving the software side. In Fig. 1, a_i denotes the area penalty of moving block B_i to hardware, s_i denotes the inherent speedup of moving block B_i to hardware, and e_i denotes the extra speedup which is incurred because of blocks being able to communicate directly with each other when they are both placed in hardware. The objective is to find the optimal partition to realize the best possible speedup on a given hardware area $\mathcal{A}$.

Let $\mathcal{A}$ correspond to the knapsack size, and the block B_i correspond to the item i of the knapsack problem for $1 \leqslant i \leqslant n$. This problem can be reduced to the standard 0–1 knapsack problem, one of the NP-complete problems, for the particular case where the communications are ignored. It is clear that the problem which considers communication is more difficult than the one that does not, and thus the hardness of the problem considered in this paper is also NP-hard.

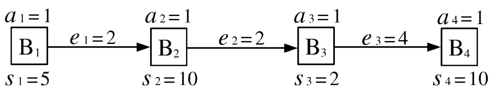

Fig. 1. Computational model of 4 blocks.

3. Algorithms

The algorithm PACE is a dynamic programming approach. It is based on the fact, that *any possible partition can be thought of as composed of sequences of blocks* [19,20], which leads to the higher computational complexity. Let $S_{i,j}$, $j \geqslant i \geqslant 1$, denote the sequence of blocks $B_i, B_{i+1}, \ldots, B_j$. G_j is defined as $\{S_{1,j}, S_{2,j}, \ldots, S_{j,j}\}$, which is called the jth group of the sequence. G_0 is defined as an empty set $\emptyset$. The area penalty $a_{i,j}$ of moving $S_{i,j}$ to hardware is computed as the sum of the individual block areas, i.e., $a_{i,j} = \sum_{k=i}^{j} a_k$. We use following notations to formulize PACE.

1. $\mathit{speedup}(S_{i,j}, a)$ denotes the inherent speedup of moving $S_{i,j}$ to hardware with available area a. For example, in Fig. 1, $\mathit{speedup}(S_{2,3}, 2) = 14$, that is the sum of s_2, e_2 and s_3. While $\mathit{speedup}(S_{2,3}, 1) = 0$ because of not enough hardware area for $S_{2,3}$, i.e., $a_2 + a_3 = 2 > 1$.
2. $\mathit{Bestsp}(G_j, a)$ denotes the best speedup achievable by first moving a sequence from G_j to hardware of area a, and then in the remaining area moving a sequence from one of the previous groups, $G_{j-1}, G_{j-2}, \ldots, G_1$, to hardware. $\mathit{Bestsp}(G_j, a)$ is set to 0 for $G_j = \emptyset$ or $a \leqslant 0$.
3. $\mathit{Bestsp}(G_1 G_2 \cdots G_j, a)$ denotes the best speedup achievable by moving sequences from $G_1, G_2, \ldots,$ or G_j to hardware of area a.

The algorithm PACE can be equivalently formulized to (A). The operation max over all values of j returns the maximum of the corresponding set.

$$
\text{(A)} \quad \begin{cases}
\mathit{Bestsp}(G_j, a) = 0 \quad \text{for } j = 0 \text{ or } a \leqslant 0; \\
\mathit{speedup}(S_{i,j}, a) = \begin{cases} 0 \quad \text{for } a < a_{i,j}; \\ \sum_{k=i}^{j} s_k + \sum_{k=i}^{j-1} e_k \\ \qquad \text{for } a \geqslant a_{i,j}; \end{cases} \\
\mathit{Bestsp}(G_j, a) = \max_{1 \leqslant i \leqslant j} \big\{\mathit{speedup}(S_{i,j}, a) \\
\qquad\qquad + \mathit{Bestsp}(G_{i-1}, a - a_{i,j})\big\}; \\
\mathit{Bestsp}(G_1 G_2 \cdots G_j, a) \\
\quad = \max\big\{\mathit{Bestsp}(G_j, a), \mathit{Bestsp}(G_1 G_2 \cdots G_{j-1}, a)\big\}; \\
\quad i \leqslant j, \ j = 1, 2, \ldots, n.
\end{cases}
$$

Let μ be an integer value called the area granularity, then the list of trial area is defined as $\langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i, \ldots, \mathcal{A}_m \rangle$, where $\mathcal{A}_i = i \cdot \mu$, and $\mathcal{A}_m = \mathcal{A}$. As the analysis in [19,20], the time complexity of PACE is $O(n^2 \cdot m)$. It is $O(n^2 \cdot \mathcal{A})$ for $\mu = 1$.

Unlike PACE, which relies on a sequence of blocks for computation, the proposed algorithm called SPACE (Simplified PACE) is based on the assignments of only the current block at a time. For example, assuming that the optimal HW/SW partitioning for $B_1, B_2, \ldots, B_{k-1}$ has been computed where the hardware area utilization is less than a, we now consider the method to partitioning the blocks $B_1, B_2, \ldots, B_k$ within the available area a. This is achieved by first arriving at all partitioning possibilities based on representing the current block B_k in software or in hardware. The optimal partitioning results in the best possible speedup. If B_k is implemented in software, the optimal partitioning for $B_1, B_2, \ldots, B_k$ for the hardware area a is identical to the optimal partitioning for $B_1, B_2, \ldots, B_{k-1}$ for hardware area a. If B_k is moved to hardware, the optimal partitioning for $B_1, B_2, \ldots, B_k$ can be found by examining partitioning for the blocks $B_1, B_2, \ldots, B_{k-1}$ for area $a - a_k$. We employ the following notations to further describe our algorithm.

1. $Bsp(k, a)$ denotes the best speedup achievable by moving some or all the blocks from $B_1, B_2, \ldots, B_k$ to hardware of size a. $Bsp(k, a)$ is set to 0 for $k = 0$ or $a = 0$.
2. $Bsp\_sw(k, a)$ denotes the best speedup achievable by keeping B_k in software and moving some or all the blocks $B_1, B_2, \ldots, B_{k-1}$ to hardware of size a. It is clear that $Bsp\_sw(k, a) = Bsp(k - 1, a)$ when the hardware area will not be occupied by block B_k. Set $Bsp\_sw(k, a)$ to 0 for $k = 0$ or $a = 0$.
3. $Bsp\_hw(k, a)$ denotes the best speedup achievable by moving B_k to hardware and then moving some or all blocks from $B_1, B_2, \ldots, B_{k-1}$ to area $a - a_k$. $Bsp\_hw(k, a)$ recursively depends on $Bsp\_sw(k - 1, a - a_k)$ and $Bsp\_hw(k - 1, a - a_k)$ because B_{k-1} has two possible assignments, each for the case of software and hardware. Set $Bsp\_hw(k, a)$ to $-\infty$ for $k = 0$ or $a = 0$.

The best speedup $Bsp(k, a)$ is the maximum between $Bsp\_sw(k, a)$ and $Bsp\_hw(k, a)$ as the block B_k is assigned either to software or to hardware, respectively. Thus, the proposed algorithm SPACE can be formulized to the following (B). e_0 is set to 0 throughout this paper.

$$\text{(B)}\ \begin{cases} Bsp\_sw(k,a) = 0, \qquad Bsp\_hw(k,a) = -\infty, \\ Bsp(k,a) = 0 \quad \text{for } k = 0 \text{ or } a = 0; \\ Bsp\_sw(k,a) = Bsp(k-1,a); \\ Bsp\_hw(k,a) \\ \quad = \begin{cases} -\infty & \text{for } a < a_k; \\ \max \begin{Bmatrix} Bsp\_sw(k-1, a-a_k) + s_k, \\ Bsp\_hw(k-1, a-a_k) + s_k + e_{k-1} \end{Bmatrix} \\ \qquad \text{for } a \geqslant a_k; \end{cases} \\ Bsp(k,a) = \max\{Bsp\_sw(k,a), Bsp\_hw(k,a)\}; \\ \quad k = 1, 2, \ldots, n. \end{cases}$$

Furthermore, according to $Bsp\_sw(k, a) = Bsp(k - 1, a)$, the formula (B) can be simplified to the formula (C) by replacing Bsp_sw with the corresponding Bsp.

$$\text{(C)}\ \begin{cases} Bsp\_hw(k,a) = -\infty, \qquad Bsp(k,a) = 0 \\ \quad \text{for } k \leqslant 0 \text{ or } a = 0; \\ Bsp\_hw(k,a) \\ \quad = \begin{cases} -\infty & \text{for } a < a_k; \\ \max \begin{Bmatrix} Bsp(k-2, a-a_k) + s_k, \\ Bsp\_hw(k-1, a-a_k) + s_k + e_{k-1} \end{Bmatrix} \\ \qquad \text{for } a \geqslant a_k; \end{cases} \\ Bsp(k,a) = \max\{Bsp(k-1,a), Bsp\_hw(k,a)\}; \\ \quad k = 1, 2, \ldots, n. \end{cases}$$

Let $(x_1, x_2, \ldots, x_n)$ be a feasible solution of the partitioning problem, where $x_i \in \{1, 0\}$. $x_i = 1$ ($x_i = 0$) indicates B_i is assigned to hardware (software), $1 \leqslant i \leqslant n$. Assuming, without loss of generality, that PACE achieves the optimal solution $(x_1, x_2, \ldots, x_{k-1}, 1, 0, \ldots, 0)$, which implies that B_k is the last block moving to hardware, we show our algorithm SPACE can find the optimal solution.

In PACE, one block is treated as a special block sequence of length 1, e.g., B_k can be viewed as the block sequence $S_{k,k}$. It is clear that the hardware area required by B_k is no larger than the given area a because B_k is assigned to hardware in the optimal solution. Thus, it is confirmed that, in formula (A), the optimal solution is produced from the sub-formula

$$Bestsp(G_k, a) = speedup(S_{k,k}, a) + Bestsp(G_{k-1}, a - a_{k,k}).$$

It implies that the best speedup in $B_1, B_2, \ldots, B_k$ consists of the speedup of B_k and the best speedup of the blocks $B_1, B_2, \ldots, B_{k-1}$, and this is directly reflected in the calculation for $Bsp\_hw(k, a)$ in formula (C). Therefore, the two algorithms are equivalent with respect to the ability of achieving the optimal solution.

The following pseudo-code of the algorithm SPACE is for n blocks and the list of trial area $\langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_j, \ldots, \mathcal{A}_m \rangle$. The best speedup $Bsp(k, \mathcal{A}_j)$ is calculated by the nested for-loops according to the formula (C), followed by backtracking calculations as those employed in PACE for determining the optimal solution.

In SPACE, the calculations for *Bsp* and *Bsp_hw* are tracked by using the 3-tuple arrays *trace* and *trace_hw*, respectively. For example, $trace(k, \mathcal{A}_j) = \langle 'Bsp', k-1, \mathcal{A}_j\rangle$ means that the kth block is assigned to software and the backtracking continues in $trace(k-1, \mathcal{A}_j)$, and $trace(k, \mathcal{A}_j) = \langle 'Bsp\_hw', k, \mathcal{A}_j\rangle$ means that the kth block is assigned to hardware and the backtracking continues in $trace\_hw(k, \mathcal{A}_j)$. The array $partition\_list[1:n]$ is used to store the solution partitioning n blocks within the area $\mathcal{A}_m$.

```
Input: area penalty $a_i$, inherent speedup $s_i$ and
  extra speedup $e_i$, for $1 \leqslant i \leqslant n$;
  trial area $\mathcal{A}_j$ for $1 \leqslant i \leqslant m$.
Output: the solution $partition\_list[1:n]$.
Algorithm SPACE
begin
for all $a \leqslant \mathcal{A}_m$ and $k \leqslant n$ do {/* initializing */
    $Bsp\_hw(0,a) := Bsp\_hw(k,0) := -\infty$; $e_0 := 0$;
    $Bsp(-1,a) := Bsp(0,a) := Bsp(k,0) := 0$; }
  for $k := 1$ to $n$ do
    for $j := 1$ to $m$ do {
      /* computing Bsp_hw and making trace_hw */
      if $\mathcal{A}_j < a_k$ then {
        $Bsp\_hw(k,\mathcal{A}_j) := -\infty$;
        $trace\_hw(k,\mathcal{A}_j) := \langle 'Bsp', k-1, \mathcal{A}_j\rangle$}
      else {
        $temp1 := Bsp(k-2, \mathcal{A}_j - a_k) + s_k$;
        $temp2 := Bsp\_hw(k-1, \mathcal{A}_j - a_k) + s_k + e_{k-1}$;
        if $temp1 > temp2$ then {
          $Bsp\_hw(k,\mathcal{A}_j) := temp1$;
          $trace\_hw(k,\mathcal{A}_j) := \langle 'Bsp', k-2, \mathcal{A}_j - a_k\}$
        else {
          $Bsp\_hw(k,\mathcal{A}_j) := temp2$;
          $trace\_hw(k,\mathcal{A}_j) := \langle 'Bsp\_hw', k-1, \mathcal{A}_j - a_k\rangle\}$
        };
        /* computing Bsp and making trace */
        if $Bsp(k-1,\mathcal{A}_j) > Bsp\_hw(k,\mathcal{A}_j)$
        then { $Bsp(k,\mathcal{A}_j) := Bsp(k-1,\mathcal{A}_j)$;
          $trace(k,\mathcal{A}_j) := \langle 'Bsp', k-1, \mathcal{A}_j\rangle\}$;
        else { $Bsp(k,\mathcal{A}_j) := Bsp\_hw(k,\mathcal{A}_j)$;
          $trace(k,\mathcal{A}_j) := \langle 'Bsp\_hw', k, \mathcal{A}_j\rangle\}$;
      }; /* end of the two for-loops */
  /* backtracking along the trace for the solution */
  $\langle answ, numb, area\rangle := trace(n, \mathcal{A}_m)$; /* initializing */
  for $i := 1$ to $n$ do $partition\_list[i] := 'sw'$;
  repeat
    if $answ = 'Bsp\_hw'$ then {
      $partition\_list[numb] := 'hw'$;
      $\langle answ, numb, area\rangle := trace\_hw(numb, area)$ }
    else $\langle answ, numb, area\rangle := trace(numb, area)$;
  until $(numb < 1)$ or $(area \leqslant 0)$;
  end.
```

Intuitively, Figs. 2 and 3 show how PACE and SPACE execute for the example given in Fig. 1. It is clear that SPACE is simpler than PACE whereas both algorithms produce the same optimal solution. In Fig. 2, S_{12} and S_{22} (denote $speedup(S_{1,2}, a)$ and $speedup(S_{2,2}, a)$, respectively) are calculated in advance for $Bestsp(G_2, a)$ and $Bestsp(G_1G_2, a)$. The operation max works on the set of 2 elements calculated by *addition*, but the size of the set increases to 4 for $Bestsp(G_4, a)$. However, the calculations in Fig. 3 are quite simple. Unlike PACE, the operation max in SPACE always works on the set of at most two elements calculated by *addition* for all $Bsp\_hw(k, a)$, $1 \leqslant k \leqslant 4$. This provides for elegant means to accelerating the computation. The computing trace of the optimal solution is shown by the underlined data.

Theorem 1. *Given n blocks and the list of trial hardware area $\langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m\rangle$, both the time complexity and the space complexity of SPACE are* $O(n \cdot m)$, *i.e.,* $O(n \cdot \mathcal{A})$ *for total hardware area $\mathcal{A}$ with granularity of* 1.

Proof. According to the formula (C), SPACE directly uses the basic information of each block, i.e., a_k, s_k and e_{k-1} for block B_k, $k \leqslant n$, to calculate the current optimal partition. The operation max works on the set of only 2 elements produced by at most 3 *addition*s per iteration. Hence, the computing time for $Bsp(k, a)$ is bounded by $O(1)$ for the current k and a. This concludes that the computing time for $Bsp(n, \mathcal{A}_m)$ is bounded by $O(n \cdot m)$, which corresponds to the running time of the nested for-loops in the pseudo-code of SPACE, for n blocks and the list of trial area $\langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m\rangle$. On the other hand, the backtracking for finding the optimal solution can be finished in $O(\max(n, m))$. Hence, the time complexity of SPACE is dominated by the nested for-loops of pseudo-code and it is bounded by $O(n \cdot m)$.

On the space requirement, SPACE has to keep the values of $Bsp(i, j)$ for all $i \leqslant n-1$ and $j \leqslant m$ to compute $Bsp(n, m)$. That is the characteristic of dynamic programming algorithms. In the pseudo-code, $Bsp(n, m)$, $Bsp\_hw(n, m)$, $trace(n, m)$ and $trace\_hw(n, m)$ are $n \times m$ arrays. This concludes that the space complexity of SPACE is bounded by $O(n \cdot m)$. □

4. Simulations

To verify above analysis in complexity, SPACE and PACE are simulated in C on a personal computer—Intel Pentium-4, 3 GHz, 1 GB RAM. For block B_k, $1 \leqslant k \leqslant n$, a_k is randomly generated and satisfies $\sum_{k=1}^{n} a_k \leqslant \mathcal{A}$ for a given area $\mathcal{A}$. The speedup s_k and e_k are randomly generated in $[100, 1000]$ and in $[10, 100]$, respectively.

		Area a		
		1	2	3
G_1	S_{11}	5	5	5
$Bestsp(G_1, a)$		$\max\{5\} = 5$	$\max\{5\} = 5$	$\max\{5\} = 5$
G_2	S_{12}	0	17	17
	S_{22}	10	10	10
$Bestsp(G_2, a)$		$\max\{0, 10\} = 10$	$\max\{17, 10 + 5\} = 17$	$\max\{17, 10 + 5\} = 17$
$Bestsp(G_1G_2, a)$		$\max\{10, 5\} = 10$	$\max\{17, 5\} = 17$	$\max\{17, 5\} = 17$
G_3	S_{13}	0	0	21
	S_{23}	0	14	14
	S_{33}	2	2	2
$Bestsp(G_3, a)$		$\max\{0, 0, 2\} = 2$	$\max\{0, 14 + 0, 2 + 10\} = 14$	$\max\{21, 14 + 5, 2 + 17\} = 21$
$Bestsp(G_1G_2G_3, a)$		$\max\{2, 10\} = 10$	$\max\{14, 17\} = 17$	$\max\{21, 17\} = 21$
G_4	S_{14}	0	0	0
	S_{24}	0	0	28
	S_{34}	0	16	16
	S_{44}	10	10	10
$Bestsp(G_4, a)$		$\max\{0, 0, 0, 10\} = 10$	$\max\{0, 0, 16 + 0, 10 + 10\} = 20$	$\max\{0, 28 + 0, 16 + 10, 10 + 17\} = 28$
$Bestsp(G_1G_2G_3G_4, a)$		$\max\{10, 10\} = 10$	$\max\{20, 17\} = 20$	$\max\{28, 21\} = 28$

Fig. 2. Computations of PACE for the example shown in Fig. 1.

	Area a		
	1	2	3
$Bsp\_hw(1, a)$	$\max\{0 + 5, -\infty + 5 + 0\} = 5$	$\max\{0 + 5, -\infty + 5 + 0\} = 5$	$\max\{0 + 5, -\infty + 5 + 0\} = 5$
$Bsp(1, a)$	$\max\{(0, 5\} = 5$	$\max\{0, 5\} = 5$	$\max\{0, 5\} = 5$
$Bsp\_hw(2, a)$	$\max\{\underline{0 + 10}, -\infty + 10 + 2\} = 10$	$\max\{0 + 10, 5 + 10 + 2\} = 17$	$\max\{0 + 10, 5 + 10 + 2\} = 17$
$Bsp(2, a)$	$\max\{5, 10\} = 10$	$\max\{5, 17\} = 17$	$\max\{5, 17\} = 17$
$Bsp\_hw(3, a)$	$\max\{0 + 2, -\infty + 2 + 2\} = 2$	$\max\{5 + 2, \underline{10 + 2 + 2}\} = \underline{14}$	$\max\{5 + 2, 17 + 2 + 2\} = 21$
$Bsp(3, a)$	$\max\{10, 2\} = 10$	$\max\{17, 14\} = 17$	$\max\{17, 21\} = 21$
$Bsp\_hw(4, a)$	$\max\{0 + 10, -\infty + 10 + 4\} = 10$	$\max\{10 + 10, 2 + 10 + 4\} = 20$	$\max\{17 + 10, \underline{14 + 10 + 4}\} = \underline{28}$
$Bsp(4, a)$	$\max\{10, 10\} = 10$	$\max\{17, 20\} = 20$	$\max\{21, \underline{28}\} = \underline{28}$

Fig. 3. Computations of SPACE for the example shown in Fig. 1.

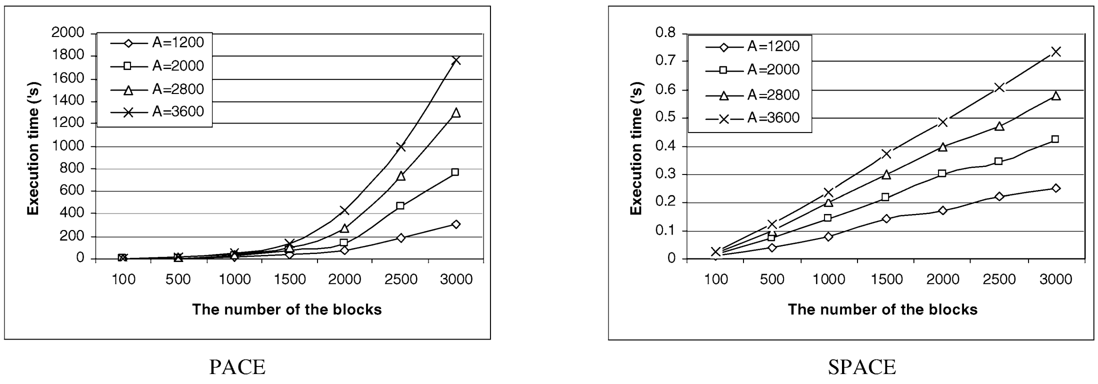

Fig. 4. Comparisons in execution time between PACE and SPACE.

Fig. 4 shows the simulation results for the execution times of the algorithms SPACE and PACE. It is clearly evident that the execution time of PACE increases with the number of the blocks (n) in the order of $O(n^2)$ for a given hardware area $\mathcal{A}$. The execution time of SPACE on the other hand increases only in the order of $O(n)$. Moreover, the execution time of SPACE is far less than that of PACE. For example, the execution time of PACE is about 1000 s while the execution time of SPACE is only about 0.6 s for the case of 2500 randomly generated blocks and area of size 3600 units. Simulation results clearly show that the proposed algorithm SPACE is notably faster than PACE.

5. Conclusions

We have proposed a new dynamic programming algorithm to accelerate the Hw/Sw partitioning process. It is shown that the proposed algorithm is superior to PACE in terms of time complexity. Simulation results confirm that it provides for optimal partitioning even when communication overheads are incorporated. Furthermore, it has been verified that the time complexity of the latest algorithm is reduced from $\mathrm{O}(n^2 \cdot \mathcal{A})$ to $\mathrm{O}(n \cdot \mathcal{A})$, without increase in space complexity, where n refers to the number of blocks for hardware area $\mathcal{A}$.

References

[1] J.I. Hidalgo, J. Lanchares, Functional partitioning for hardware–software codesign using genetic algorithms, in: Proc. of 23rd EUROMICRO Conf. New Frontiers of Information Technology, 1997, pp. 631–638.

[2] R. Ernst, J. Henkel, T. Benner, Hardware–software co-synthesis for micro-controllers, IEEE Design Test Comput. 10 (1993) 64–75.

[3] J. Harkin, T.M. McGinnity, L.P. Maguire, Partitioning methodology for dynamically reconfigurable embedded systems, IEE Proc. Comput. Digital Techniques 147 (2000) 391–396.

[4] L. Bianco, M. Auguin, G. Gogniat, A. Pegatoquet, A path analysis based partitioning for time constrained embedded systems, in: Proc. 6th Internat. Workshop on Hardware/Software Codesign (CODES/CASHE), 1998, pp. 85–89.

[5] V. Srinivasan, S. Govindarajan, R. Vemuri, Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures, IEEE Trans. VLSI Syst. 9 (2001) 140–158.

[6] R. Niemann, P. Marwedel, Hardware/software partitioning using integer programming, in: Proc. of IEEE/ACM European Design Automation Conference (EDAC), 1996, pp. 473–479.

[7] R. Gupta, G.D. Micheli, Hardware–software cosynthesis for digital systems, IEEE Design Test Comput. 10 (1993) 29–41.

[8] R.K. Gupta, C. Coelho, G. De Micheli, Synthesis and simulation of digital systems containing interacting hardware and software components, in: Proc. 29th ACM, IEEE Design Automation Conference, 1992, pp. 225–230.

[9] F. Vahid, D.D. Gajski, J. Gong, A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning, in: Proc. of IEEE/ACM European Design Automation Conference (EDAC), 1994, pp. 214–219.

[10] F. Vahid, D.D. Gajski, Clustering for improved system-level functional partitioning, in: Proc. 8th IEEE/ACM Internat. Symp. System Synthesis, 1995, pp. 28–33.

[11] G. Quan, X. Hu, G.W. Greenwood, Preference-driven hierarchical hardware/software partitioning, in: Proc. of IEEE Internat. Conf. on Computer Design, 1999, pp. 652–657.

[12] R. Niemann, P. Marwedel, An algorithm for hardware/software partitioning using mixed integer linear programming, in: Design Automation for Embedded Systems, Special Issue: Partitioning Methods for Embedded Systems, vol. 2, 1997, pp. 165–193.

[13] M. Weinhardt, in: Integer Programming for Partitioning in Software Oriented Codesign, in: Lecture Notes in Comput. Sci., vol. 975, Springer, Berlin, 1995, pp. 227–234.

[14] Z. Peng, K. Kuchcinski, An algorithm for partitioning of application specific system, in: Proc. IEEE/ACM European Design Automation Conference (EDAC), 1993, pp. 316–321.

[15] G. Wang, W. Gong, R. Kastner, A new approach for Task level computational resource bi-partitioning, in: Proc. Internat. Conf. on Parallel and Distributed Computing and Systems (PDCS), 2003.

[16] S. Jinwoo, K. Dong-In, S.P. Crago, A communication scheduling algorithm for multi-FPGA systems, in: Proc. IEEE Symp. Field-Programmable Custom Computing Machines, 2000, pp. 299–300.

[17] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, B. Schmidt, Dynamic scheduling of tasks on partially reconfigurable FPGAs, IEE Proc. Comput. Digital Techniques 147 (2000) 181–188.

[18] H. Oudghiri, B. Kaminska, Global weighted scheduling and allocation algorithms, in: Proc. of IEEE/ACM European Design Automation Conference, 1992, pp. 491–495.

[19] P.V. Knudsen, J. Madsen, PACE: A dynamic programming algorithm for hardware/software partitioning, in: Proc. of 4th IEEE/ACM Internat. Workshop Hardware/Software Codesign, 1996, pp. 85–92.

[20] J. Madsen, J. Grode, P.V. Knudsen, M.E. Petersen, A. Haxthausen, LYCOS: The Lyngby co-synthesis system, Design Automat. Embedded Syst. 2 (1997) 195–235.