Journal of Circuits, Systems, and Computers
Vol. 13, No. 6 (2004) 1165–1181
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

AN EFFICIENT EXPLORATION OF MULTIMEDIA ARCHITECTURE USING WEIGHTED PARAMETER DEPENDENCY AND SAMPLED-DATA SIMULATION

YEONG GEOL KIM* and TAG GON KIM†

Department of Electrical Engineering and Computer Science,
Korea Advanced Institute of Science and Technology,
373-1 Guseong-dong, Yuseong-gu, Daejeon, 305-701, Republic of Korea
* *yg.kim@kaist.ac.kr*
† *tkim@ee.kaist.ac.kr*

This paper proposes a system-level design methodology for efficient exploration of parameterized multimedia architecture. The purpose is to find a near-optimal configuration, as far as possible, without performing exhaustive analysis of the design space. This is done through synergistic integration of two independent methodologies, first of which is the multi-stage dynamic optimization based on parameters clustering and sparsing, while the second one being a quick performance estimation through sampled-data simulation of target multimedia application. The experimental results with mediabench show speedup ranging from 186.4 to 339.8, while the corresponding error-to-global optimum ranges from 14.8% to 16.4%.

Keywords: Design space exploration; parameter dependency; sampled-data simulation.

1. Introduction

The demand for high-performance real-time application is increasing due to the growth of computing power and user's desire to new and better applications. One of the representative applications is the multimedia domain. Multimedia now defines a significant portion of the computing market, and this is expected to grow considerably. As described in Ref. 1, for the complex applications like multimedia ones, more flexibility is required to accommodate design errors and specification changes, which may happen at the later design stages. Since an ASIC is specially designed for one behavior, it is difficult to make changes at later stage. In such a situation, ASIP offers a required flexibility at lower cost than general programmable processors. Therefore, we adopted an ASIP as a research target for multimedia processors.

One of the major problems in developing ASIP is that it may result in delay for time-to-market because of large overheads in designing instruction set, special hardware modules, developing compiler, debugger, and so on. Therefore, frameworks for supporting rapid development of ASIP are crucial for multimedia

systems design. A recent innovation addressing this problem is an embedded soft core, a general-purpose processor which has parameterizable components such as Tensilica,[2] ArcCores[3] and HP.[4] Although this parameterized ASIP approach can reduce the design overhead significantly, an exhaustive search of the optimal values of a large number of parameters are still very time-consuming. For example, a full search of 20 parameters, each with three alternatives, leads to about 35 billion configurations. This kind of problem is well known as the combinatorial explosion problem in the global optimization research field.[5]

To rapidly find a near-optimal configuration from such a large parametric design space, this paper proposes a very efficient search technique. This is done through integrating two independent methodologies synergistically, first of which is the multi-stage dynamic optimization based on parameters clustering, while the second one being a quick performance estimation through sampled-data simulation.

Previous work towards finding globally (sub)optimal configurations for parameterized architectures include Refs. 6 and 7. The goal of Ref. 6 is to find a sub-optimal configuration for energy-delay product of a memory hierarchy without performing an exhaustive analysis of a parameters space. It applied *separate* optimization for an instruction cache and a data cache which were assumed to be almost independent of each other. This paper reported a large optimization speedup with respect to full search and the found solution has about 10% distance from the true optimum. The main restriction of this work is that it was only applied to a very simple and modular cache sub-system. Therefore, the quality of the found solution is questionable if the method is applied to more general and complex design parameters in a practical situation.

On the contrary, Ref. 7 provided correct pareto-optimal configurations by exploiting parameter dependency. This paper showed a significant speedup against the full search when many architectural parameters are independent, as in the case of simple parameterized system-on-chip design. However, if parameters are highly interdependent as in the case of ASIP, this approach inevitably reduces to almost a full-search algorithm, as will be described later. The proposed methodology is an extension of these previous works for achieving large speedup and finding acceptably good configuration, so that it can be employed across various processor architectures including ASIC and SoC as well as ASIP for multimedia.

Among several choices of architecture for targeting high-performance and real-time multimedia applications, we selected the superscalar architecture as a research vehicle for several reasons. Firstly, it is still one of the most popular and verified processor architecture frequently used for high-performance applications. Secondly, there comes out lots of variant architecture solving limitations of traditional superscalar architecture so that the proposed framework can be extended to be used even in the future.[8,9] Thirdly, we could verify our idea very quickly and convincingly with the most famous and stable research-purpose SimpleScalar tool-set,[10] which supports the parameterized superscalar architecture.

The rest of the paper is organized as follows: Sec. 2 describes the dynamic programming approach for rapid search of near-optimal configurations, while Sec. 3 proposes a weighted PD-graph based optimization scheme. Section 4 reviews the estimation scheme[11] focusing on characteristics relevant to this paper, and Sec. 5 shows the seamless integration of these two techniques. After showing the experimental results in Sec. 6, we conclude in Sec. 7.

2. Dynamic Programming Approach to ASIP Synthesis

Dynamic programming[12] is a mathematical procedure designed primarily to improve the computational efficiency of mathematical programming problems by decomposing into smaller sub-problems. The usual application of dynamic programming entails breaking down the problem into *stages* at which the decisions take place and finding a *recurrence* relation that takes us backward from one stage to the previous stage. Before formulating our problems with dynamic programming, let us review key idea of parameter dependency.[7]

2.1. *Parameter dependency*

A parameter p_j is dependent on the parameter p_i, if and only if the changing of value p_i affects the *optimal* value of p_j. Otherwise, a parameter p_j is independent of the parameter p_i. Figure 1 shows an example of parameter p_2 being dependent upon p_1 (1(a)), and one that p_4 being independent of p_3 (1(b)). Every mapping (v_i, v_j) in this figure represents the optimal value of p_j, given the value of p_i as v_i, is v_j. The parameter dependency graph, abbreviated as *PD-graph*, is a directed graph in which a node represents the parameter while an edge represents the dependency between two parameters. Therefore, this graph is a collection of parameter dependencies for all combinations of parameters. Following the definition of the

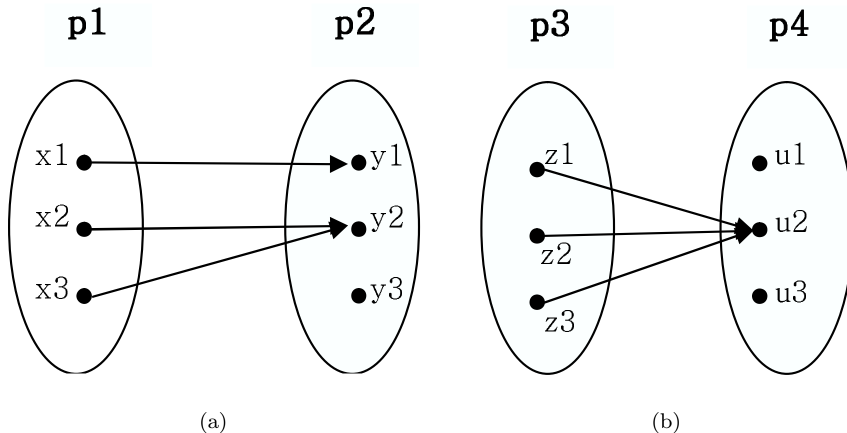

Fig. 1. Parameter dependency.

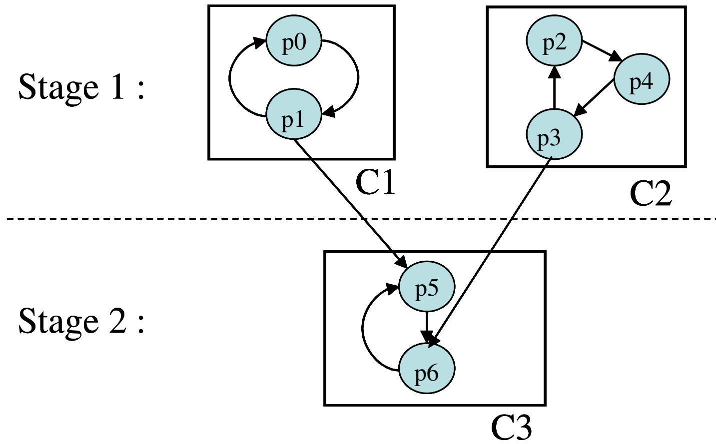

Fig. 2. Clustering PD-graph for dynamic programming.

parameter dependency, we can determine the optimization sequence for a given PD-graph.

- An edge from node p_i to p_j indicates that the optimal configuration of p_j should be calculated only after the optimal configuration of p_i is computed.
- More generally, a path from p_i to p_j indicates that the optimal configuration of all the nodes from p_i to p_j, residing on the path, are calculated, *in that order*.
- If there is an edge from p_i to p_j and an edge from p_j to p_i, the optimal configuration of parameters p_i and p_j must be calculated *simultaneously*.
- More generally, a path from p_i to p_j and back to p_i, which forms a cycle, indicates that the optimal configuration of all the parameters on the cycle need to be calculated simultaneously.

The first two cases show the example of the multi-stage optimization by which the number of calculations is reduced, while last two show the full-search case.

2.2. *PD-graph-based dynamic programming*

To apply the dynamic programming scheme for a given PD-graph, a *cluster* is defined as *maximal* strongly connected sub-components of a PD-graph. After all the clusters of the PD-graph are found, these are topologically ordered for correct optimization sequence as shown in Fig. 2. It is easy to observe that there can be no cycles between clusters due to its definition. As a result, the *tree* composed of clusters make it easy to employ dynamic programming. Followings are some formal notations regarding to the dynamic programming.

Definition 1. Parameter dependency function
$pd : P \times P \to \{0, 1\}$ is a parameter dependency function such that $pd(p_1, p_2) = 1$ if there is a dependency from p_1 to p_2, and $pd(p_1, p_2) = 0$ otherwise.

Definition 2. Parent function
$parent : P \rightarrow 2^P$ is a function such that
for $p_j \in P$, $parent(p_j) = \{p_i | pd(p_i, p_j) = 1 \wedge p_i \in P\}$.
The parent function for a parameter, say p_j, is a set of parameters p_i that affects the optimal value of p_j.

The stages shown in Fig. 2, for dynamic programming, are determined by the topology of the clusters as follows.

Definition 3. Stage function
$stage : C \rightarrow N$ is a function such that returns the stage number for any cluster of a given PD-graph, where C is a set of clusters of a given PD-graph, and N is a set of natural numbers.

$$stage(c_i) = \begin{cases} 0\,, & \text{if } parent(c_i) = \emptyset\,, \\ \max\{stage(c_j) | c_j \in parent(c_i)\} + 1\,, & \text{otherwise}\,, \end{cases}$$

where the *parent* function is a cluster-version of Definition 2.

Definition 4. Cluster-optimum configuration
A cluster-optimum configuration for a cluster $c_i \in C$, notated as $c_i^* = (p_{i1}^*, p_{i2}^*, \ldots, p_{in}^*)$, is defined to be the best configuration to which any other configuration $(p_{i1}, p_{i2}, \ldots, p_{in})$ is inferior in terms of the performance metric value, where parameter p_{ij}, $j = 1, \ldots, n$ is a member of the cluster c_i.

Definition 5. Optimal state
An optimal state, also called stage-optimum configuration for a stage $s_i \in S$, notated as $s_i^* = c_1^* | c_2^* | \cdots | c_n^*$ is a concatenation of all cluster-optimum configuration, where $stage(c_i)$ are same.

Definition 6. Global-optimum configuration
A global-optimum configuration notated as $g^* = c_1^* | c_2^* | \cdots | c_n^*$ is a concatenation of all cluster-optimum configuration.

To obtain the global-optimum configuration from each cluster-optimum, it is important to keep the optimization sequence. The recurrence relation between stages is that the optimal state of ith stage s_i^* can go back far to that of jth stage, s_j^*, where $j = min\{stage(c_j) | c_j \in \cup\, parent(c_i) \wedge stage(c_i) = i\}$. To make things simpler, we only need to calculate the cluster-optimum configuration in the order of increasing stages to find the global-optimum. Equation (1) shows the speedup obtained by applying this multi-stage dynamic programming scheme.

$$Speedup_{MS} = \frac{\prod_{i=0}^{\#clusters} F_i}{\sum_{i=0}^{\#clusters} F_i}\,, \tag{1}$$

where the subscript MS means multi-stage.

In this equation, the term F_i denotes the number of simulations required for *full-search* of all configurations *within cluster* c_i. This is also the speedup provided

by the previous work,[7] called *PDO — Parameter Dependency based Optimization*, against full-search over all parameters of the PD-graph without clustering. The problem in terms of the achievable speedup with this *PDO* approach is that the speedup rapidly decreases as the number of clusters decreases, i.e., each cluster has lots of parameters, since the term F_i rapidly increases. Therefore, when the target architecture has highly inter-dependent parameters, we cannot expect large speedup with this approach. Furthermore, it is difficult and time-consuming to construct a correct PD-graph. More specifically, it is difficult to verify that a parameter being independent of the other due to the completeness problem. Moreover, a problem caused by defining *strict* parameter dependency is that only one exceptional case of a parameter being dependent upon another *declares* the dependency. Therefore, it may become practically unuseful, especially for complex architecture, because the resultant PD-graph would be so dense that many parameters form a cycle, i.e., cluster.

The proposed idea for increasing speedup of Eq. (1) is two-fold. First idea is to increase number of clusters by weighting all arcs in the PD-graph and eliminating arcs of weak strength to make the PD-graph sparser than original one. Second idea is to decrease the F_i term by conducting sampled-data simulation rather than full-data one. These two ideas are more specifically described in the next two section, respectively.

3. Weighted PD-Graph and Sparsing

This section describes the natural extension of the conventional PD-graph that can be helpful to solve the practical problems mentioned before. As previously said, the extension is done through attaching every arc of the PD-graph real value ranging from 0 to 1. Some formal definitions and notations are shown below.

Definition 7. Context function
$context : P \times P \rightarrow C$ is a function such that
for any p_i, $p_j \in P$ that satisfies $p_i \rightarrow p_j$, $context(p_i, p_j) = parent(p_j) - \{p_i\}$.
Simply said, a context is a set of all parent parameters that affects the optimal value of p_j except for p_i.

Definition 8. Weighted PD-graph: wPDG
A weighted PD-graph is a directed graph, wPDG $= (V, A, w)$ where
V: A set of vertices representing architectural parameters
A: A set of arcs such that $(v_i, v_j) \in A$ if and only if $pd(v_i, v_j) = 1$.
$w : A \rightarrow (0, 1]$. A weighting function that is associated to every arc in the graph such that $w(v_i, v_j)$ represents the probability of $pd(v_i, v_j) = 1$.

Figure 3 explains the concept of weight for parameter dependency with context. The point here is that the parameter p_j may be dependent on p_i or not depending on what values are assigned to each parameter $c_k \in context(p_i, p_j) = \{c_1, c_2, \ldots, c_n\}$.

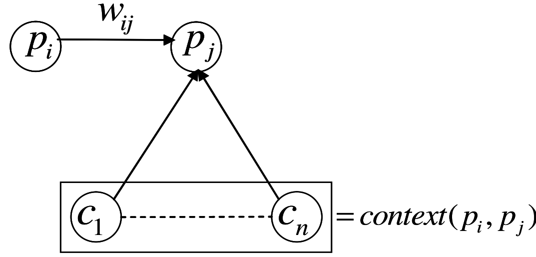

Fig. 3. Weight of parameter dependency defined by context.

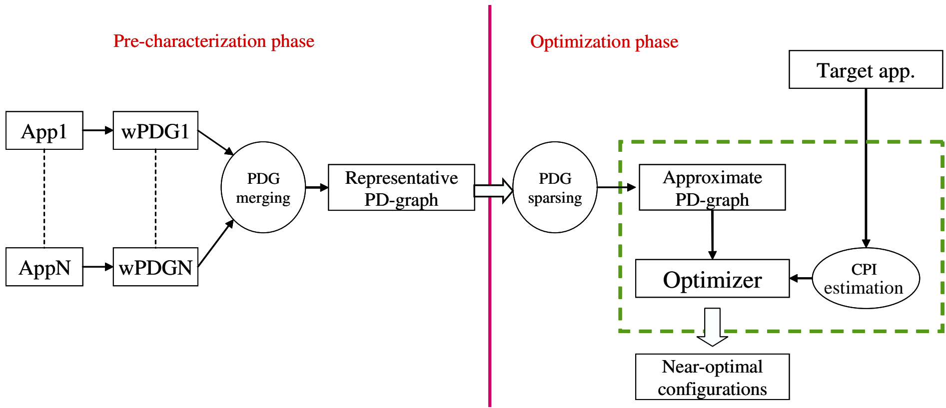

Fig. 4. Overall optimization flow.

This is because parameters of $context(p_i, p_j)$ also affect the optimal value of p_j as well as p_i.

Then, the weight w_{ij} representing the strength of parameter dependency $p_i \rightarrow p_j$ is determined empirically through large number of experiments with a set of representative benchmarks in pre-characteristic phase of Fig. 4

$$w_{ij} = \frac{\#(p_i \rightarrow p_j)}{context_size},$$

where $\#(p_i \rightarrow p_j)$ denotes the number of occurrences of $pd(p_i, p_j) = 1$ and $context_size = c_1 \times c_2 \times \cdots c_n$ denotes the total number of alternatives for context configuration. Note that, if there is only one parent for p_j, say p_i, $context(p_i, p_j) = \emptyset$. In this case, w_{ij} reduces to simply 1 or 0 according to p_j depending on p_i.

Now we can describe the overall optimization flow as shown in Fig. 4. The proposed optimization procedure is a two-phase approach: (1) pre-characterization phase at which representative PD-graph is obtained with a set of representative benchmarks, e.g., mediabench for multimedia and communications; (2) optimization phase at which near-optimal configuration for *target* application is found by sparsing the representative PD-graph and employing CPI estimation scheme.

Note that this optimization procedure is based on an assumption that PD-graph for target application is *similar* to the representative PD-graph obtained at pre-characterization phase. This means that weight distribution is preserved across the applications, if they have similar characteristics as in the case of different applications of same multimedia domain. We will justify this assumption with the experimental results in Sec. 6.

3.1. *Sparsing weighted PD-graph*

The representative PD-graph is obtained at pre-characterization phase by merging a set of weighted PD-graphs $\{wPDG_1, wPDG_2, \ldots, wPDG_n\}$ corresponding to the set of representative applications $\{app_1, app_2, \ldots, app_n\}$. The sparsing criterion in Fig. 4 specifies the condition with which the representative PD-graph is made sparse. There may be different kinds of sparsing criteria depending on the underlying sparsing algorithm, and this is one of the factors that affects the overall speedup and accuracy. We selected a simple sparsing criterion, called *tolerance factor*, that specifies the threshold with which all arcs having smaller weight are eliminated. The effect of such elimination is to increase the number of clusters as shown in the left part of Fig. 6. In this example, let us assume that the tolerance factor is larger than the weight of arc (p_4, p_3), which is w_5, and smaller than those of all other arcs in the PD-graph. Note that the original cluster encompassing parameters p_1 to p_5 is divided into two clusters $cluster_1 = \{p_1, p_2, p_3\}$ and $cluster_2 = \{p_4, p_5\}$ with the elimination of the arc (p_4, p_3). In this way, PD-graph sparsing contributes the overall optimization speedup.

4. CPI Estimation for Multimedia Applications

Our recent work on the cycle counts estimation is described in detail in Ref. 11. After briefly describing the overall characteristics of this approach, we will focus on the part relevant to the integration with wPDG-based dynamic optimization. Reference 11 proposed a very accurate and relatively fast method of estimating cycle counts of target applications to rapidly find architecture parameters that satisfy user-provided real-time constraints. Furthermore, by giving a tight upper bound on the estimation error, user can convince himself of the estimation result.

The proposed method is based on an assumption that the program structure does not change even though the architecture configuration changes. This assumption holds for our framework thanks to the nature of the run-time instruction scheduling feature of superscalar architecture. In other words, there is no need to recompile the benchmark program even when the parameters' values change. Furthermore, the number of each basic block being visited remains constant across parameter value change because the control flow of the program does not depend on the machine configuration. To summarize, the number of execution counts for each block is architecture-independent information. On the contrary, cycle counts taken for executing each basic block is surely architecture-dependent. For example, cycle

Table 1. Experimental results of the proposed estimation scheme.

Desired error			5%		10%		20%	
Apps.	Exec. time loops (%)	# loops	speedup	avg. error (%)	speedup	avg. error (%)	speedup	avg. error (%)
JPEG	98.44	278	3.63	1.61	3.78	1.72	4.00	2.14
GSM	96.97	27	1.22	0.63	9.92	1.67	15.04	1.16
G.721	99.78	10	436.6	0.72	505.08	1.50	667.43	2.89
RASTA	91.84	108	4.09	0.29	5.87	0.85	5.98	2.1
EPIC	98.07	183	8.16	0.84	29.49	0.97	34.07	1.9
ADPCM	98.87	1	64.88	0.72	70.77	1.87	77.85	6.55
Average			86.1	0.8	104.15	1.43	134.06	2.79

counts taken for executing each basic block varies with the change of parameters, such as the number of functional units, branch prediction strategy, decode/issue width, and so on.

The speedup comes from the fact that we need only one full-simulation to obtain the architecture-independent information instead of conducting simulation whenever parameters change. Moreover, we can reduce the simulation time in obtaining architecture-dependent ones through sampled-data simulation with a little loss of accuracy. The experimental results with Mediabench[13] show 86 times of speedup against full data simulation with 0.8% estimation error on average, if the desired error bound is set to 5%. When we set the desired error bound to 20%, speedup increases to 134 with 2.8% estimation error on average.

Finally, there is a popular saying that most programs spend 90% of their execution time in 10% of the code.[14] While the actual percentages may vary, it is often the case that a small fraction of a program accounts for most of the running time, especially, a loop that contains no other loop is called an inner loop. These loops make execution of the same basic block over and over again. Therefore, we need not repeatedly simulate the same basic blocks again and again to obtain the execution cycle counts of basic blocks. In general, it can be obtained with negligible error by sampled data simulation, as our experimental results show.

It should be noted that multimedia applications are naturally more loop-intensive than general application. This fact implies that we can obtain significantly accurate estimated values with small-sized sampled-data. Table 1 shows the resultant speedup and statistics about loops for every application of mediabench.

4.1. *Adaptive sample size determination*

Since sampled-data size is closely related to the speedup and estimation error, sample size determination strategy is important. As the size of the sample gets larger, the estimation error is reduced with the sacrifice of speedup. Moreover, the size of sample satisfying the desired error bound is different depending on

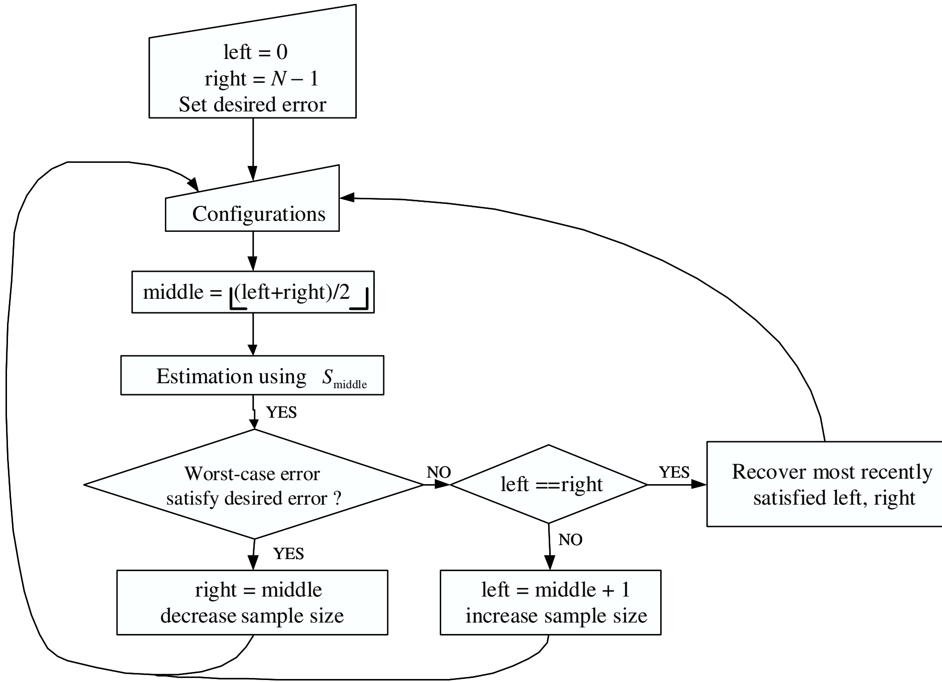

Fig. 5. Adaptive sample size determination.

applications. Therefore, we propose an efficient heuristic to determine the sample data size. We define the sample size set as follows:

$S = \{S_k | 0 < k < N, S_{k-1} < S_k\}$,
S_k: size of kth sample data,
N: number of admissible sample data.

First of all, designer need to prepare the sample size set which is divided into equal space or user-defined. For example, if there is a 512-KB input data, a designer can divide it into equal space with 0.1 KB. In this case, the number of element becomes 5120 in the sample size set. However, for some application such as EPIC, the sample size must be in the power of two. In this case, designer needs to prepare only feasible sampled-data set. After this sampled-data set is given, we use binary search to determine best-fit size of sample data through the design space exploration. Figure 5 shows the estimation flow incorporating the sample-size decision process. Initially, *left* is assigned to be the smallest sample size while *right* assigned to be the largest sample size. Meanwhile, the user provides a desired *tolerable* error about estimation result.

Note that the purpose of this procedure is trying to make the sample size converge to the best-fit one, as soon as possible, where the speedup is maximized while maintaining the estimation error under the desired bound. This can be practically

important because even a single target application would require lots of simulations by changing the parameter configuration, in which case the gross speedup over the design space exploration would be largely dependent upon the chosen size of sampled-data.

During design space exploration, the sample size will decrease by reducing *right* to *middle*, if the worst-case error of the estimation result satisfies the desired error. We can then use half of the previous sample size to estimate the total execution cycles. Finally, the size of sample data will converge to specific size when *right* equals to *left*. From then on, this sample size is fixed to explore design space. However, if the desired error is not satisfied on a particular processor configuration, we recover *left* and *right* to the most recent values that satisfied the desired error.

4.2. *Sample factor and estimation error bound*

A sample-data size determined by the above procedure is used to calculate the speedup in terms of simulation time for one configuration. *Sample factor* is defined as the ratio of *number of committed instructions* between them, because it is the number of instructions that directly affects the simulation time.

$$sample_factor = \frac{NCI_{\text{sample}}}{NCI_{\text{original}}}, \tag{2}$$

where NCI is abbreviation for the *number of committed instructions*.

As briefly mentioned before, one of the key benefits of the proposed estimation scheme is that it provides a tight bound for the actual CPI value, and the *tightness* of the bound can be controlled by the user's *desired error bound*. In other words, we can obtain the *interval* for a given configuration in which the actual CPI value resides. Therefore, we can safely use the reported bound to compare any pair of configurations to decide which one is superior/inferior to another. The detailed schemes of calculating the bounds are omitted, but interested user can refer Ref. 11.

5. Embedding Estimation Scheme into PD-Graph

This section describes the overall optimization method wherein the proposed estimation scheme is exploited to find the cluster-wise optimum configuration. The process of finding optimum configuration involves lots of comparison between every pair of possible configurations, in which the estimation results are used. As previously described, the estimation results on the CPI is of the form (min, max), and there arise two distinct cases comparing two configurations, as shown in the right side of Fig. 6.

The first case is that two intervals do not overlap. In this case, we can tell which configuration is inferior/superior to the other, implying that sampled-data simulation suffices for the comparison. On the contrary, the second case shows that the intervals overlap which requires full-simulation for correct comparison. The key point here is that the proposed estimation scheme generally provides tight bound for the

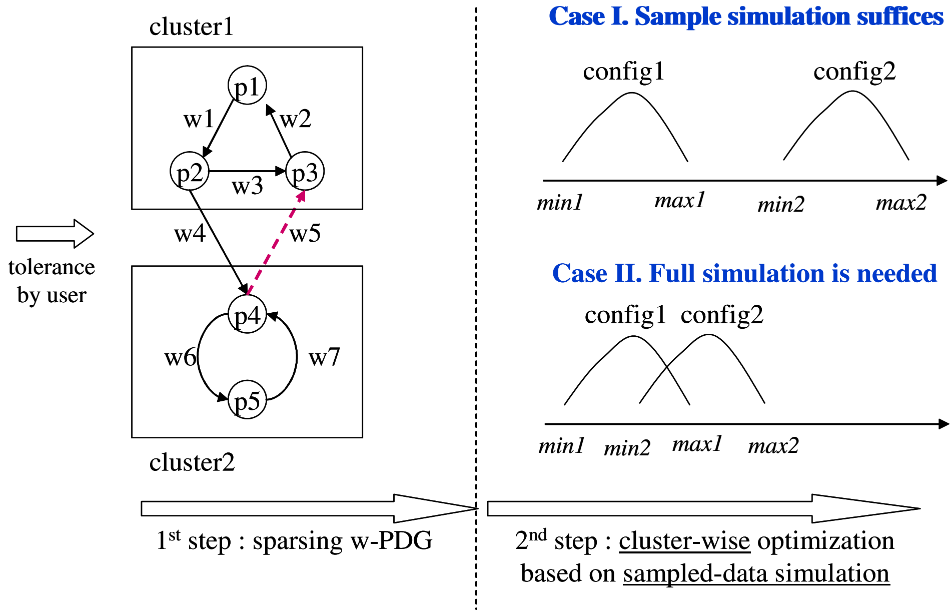

Fig. 6. Interval-based comparison with estimation results.

interval with relatively short sampled-data size. This means that (1) full-data simulation is not often required, and (2) the time required for sample-data simulation is much shorter than full-data one. These two desired properties are synergistic in terms of speedup against conventional full-search with full-data simulation in obtaining near-optimal configurations for the target application. The benefit is effectively reduce, the required number of full-data simulations as following equation says.

- effective # of full simulation = # of overlapping cases + sample_factor * # of nonoverlapping cases.

For example, if the # of full-data simulation is 1000 and there were 100 cases of overlapping in comparing configurations with sample factor of 10%, then the effective # of full simulation reduces to $100 + 0.1 * 900 = 190$, which leads to speedup over five times. It is remarkable that such a speedup-effect is *multiplied* with the speedup resulting from wPDG-based dynamic programming.

6. Experimental Results

As previously mentioned, we used the Simplescalar simulator framework[10] to evaluate the speedup and accuracy of the proposed algorithm. We selected Mediabench[13] as target applications because it is most popular and well-organized benchmarks for various media applications, ranging from signal and image processing to cryptography. This experiment used the cycle-per-instruction (CPI) as a performance

Table 2. Selected simplescalar parameters.

Parameter	Range
mem:lat	(9, 1) or (9, 2)
mem:width	8, 16
res:memport	1, 2, 4
fetch:ifqsize	2, 4, 8
decode:width	2, 4, 8
issue:width	2, 4, 8
commit:width	2, 4
fetch:mplat	1, 2, 3
ruu:size	4, 8, 16
lsq:size	4, 16
Num. of alternatives	11664

metric assuming the situation the designer wants to find a high-performance configuration as soon as possible. Since considering whole parametric design space of the Simplescalar simulator is not possible for the experiment with available computation facility, we selected ten important parameters affecting the performance characteristics. These parameters with their admissible range of values are shown in Table 2.

Figure 7 shows a representative PD-graph for gsm application, with its sparsed version together. The representative PD-graph was obtained by merging six weighted PD-graphs out of seven applications in the mediabench suite. The only one application excluded in making the representative PD-graph at pre-characterization phase is the very gsm application. The dotted lines represent the arcs eliminated, by sparsing the original PD-graph, with tolerance factor of 1.0%. In other words, the weights of the dotted lines are smaller than 0.01. Note that the remaining subgraph after eliminating the dotted arcs is sparsed PD-graph for gsm application. We can observe in this figure that the number of clusters increased from 1 to 4 by sparsing. Four shaded nodes in the figure represent three clusters, while the remaining (unshaded) nodes form the largest cluster as follows:

- 1st cluster = {fetch:mplat},
- 2nd cluster = {commit:width},
- 3rd cluster = {ruu:size, lsq:size},
- 4th cluster = {mem:lat, mem:width, issue:width, decode:width, fetch:ifqsize, res:memport}.

Table 3 shows the experimental results after the proposed optimization algorithm was applied. The 2nd to 4th column and 6th to 8th column shows the result when the tolerance factor is set to 0%, 1%, and 2%, respectively. Since all applications equally have five rows of same information, the first five row is explained.

- *# clusters*: number of clusters after sparsing the original PD-graph with corresponding tolerance factor,

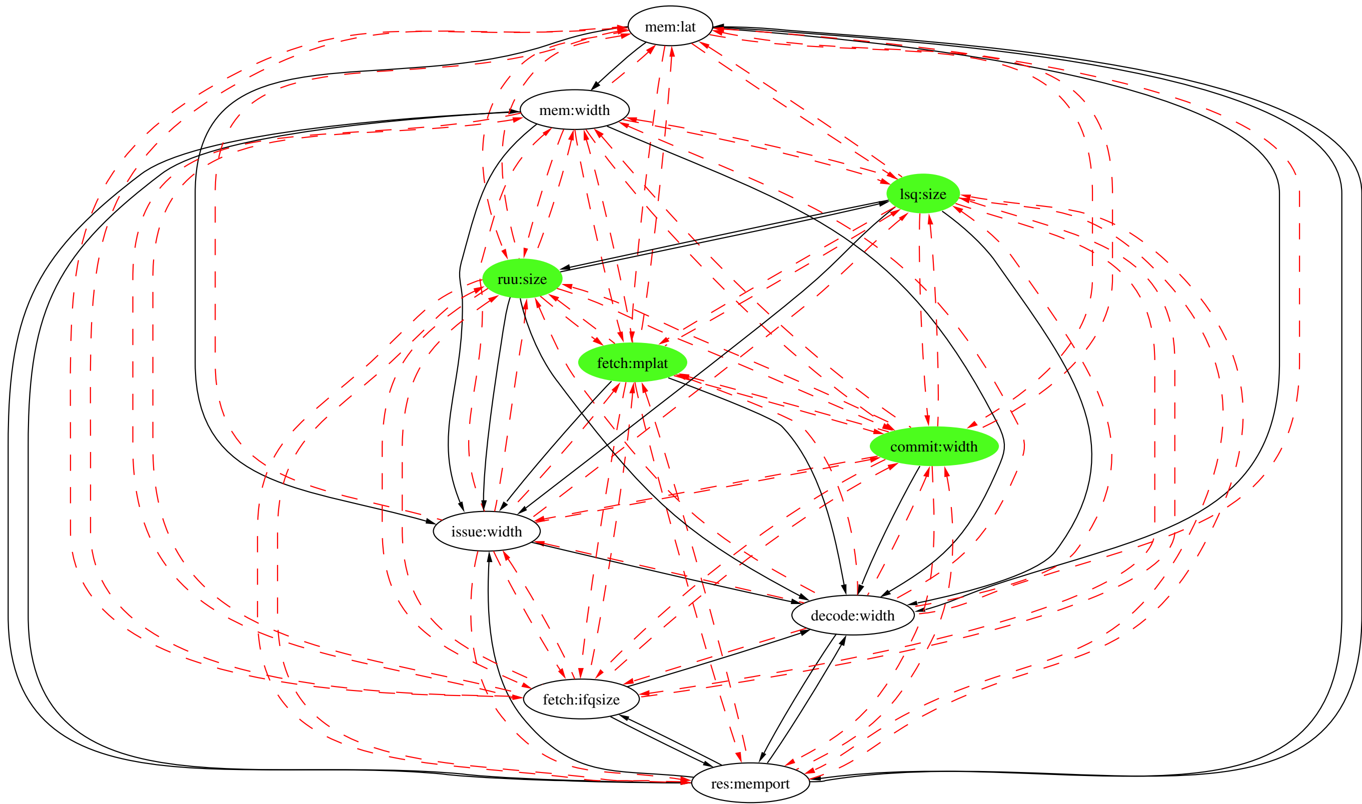

Fig. 7. Weighted PD-graph for GSM with tolerance = 0.01.

Table 3. Optimization results with mediabench.

adpcm	0%	1%	2%	**jpeg**	0%	1%	2%
# clusters	1	5	6		1	4	6
rank	0	590 (5.1%)	561 (4.8%)		0	720 (6.2%)	720
est. optimal		0.682 (22.2%)	0.6818 (22.1%)			0.5546 (17.6%)	
speedup	**18.2**	**153.4**	**303.9**		**8**	**99.1**	**297.6**
s_factor	**0.042**	**b, w**	**0.53,1.21**		**0.12**	**b, w**	**0.45, 1.05**
epic	0%	1%	2%	**pegwit**	0%	1%	2%
# clusters	1	5	8		1	7	9
rank	0	864 (7.4%)	0		0	650 (5.6%)	1310 (11.2%)
est. optimal		0.5762 (17.8%)				0.6513 (8.0%)	0.682 (10.9%)
speedup	**47.3**	**382.6**	**767**		**1**	**266.3**	**459.2**
s_factor	**0.016**	**b, w**	**0.47, 1.04**		**0.8**	**b, w**	**0.56, 1.65**
g.721	0%	1%	2%	**rasta**	0%	1%	2%
# clusters	1	5	6		1	5	7
rank	0	0	0		0	3178 (27.2%)	4278 (36.7%)
est. optimal						0.828 (41.7%)	0.8526 (45.7%)
speedup	**71.9**	**228.5**	**333**		**11.3**	**130.8**	**395.4**
s_factor	**0.0008**	**b, w**	**0.53, 1.17**		**0.0625**	**b, w**	**0.57, 1.19**
gsm	0%	1%	2%				
# clusters	1	4	8				
rank	0	452 (3.9%)	455				
est. optimal		0.5883 (7.4%)	0.5884 (7.4%)				
speedup	**1.6**	**44.3**	**478**				
s_factor	**0.6**	**b, w**	**0.54, 1.15**				

- *rank*: the rank of the found solution after optimization procedure is finished (among 11 664 configurations),
- *est. optimal*: the *estimated* optimal value reported by the proposed optimization algorithm,
- *speedup*: speedup against full search with full-data simulation,
- *s_factor*: sample factor for the application (ratio of sampled-data simulation to full-data one),

Table 4. Averaged results for mediabench.

tolerance (%)	avg. speedup	avg. ranking (%)	opt. distance (%)
0	22.8	0	0
1	186.4	7.9	16.4
2	339.8	9	14.8

- (b, w): abbreviation for (best, worst) meaning best and worst CPI value for the application, obtained with full-data simulation with full search (done only for evaluating the accuracy of the proposed algorithm).

The % within blank in the table represents the normalized value by 100. Several interesting facts can be observed from this table.

(1) # of clusters increased rapidly with tolerance factor,
(2) the speedup increased with tolerance factor in a similar pattern,
(3) all applications have found the global optimum configuration *without* sparsing (tolerance factor of zero),
(4) Without sparsing and sampling, there would be no speedup at all in applying the optimization. This is the major restriction of the previous work[7] when applied to complex core.

Table 4 is a summarized version of Table 3 in which the speedup and accuracy is averaged for all applications. In this table, it is remarkable that the result with tolerance of 2% is better than that with 1% for all aspects. This implies the fact that the proposed algorithm needs to be enhanced for best expected results, which is one of our on-going researches.

7. Conclusions

This paper proposed a system-level design methodology for efficient exploration of the parameterized ASIP. This is done through synergistic integration of two independent methodologies, first of which is the multi-stage optimization based on weighted parameter dependency, while the second one being a quick performance estimation through sampled-data simulation. The experimental results with mediabench applications show speedup ranging from 186.4 to 339.8 with corresponding distance-to-optimum values ranging from 14.8% to 16.4%. Although the experiment was done on somewhat general superscalar architecture provided by SimpleScalar environment, it is expected that the presented methodology can be applied to the multimedia-specific architecture without any change of the basic idea. Further works include experiments on a carefully designed multimedia-specific core offering variations for the special addressing mode, address generation unit, special functional unit, and so on. Another work is to integrate the energy estimation capability into the proposed framework.

References

1. H. Choi, I.-C. Park and S. H. Hwang, Synthesis of application specific instructions for embedded DSP software, *Proc. ICCAD*, San Jose, California, November 1998, pp. 665–671.
2. J. Turley, Tensilica CPU bends to designers' will, *Microprocessor Rep.* **13** (1999) 12.
3. Technical summary of the ARC core, ARC Cores Ltd. (2000), http://www.arccores.com.
4. P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli and F. Homewood, Lx: A technology platform for customizable VLIW embedded processing, *Proc. ISCA*, Vancouver, Canada, June 2000.
5. M. C. Fu, Simulation optimization, *Proc. WSC*, Arlington, VA., December 2001, pp. 53–61.
6. W. Fornaciari, D. Sciuto, C. Silvano and V. Zaccaria, A design framework to efficiently explore energy-delay tradeoffs, *Proc. CODES*, Copenhagen, Denmark, April 2001, pp. 260–265.
7. T. Givargis, F. Vahid and J. Henkel, System-level exploration for pareto-optimal configurations in parameterized system-on-a-chip, *IEEE Trans. VLSI* **10** (2002) 416–422.
8. V. V. Zyuban, Inherently lower-power high-performance superscalar architectures, Ph.D. thesis, Notre Dame University (2000).
9. S. Palacharla, N. P. Jouppi and J. E. Smith, Complexity-effective superscalar processors, *Proc. ISCA*, Denver, Colorado, June 1997, pp. 206–218.
10. D. Burger and T. M. Austin, The SimpleScalar tool set, Version 2.0, *Computer Architecture News*, June 1997, pp. 13–25.
11. S. B. Jee, Y. G. Kim and T. G. Kim, A fast, accurate and reliable estimation for rapid design space exploration of superscalar architecture, *Proc. SPECTS*, Montreal, Canada, July 2003, pp. 585–592.
12. H. A. Taha, *Operations Research — An Introduction*, 5th edn. (Macmillan Publishing Company, 1992).
13. C. Lee, M. Potkonjak and W. H. Mangione-Smith, MediaBench: A tool for evaluating and synthesizing multimedia and communications systems, *Proc. MICRO*, Research Triangle Park, NC., December 1997, pp. 330–335.
14. J. D. Ullman, *Compilers — Principles*, *Techniques*, *and Tools* (Addison-Wesley, 1986).

Copyright of Journal of Circuits, Systems & Computers is the property of World Scientific Publishing Company and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.