Loop Pipelining in Hardware-Software Partitioning

Jinhwan Jeon and Kiyoung Choi
School of Electrical Engineering
Seoul National University
Seoul, Korea 151-742
Tel: +82-2-880-5457
Fax: +82-2-887-6575
e-mail: {jeonjinh,kchoi}@poppy.snu.ac.kr

Abstract

This paper presents a hardware-software partitioning algorithm that exploits a loop pipelining technique. The partitioning algorithm is based on iterative improvement. The algorithm tries to minimize hardware cost through hardware sharing and hardware implementation selection without violating given performance constraint. The proposed loop pipelining technique, which is an adaptation of a compiler optimization technique for instruction level parallelism, increases parallelism within a loop by transforming the structure of an input system description. By combining this technique with our partitioning algorithm, we can further reduce the hardware cost and/or improve the performance of the partitioned system. Experiments show about 19% performance improvement and 44% reduced hardware for a JPEG encoder design, compared to the results without loop pipelining.

1. Introduction

Mixed hardware and software implementation is common in the design of digital systems such as communication systems, DSP applications, and embedded systems. In general, software is easy to modify, maintain, and upgrade, though it is slow compared to hardware. Hardware can be made faster than software but the cost for all hardware solution is usually too high. An issue raised in designing such systems is to find an optimal point in between all hardware solution and all software solution where we obtain maximum performance at a minimum hardware cost. This process is called hardware-software partitioning.

There are some heuristics proposed for hardware-software partitioning problem [1, 3, 4, 5, 6, 7, 9, 11]. Gupta et al.[4] proposed a hardware-oriented approach in which all the operations except for the data dependent delay operations are initially mapped to hardware. Then the partitioner repeats moving a node to software while performance constraints are met. Selection of the node to be moved to software is done by a greedy method. Ernst, Henkel, and Benner [3, 5] proposed a software-oriented approach in which all the units of partitioning(called BSB) are initially mapped to software. They used a simulated annealing algorithm for partitioning. Vahid et al. [11] proposed a binary-constraint search algorithm which searches the design space while changing the hardware constraint in a binary-search fashion. For each hardware constraint, they run a simulated annealing algorithm and check to see if the performance constraint is met. Kalavade et al. [6, 7] proposed a global criticality/local phase(GCLP) driven algorithm and hardware-software mapping and implementation bin selection(MIBS) algorithm. The key feature of the algorithm is the adaptive objective mechanism by global and local measures. This algorithm applies two objectives according to global-time criticality. If the global time is critical, the objective function is to reduce the latency. Otherwise, the objective function is to minimize the hardware resources. Since GC does not contain the information on each node, local-phase(LP) is used to represent the preference of each node. MIBS is the extension of GCLP. This algorithm not only partitions the nodes but also finds the implementation method of a hardware node to minimize hardware resource. Knudsen et al. [9] proposed a dynamic programming algorithm. This algorithm assumes an execution model in which software cannot execute other jobs while hardware is running, and assumes a realistic communication model. Based on this model, the algorithm finds an optimal solution using dynamic programming method. Adams et al. [1] proposed a multiple-process behavioral synthesis algorithm for heterogeneous systems. They use an inter-process code motion to partition and allocate an input system description which is originally composed of one process. To schedule code segments within a process, intra-process code motion is used. Such code motions are made at random to increase concurrency between processes and to improve performance and cost, while overall performance and cost is optimized by simulated annealing.

The approaches in [3, 5, 9, 11] assume an execution model that does not allow parallel execution of hardware and software, leading to limited performance improvement. Though other approaches [1, 4, 6, 7] exploit the parallelism that resides in the system, they only utilize the explicit parallelism given by the input description. Therefore, codesign approach for software acceleration seems to have little advantage over pure software solution [12].

In this paper, we propose a loop pipelining technique which increases parallelism for more effective hardware-software partitioning. In addition, we propose yet another partitioning algorithm which maps nodes to hardware or software considering various hardware implementation alternatives and hardware resource sharing. Though the proposed loop pipelining technique is not a new idea, we show that a simple combination of pipelining technique with partitioning algorithm gives more room for software acceleration with less hardware cost than existing algorithms, which is the contribution of our work. Our partitioning algorithm with loop pipelining is suitable for computation-intensive applications mainly composed of loops, as is common in most DSP applications. Bakshi et al. recently proposed a method [13] which also deals with the same subject: partitioning and pipelining. They perform hardware-software partitioning by simply mapping a node to hardware if it cannot meet throughput constraint in software by itself. Then they repeat an optimization process consisting of pipelining, scheduling and processor allocation until the throughput constraint is met. However, their simple partitioning scheme does not work - that is, all the nodes are mapped to software - when every node meets the throughput constraint, as is often the case when small granularity is used. In that case, they prefer all software solution in multi-processor target architecture to mixed hardware-software solution. However, multi-processor solution is not always cheaper than one-processor solution with small ASIC. In our approach we perform pipelining before partitioning. Therefore, we can consider all the nodes as hardware candidates. Moreover, we can combine our loop pipelining technique with any other existing partitioning algorithm.

Our paper is organized as follows. In section 2 we give an overview of our partitioning algorithm. In section 3, we propose a loop pipelining technique for partitioning and an algorithm for solving the extended hardware-software partitioning problem. Section 4 shows experimental results before we conclude in section 5.

2. Overview

Figure 1 illustrates the steps used to partition an input system description. The first step is to transform an input behavioral description into a CDFG which is used as an intermediate format for hardware-software partitioning. The CDFG is formally defined as a graph G=(N, E), where each node represents an operation or a set of operations (e.g., task, process, and code grouping) and each edge represents data and control dependency between nodes.

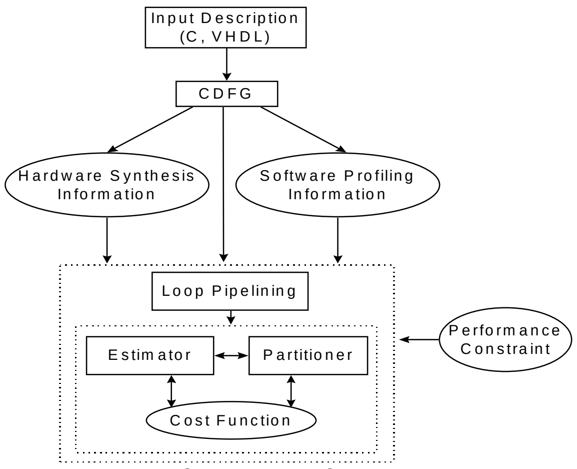

Input Description (C, VHDL)
CDFG
Hardware Synthesis Information
Software Profiling Information
Loop Pipelining
Estimator
Partitioner
Cost Function
Performance Constraint

Figure 1. Overview of hardware-software partitioning steps.

The second step is to obtain hardware synthesis and software profiling information for partitioning. Task(leaf procedure or function) level granularity is used to obtain such information from the CDFG. Hardware synthesis is done by Hyper [2]. Hyper is a performance-constrained area-minimizing high-level synthesis tool. We can obtain many implementation alternatives by changing the performance constraint. The synthesis information from Hyper includes the number of used execution units, the number of register files, total execution delay, and the cost for each execution unit. For the purpose of hardware synthesis, we must translate the CDFG into Silage [2]. Since we have not implemented an automatic translator yet, the translation is done manually. However, we have implemented a tool for translation from CDFG to C code and we use it to obtain software profiling information including execution delay and invocation count of each task.

The final step is to partition the CDFG into hardware and software part to satisfy the performance constraint given by the user. The hardware synthesis information and the software profiling information obtained in the previous step are used for estimating the execution time and the hardware cost. The partitioning step is composed of loop pipelining stage and iterative improvement stage. Loop pipelining is performed before the iterative improvement stage in order to increase parallelism within a loop. In the iterative improvement stage, hardware-software partitioning is performed such that the cost is minimized while maintaining the performance above the constraint. In this stage, we consider various hardware implementation alternatives to select possibly the best one, share hardware modules, as well as perform hardware-software mapping.

Currently our target architecture consists of a single general purpose processor and multiple ASICs, although the proposed algorithm can be extended to the case of multiple processors by replacing the performance estimation method(in section 3.2) with a scheduling method proposed in [4]. We assume a memory mapped communication model where no hardware is dedicated for communication - that is, software is blocked until communication completes.

3. Partitioning Approach

3.1. Notation

Our partitioning algorithm focuses on performance-constrained hardware cost minimization. The performance constraint given by the user is denoted as D. We denote a node in a CDFG as n_i and an edge from n_i to n_j as $e_{i,j}$. Each node n_i has information including the execution delay(d_i) and hardware-software mapping. Each node n_i has hardware implementation alternatives which can be represented by an implementation curve IH_i. Such curves can be obtained using Hyper [2].

We denote the set of all the predecessors(successors) of n_i as $pred(n_i)(succ(n_i))$. If a node n_i is chosen to be implemented in hardware and the predecessors(successors) are implemented in software we insert a communication nodes between n_i's predecessors(successors) and n_i. We denote the communication node between n_j and $n_i(n_i$ and $n_k)$ as $nc_{j,i}(nc_{i,k})$, and the communication delay as $dc_{j,i}(dc_{i,k})$, where $n_j \in pred(n_i)(n_k \in succ(n_i))$.

3.2. Estimation

The partitioner evaluates the quality of a partitioned system based on two metrics; total execution delay and total hardware cost. First, the total execution delay is estimated by a simple list scheduling algorithm, which is similar to the one proposed in [10]. For the list scheduling of hardware nodes, priority is given to a node with the largest sum of its own delay and all successors' delays, thereby allowing the most critical hardware node to be scheduled first. For software nodes, priority is given to a node which has a hardware successor with the highest priority, thereby allowing a software node that leads to the most critical hardware node to be scheduled first. We prioritize software nodes only for a better scheduling of hardware nodes because ordering of software nodes does not affect the performance of the software when we use a single processor. According to this scheme, priority value p_i of a node n_i is defined as

$$p_i = \begin{cases} \sum_{n_k \in succ(n_i)} d_k + d_i & if\, n_i \in N_{HW} \\ \max_{n_k \in succ(n_i) \cap N_{HW}} (p_k) & if (n_i \in N_{SW}) \wedge (succ(n_i) \cap N_{HW} \neq \emptyset) \\ 0 & otherwise \end{cases}$$

where $N_{HW}(N_{SW})$ denotes the set of all hardware(software) nodes. We consider hardware sharing effect during list scheduling by making the sharing nodes have the same resource id. This list scheduling algorithm is applied to each basic block in the CDFG to obtain an estimation of the execution delay for each basic block. Then, by recursively summing up all the values obtained by multiplying invocation count of each basic block to the block's execution delay, we can calculate the total execution delay.

We estimate the total hardware cost based on the synthesis information provided by Hyper. In Hyper, this cost is hardware area. If there is no sharing among hardware nodes, hardware cost is estimated simply by summing up the hardware cost of each hardware node. If multiple hardware nodes share hardware resources, the total cost is reduced by the amount of shared resources. To consider resource sharing, we need to example the hardware architecture. The target architecture of Hyper is composed of execution units, register files, a control unit, and multiplexers which are connected by a crossbar network. Currently, among these hardware resources, we consider only the execution units as hardware resources that can be shared. We estimate the total hardware cost by subtracting the cost of shared resources from the sum of all the hardware nodes' costs. We ignore the area increase due to the added multiplexers and wiring.

3.3. Loop Pipelining

Since loop is generally the most time-critical part in the computation-intensive applications, there have been many loop optimization techniques for parallel computing. Software pipeline is one of those techniques, which overlaps the execution of code blocks in different iteration steps. To allow such an execution overlap, there must be no data dependency between subsequent loop iterations. Most data processing algorithms, which receive an input data stream and generates an output data stream, have a structure suitable for this kind of optimization.

Loop pipelining technique for partitioning, which we propose in this paper, is an adaptation of software pipeline technique to increase the parallelism within a loop. We can exploit the parallelism through concurrent execution of hardware and software as well as concurrent execution of hardware modules. Our loop pipelining technique for partitioning consists of the following three steps. We assume that user gives the number of pipeline stages(N_{ps}) beforehand.

1. Find feedback edges which represent data dependencies to the next iteration of the loop. Then, for each feedback edge $e_{i,j}$, make a cluster node which consists of nodes that exist between n_i and n_j. Finally, recursively merge cluster nodes that share a node into a cluster node such that there are no common nodes among cluster nodes.
2. By grouping nodes and/or cluster nodes in topological order, make initial pipeline blocks which can be overlapped within the loop. Then by repeatedly moving a node from one pipeline block to the neighboring pipeline block, find an optimized set of pipeline blocks such that the communication between pipeline blocks is minimized and delays of all the blocks are balanced. During this process, we make the number of pipeline blocks equal to N_{ps}.
3. Transform the loop such that all the pipeline blocks can run in parallel.

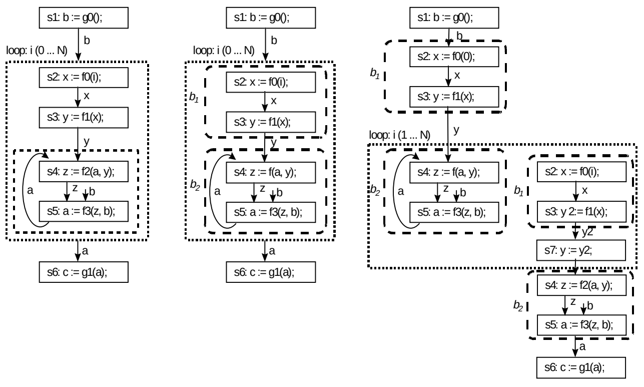
The purpose of the first step is to prevent a feedback edge from being cut by pipeline block boundary. All the nodes connected by a feedback edge are put into a cluster node. Otherwise, the pipelining may cause data dependency violation. Figure 2 (a) shows this step.

In the second step, an initial set of pipeline block is built by grouping nodes such that the number of pipeline blocks is equal to N_{ps}. In Figure 2 (b), two pipeline blocks, b_1 and b_2, are built by the procedure listed in step 2. The criteria for grouping is the execution delay of each pipeline block and the communication between subsequent pipeline blocks. First, it is desirable that the execution delay of each pipeline block be equal in order to reduce the critical path of the transformed loop and increase parallelism among partitioned blocks. Secondly, since variable copy instructions (*s7* in Figure 2 (c)) should be inserted to compensate for the cut edges, it is desirable that the communication between subsequent pipeline blocks be minimized in order to reduce the overhead induced by variable copy instructions and communication overhead from or to hardware nodes. For these optimizations, we use a greedy method which reduces cost function f_L defined as

$$f_L = \sum_{i=1}^{P} \left| d_{b_i} - \frac{d_{loop}}{P} \right| + \alpha \cdot \sum_{i=1}^{P-1} n_{comm}(b_i)$$

where P, d_{bi}, d_{loop}, $n_{comm}(b_i)$, and α are number of pipeline blocks, delay of a pipeline block b_i, execution delay of the loop, communication overhead from b_i to b_{i+1}, and weighting factor, respectively.

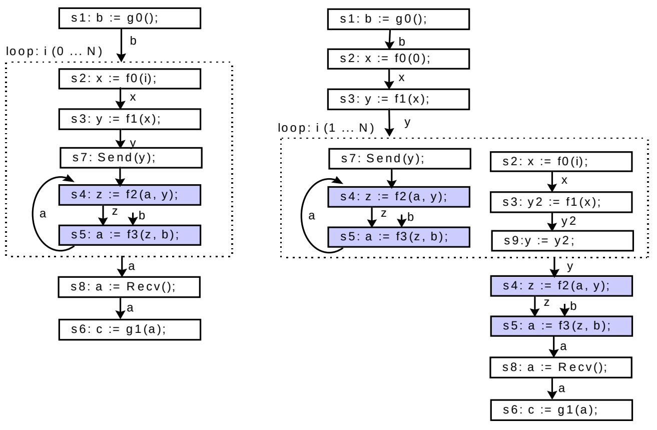
In the final step, we transform the loop such that all the pipeline blocks can run in parallel, as shown in Figure 2 (c). Note that variable *y* in *s3* in the loop is renamed as *y2* so that b_1 and b_2 can run in parallel, and *s7* is added as an epilogue code of the loop so that b_2 can use the updated value of variable *y* in the next iteration. Recall that in the second step of loop pipelining technique, we try to reduce the overhead of this epilogue code.



(a) merging nodes connected by a feedback edge (b) making pipeline block (c) overlapping pipeling blocks

Figure 2. Loop transformation by loop pipelining.

This method can also be used as a post optimizer to improve the performance of a partitioned system. Figure 3 illustrates the performance improvement by loop transformation. Assume Figure 3 (a) is the partitioned system by hardware-software partitioner, where *s4* and *s5* is mapped to hardware. In the original structure, the processor should be idling while *s4* and *s5* are running because there are no jobs to execute in parallel. However, if we transform the structure of the loop as shown in Figure 3 (b), the processor can execute *s2* and *s3* while the hardware is running *s4* and *s5*.



(a) partitioned system (b) transformed system

Figure 3. Loop transformation as a post optimizer for a partitioned system.

3.4. Partitioning Algorithm

Our partitioning algorithm makes decisions regarding the implementation of a hardware node, sharing among hardware nodes, and hardware-software mapping, subject to given performance constraint. Figure 4 shows a pseudo code of the partitioning algorithm, where total_delay() procedure and total_cost() procedures are total execution delay estimator and total hardware cost estimator, respectively. The Reduce-Cost() procedure used in the algorithm tries to incrementally reduce and update total hardware cost according to the procedural steps listed below:

1. Find a pair of hardware modules which reduce total hardware cost maximally by sharing, while satisfying the performance constraint.
2. Find a node $n_i \in N_{HW}$ which reduces the total hardware cost maximally while satisfying the performance constraint, when it is moved to another point in the implementation curve IH_i.
3. Find a node $n_i \in N_{HW}$ which reduces the total hardware cost maximally while satisfying the performance constraint, when it is mapped to software.
4. Among the candidates obtained from steps 1-3, adopt the candidate whose cost reduction is maximum.
5. Repeat steps 1-4 until no more candidates are available.

```
/* Input: CDFG $G(N, E)$, performance constraint $D$   */
/* Output: partitioned CDFG $G(N, E)$                  */
Partition ($G(N, E)$, $D$)
{
$G^{best}=G$;   $N_{HW}=\phi$;   $N_{SW}=N$;
/* first phase */
$G$=GreedyPartition($G$);            /* initial partition by greedy method */
$G^{best}$ = ReduceCost($G$);        /* Reduce hardware cost */
$cost^{best}$ = total_cost($G^{best}$);   / * cost of initial partition */

/* second phase : iterative improvement */
do {
  $N_{fixed}=\phi$;

  /* first inner loop: over allocate hardware node */
  for (i=0; i<$N_{max}$; i++) {
    $n_{can}=n_i \in N_{SW}$, where {speedup/cost} is maximum in HW;
    $N_{HW}=N_{HW} \cup n_{cand}$; $N_{SW}=N_{SW} - n_{cand}$;
    $G'$=ReduceCost($G$);

    if ($cost^{best}$ > total_cost($G'$))
      {   $G^{best} = G'$;          $cost^{best}$ = total_cost($G'$);     }
  }

  /* second inner loop: deallocate hardware node */
  do {
    /* map a node with the maximum hardware cost to software */
    $n_{cand} = n_i \in (N_{HW} - N_{fixed})$, where cost reduction is maximum in SW;
    $N_{SW}=N_{SW} \cup n_{cand}$; $N_{HW}=N_{HW} - n_{cand}$; $N_{fixed}=N_{fixed} \cup n_{cand}$;

    /* map SW nodes to HW to meet performance constraint */
    while (total_ delay($G$) > $D$) {
      $n_{cand}=n_i \in (N_{SW} - N_{fixed})$, where {speedup/cost} is maximum in
HW;
      $N_{HW} = N_{HW} \cup n_{cand}$; $N_{SW} = N_{SW} - n_{cand}$; $N_{fixed} = N_{fixed} \cup n_{cand}$;
    }
    $G'$=ReduceCost($G$);

    if ($cost^{best}$ > total_cost($G'$))
      {   $G^{best} = G'$;          $cost^{best}$ = total_cost($G'$);     }
  } while ($N_{fixed} \neq N$);
  $G = G^{best}$;
} while (cost improvement is obtainable);
return $G$;
}
```

Figure 4. Pseudo code of the partitioning algorithm.

The partitioning algorithm consists of two phases. In the first phase of the partitioning algorithm, starting from all software solution, GreedyPartition() procedure makes an initial partition that satisfies the performance constraint by repeatedly mapping a node, which has the maximum speedup per cost, to hardware. During the GreedyPartition(), the implementation of a hardware candidate node n_i is selected at the fastest point of IH_i.

In the second phase, the algorithm iteratively improves the initial partition within in the two nested loops. In the first inner loop, a software node is mapped to hardware while reducing the cost by ReduceCost() procedure, until the number of moved nodes reaches N_{max} which is proportional to the number of nodes. The purpose of the first inner loop is to give more chance of cost reduction during ReduceCost() procedure by allocating more hardware nodes than are needed. In the second inner loop, we select a node from the set of nodes currently mapped to hardware and move it to software. We select a node which will reduce the total hardware cost maximally after the move. Then nodes mapped to software are repeatedly moved to hardware until the performance constraint is met. The implementation of the nodes moved to hardware is selected at the fastest point of IH_i. During this procedure, once a node is moved to the other partition group (hardware to software or software to hardware), it is fixed in order to prevent from being re-selected as a candidate node. After performance constraint is met through the above procedure, ReduceCost() procedure is called in order to reduce the total hardware cost. If total hardware cost reduced by ReduceCost() is less than the best hardware cost obtained so far, current partition is saved as the best partition. Note that the partitioning result by ReduceCost() is saved to *G'* not *G*. One reason for this is that ReduceCost() can change the mapping of a fixed hardware node to reduce total hardware cost and saving the result to *G* could cause problem to the iteration process. Another reason is that accumulating the result by ReduceCost() may prevent *G* from escaping from local optimum in the iterative improvement process. The inner loop repeats the above procedure until all the nodes are fixed, while outer loop repeats the inner loop until no more improvement is available.

4. Experimental Results

The partitioning algorithm with loop pipelining is implemented in C++ under UNIX environment. We use a JPEG encoder which is described in 977 lines of C code as an example. This example is a computation-intensive application which consists of two main loops and 20 tasks. The granularity of partitioning is task-level (i.e. the leaf task), where each task has 4-5 hardware implementation alternatives. Software profiling information is obtained on SPARC 1. We assume a memory-mapped communication model with 2 clocks of communication overhead for 32bit data transfer.

Figure 5 shows design space curves for the example obtained by our partitioning algorithm in the following two different cases:

case 1: partitioning without loop pipelining.
case 2: partitioning with loop pipelining.

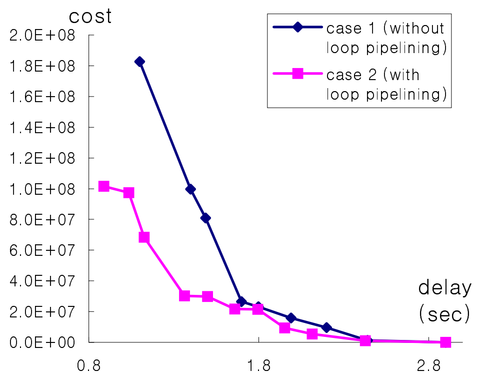


Figure 5. Hardware-software partitioning result for a JPEG encoder.

In case 2, we set the number of pipeline block *P* as 3 for each main loop. From the figure, we can see that the curve of case 2 is always on the left side of that of the case 1. This means that the partitioned system of case 2 is always faster that that of case 1 when they have the same hardware cost. The minimum delay of case 1 is 1.1 with hardware cost of 1.83E+8, whereas that of case 2 is 0.89 with hardware cost of 1.01E+8. The improvement of the minimum delay by loop pipelining technique is about 19%. The improvement of the cost with the delay kept constant is 7 to 70%. This fact explains that loop pipelining for hardware-software partitioning is effective in improving both performance and cost.

To the best of our knowledge, there is no previous algorithm that considers hardware sharing and implementation selection at the same time. Therefore, we compare the result of the proposed algorithm with that of our own simulated annealing algorithm in order to show the effectiveness of our iterative improvement partitioning algorithm. Simulated annealing is an algorithm based on the concept of the probabilistic selection of randomly generated states [8]. We generate a new state by three random moves; toggling hardware-software mapping(*M1*), changing the state of hardware sharing(*M2*), and changing the implementation of a hardware node*(M3*). The generation probability of *M1* is 0.5 while *M2* and *M3* is generate at the same probability of 0.25. The cost function f_A of the annealing is defined as

$$f_A = c_1 \cdot \Delta d + c_2 \cdot hardware\_cost$$

where c_1 and c_2 are weighting factors, and Δd is 0 if the total execution delay is smaller than *D* and is |total_execution_delay-*D*| otherwise. To satisfy the performance constraint, c_1 is scheduled to increase as annealing process proceeds with the cooling ratio of 0.9. Table 1 compares the partitioning result of the proposed algorithm with that of the simulated annealing. We exclude the information on computation time, since our comparison intends to show not the efficiency but the effectiveness of the proposed algorithm. The proposed algorithm is of course much faster (about order of magnitude) than simulated annealing. The results of the simulated annealing are obtained by selecting the best cost after running the program 5 times. From the table, we can see that our algorithm find a solution comparable to that of simulated annealing.

Table 1. Comparison of the proposed algorithm with simulated annealing.

	Our algorithm		Simulated Annealing	
	without loop pipelining	with loop pipelining	without loop pipelining	With loop pipelining
D(sec)	HW cost	HW cost	HW cost	HW cost
3.0	0	0	0	0
2.5	1349888	1118208	1349888	1096478
2.2	9493399	5404288	9493399	5404288
2.0	15751271	9458688	15751271	9842357
1.8	23116680	21457030	22836983	19753436
1.5	80949548	29907451	95992889	28140225
1.4	99761876	30139131	99761876	28218085
1.2	136188746	68323032	135950292	68468184
1.1	182659983	97306200	183178053	97428184
0.9		101499608		104438617

To see the effect of hardware sharing and implementation selection, we test our algorithm without these features. Table 2 shows the results obtained by our algorithm with loop pipelining but without hardware sharing or implementation selection. When we test our algorithm without the feature of implementation selection, we fix the implementation of each node n_i at the median point in the implementation curve $IH_{i.}$ We don't fix the implementation at the slowest point in the implementation curve because we cannot satisfy tight performance constraint by using such an implementation. The results in Table 2 shows that sharing and implementation selection is effective for reducing the total hardware cost of the partitioned system.

Table 2. Partitioning results without hardware sharing or implementation selection

	Without sharing		Without implementation selection (median point)	
D (sec)	HW cost	% cost increase	HW cost	% cost increase
3.0	0	0	0	0
2.5	1380638	23.5	1710449	53.0
2.2	5783996	7.0	6297713	16.5
2.0	9838396	4.0	10352113	9.4
1.8	22230200	3.6	21457030	0
1.5	35561532	18.9	30773958	2.9
1.4	35910490	19.1	31306822	3.9
1.2	78562734	15.0	82082528	20.1
1.1	114549312	17.7	114170208	17.3
0.9	121214978	19.4	118409952	16.7

5. Conclusions

In this paper, we proposed a hardware-software partitioning algorithm with loop pipelining technique which is suitable for computation-intensive applications composed of loops.

Loop pipelining technique is effective for partitioning because this technique increases parallelism within a loop thereby allowing partitioning algorithm to find a better solution. Thanks to increased parallelism, we can find a lower cost solution at given performance constraint and a better performance solution at given hardware cost constraint. In addition, our hardware-software partitioning algorithm, which is based on an iterative improvement method, allows hardware implementation selection and hardware sharing to minimize hardware cost, subject to performance constraint. Experimental results show that (i) loop pipelining technique is effective in that it provides more chance of overlapping the execution of hardware and software during partitioning process and (ii) the proposed algorithm efficiently finds a solution comparable to that of a simulated annealing algorithm.

Future work includes dynamically changing pipeline blocks according to the current partition state, thereby increasing the effect of loop pipelining further. Recall that in the current implementation, we determine pipeline blocks statically before the partitioning process. We are also working on considering multiple processor target architecture for software implementation.

References

[1] J. K. Adams and D. E. Thomas, "Multiple-Process Behavioral Synthesis for Mixed hardware-Software Systems," *Proceedings of International Symposium on System Synthesis*, pp. 10-15, 1995.

[2] C. Chu, et al., "HYPER: An interactive synthesis environment for high performance real time applications," *Proceedings of International Conference on Computer Design*, November 1989.

[3] R. Ernst and J. Henkel, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design & Test of Computers*, pp. 64-75, December 1993.

[4] R. K. Gupta and G. De Micheli, "System-level Synthesis using Reprogrammable Components," *Proceedings of EURO-DAC'92*, pp. 2-7, February 1992.

[5] J. Henkel, T. Benner, and R. Ernst, "Hardware generation and partitioning effects in the COSYMA system," *Proceedings of Int'l Workshop on Hardware-Software Codesign*, pp. 29-40, October 1993.

[6] A. Kalavade and E. A. Lee, "A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware-software Partitioning Problem,", *Proceedings of Int'l Workshop on Hardware/Software Codesign*, pp.42-48, September 1994.

[7] A. Kalavade and E. A. Lee, "The Extended Partitioning Problem: Hardware-software Mapping and Implementation-Bin Selection," *Proceedings of Int'l Workshop on Rapid Systems Prototyping*, June 1995.

[8] S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671-680, May 1983.

[9] P. V. Knudsen and J. Madsen, "PACE: A Dynamic Programming Algorithm for Hardware-software Partitioning," *Proceedings of Int'l Workshop on Hardware/Software Codesign*, pp. 85-92, March 1996.

[10] K. Olukotun, R. Helaihel, J. Levitt and R. Ramirez, "A Software-Hardware Cosynthesis Approach to Digital System Simulation," *IEEE Micro*, pp. 48-58, August 1994.

[11] F. Vahid, J. Gong, and D. D. Gajski, "A Binary-Constraint Search Algorithm for Minimizing Hardware during Hardware-software Partitioning," *Proceedings of EURO-DAC'94*, pp. 214-219, 1994.

[12] M. Edwards, "Software Acceleration Using Coprocessors: Is it Worth the Effort?," *Proceedings of Int'l Workshop on Hardware/Software Codesign*, pp. 135-139, March 1997.

[13] S. Bakshi and D. D. Gajski, "Hardware/Software Partitioning and Pipelining," *Proceedings of Design Automation Conference*, pp.713-716, June 1997.

[14] P. Pjorn-Jorgensen and J. Madsen, "Critical Path Driven Cosynthesis for Heterogeneous Target Architecture," *Proceedings of Int'l Workshop on Hardware/Software Codesign*, pp. 15-19, March 1997.