A Fast Instruction Set Evaluation Method for ASIP Designs

Angela Yun Zhu[1], Xi Li[1], Laurence T. Yang[2], and Jun Yang[1]

[1] Department of Computer Science
University of Science and Technology of China
Hefei, Anhui 230027, P.R. China
{yukiyun, stuart}@mail.ustc.edu.cn, llxx@ustc.edu.cn
[2] Department of Computer Science
St. Francis Xavier University
P.O. Box 5000, Antigonish, B2G 2W5, NS, Canada

Abstract. ASIPs are designed specifically for a particular application or a set of applications. Their instruction sets must be carefully tailored to provide high performance as well as to meet non-functional constraints such as silicon area and power consumption. Traditionally, evaluation of different candidate instruction sets is all carried out through simulation. However, the growing design complexity and time-to-market pressure have rendered simulation increasingly infeasible. In this paper, we present an instruction level modeling method that can rapidly evaluates several important aspects of a selected instruction set. Experimental results show that we can prune a large number of candidate instruction sets with the model, accelerate design space exploration and alleviate the pressure on simulation.

1 Introduction

Application Specific Instruction set Processors (ASIPs) are in between custom architectures such as Application Specific Integrated Circuits (ASICs) and commercial programmable processors such as General Purpose Processors (GPPs). ASIPs typically consist of a configurable base processor core and a base instruction set, plus the capability to extend the instruction set. The goal of ASIP design is to optimize performance for an application domain while minimizing the area and energy costs.

One of ASIP design approaches is language-driven design space exploration, based on Architecture Description Languages (ADLs) [2] [3][4]. An ADL specification is used to generate a software toolkit including compilers, simulators, assemblers, etc.. Design space exploration is then performed with these tools. During the instruction set design of an ASIP, first, some pre-designed extensible instructions are chosen as candidates from a provided library. Then, instruction set tailoring and extension according to a specific application domain is conducted with the help of the toolkit generated. Research done on the instruction set design for ASIPs include [8][9][10].

Traditionally, simulation approach is used throughout the ASIP designs, including the process to optimize hardware parameters, instruction matching to meet the functional requirements of the specific applications, and estimating the non-functional constraints such as size and power for the ultimate synthesizable instruction set. However, the long simulation time often gets in the way of the time-to-market, rendering the design-simulate-analyze methodology not feasible. Our method aims at reducing simulation when evaluating the instruction set, meanwhile keeping the concern on non-functional parameters such as power and size.

The remainder of this article is organized as follows: In Section 2, we outline our technical approach. Section 3 introduces our basic data structures and Section 4 elaborates on our method. In Section 5, we present our preliminary experimental results and analysis. Section 6 concludes our work.

2 Technique Outline

Fig. 1 shows the process of instruction selection and instruction set extension. The following is a corresponding explanation:

1. Application specifications in a high-level language (e.g. C, C++) and the pre-designed extensible instruction library in HDL are translated into an intermediate representation (IR) respectively. This is called the *translation phase*. We use *Data Flow Graph (DFG)* as our intermediate representation.
2. During the *instruction matching phase*, we try to cover the DAGs representing the dataflow of each basic block in an application by DFGs representing the dataflow of instructions selected from the instruction template library. This results in covered basic blocks and a candidate instruction set.
3. Construct the *Instruction Parameter Table (IPT)*, which reflects the estimated parameters of each single instruction.
4. Construct the *Instruction set Evaluation Model (IEM)*, with which we are able to rapidly evaluate overall cost of the *candidate instruction set.*
5. Evaluate the instruction set using *IEM*, under system design constraints.
6. Use *IEM* to help the instruction set optimization (instruction clustering). Instruction extensions are specified with the architecture description language *xpADL* [16], and corresponding items are added to *IPT*. Reconstruct *IEM* and repeat *5*, *6*, until the evaluation results are fair.

Representing dataflow of basic blocks and template instructions with DFGs is trivial. And we adopt the instruction matching algorithm described in [6] to do the initial DFG covering of basic blocks. In the following sections, we will focus on our methods according to step 3 through step 6 stated above.

3 Instruction Parameter Table *IPT*

In this section, we introduce a static resource model called the *Instruction Parameter Table (IPT)*, which specifies the execution and design parameters of

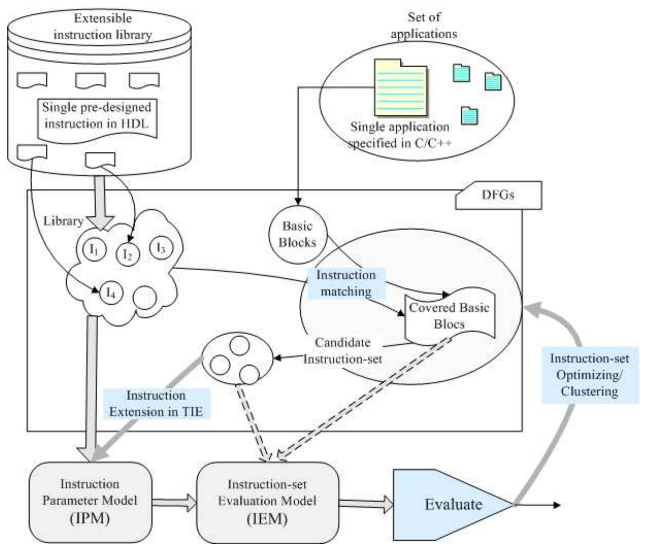
Extensible instruction library
Single pre-designed instruction in HDL
Set of applications
Single application specified in C/C++
DFGs
Library
Basic Blocks
Instruction matching
Covered Basic Blocs
Candidate Instruction-set
Instruction Extension in TIE
Instruction-set Optimizing/ Clustering
Instruction Parameter Model (IPM)
Instruction-set Evaluation Model (IEM)
Evaluate

Fig. 1. Instruction set design for an ASIP

every individual instruction. The table will be further used to construct the *Instruction set Evaluation Model (IEM)*, which aims to evaluate several aspects of a selected instruction set.

Definition 1 *An* ***Instruction Parameter Table (IPT)*** *is a table to describe the evaluated cost of a single instruction execution, where*

– $\mathcal{I}$ *is a set of* instructions*,*
– *for each* $I \in \mathcal{I}$*,*
  • cc (I) *is the clock cycle needed to execute instruction* I *on given hardware;*
  • area (I) *is the lut number* [3] *of units to implement an extended instruction* I*;*
  • power (I) *is the evaluated power consumption for the execution of instruction* I*;*

Values of the first two parameters can be obtained with some tools. We generate Verilog code for each instruction and put the codes through the Synplicity tools [1] for the timing (clock cycle) and area (lut number) numbers. Our ASIP design is based on a pre-fabricated processor core, thus the pre-designed instructions will not cause additional area on our chip. That is, the area size constraint is only relevant when we need to generate a new instruction. This is indicated in our *Instruction set Evaluation Model* later.

[3] We use the *lut* number because our experiment environment is based on FPGA. The *gate* number would be used **if synthesize to ASIC.**

Instruction level power evaluation techniques are used to get the third parameter in the table. With the popularization of embedded systems, low power design has become one of the most challenging tasks in system development. A lot of the techniques for system-level power evaluation have been proposed in the past [12][13][14]. In [17], we presented a novel two-level power estimation model, including a microarchitecture level model and an instruction level model. The microarchitecture level power model is based upon the structure of the components, and the instruction level model is based upon the microarchitecture model. Thus, the proposed methodology provides us with an accurate and rapid model to evaluate high level embedded system design. We get our *power(I)* parameter from this evaluation model.

4 Instruction Set Evaluation

Definition 2 *We use an* ***Instruction set Evaluation Model (IEM)*** *to fast evaluate a candidate instruction set for a specific application. An* IEM *includes four estimation formulae of the candidate instruction set $\mathcal{W}$ in terms of* power consumption, execution time, area needed for instruction extension, *and* code size, *formularized as:*

$$Area(\mathcal{W}) = \sum_{I \in \mathcal{W}} area(I); \tag{1}$$

$$Size(\mathcal{W}) = length(I) \times |T_W|; \tag{2}$$

$$Time(\mathcal{W}) = SPN_{delay}(transform(W)); \tag{3}$$

$$Energy(\mathcal{W}) = \sum_{I \in \mathcal{W}} power(I) \times t(I) \times count(I). \tag{4}$$

Here, *t(I)* in formula (4) is execution time needed to execute an instruction *I*. It can be got directly from *cc(I)*, which is defined in *IPT* together with *area(I)* and *power(I)*. Note that $area(I) = 0$ if I is a pre-designed instruction, so the total area cost is the number of luts we needed for adding some new kinds of instructions in an instruction set. *count(I)* is the number of times instruction I appeared in the instruction flow of the application execution. It will be explained a little later.

Fig. 2(a) shows the data flow of a basic block, which is covered by instructions selected. Take each instruction as a node we get a DAG (Directed Acyclic Graph) W in Fig. 2(b). The node number of W, $|T_W|$, shows the number of instructions in a compiled application code segment. $length(I)$ is the length of instructions for a pre-decided instruction set architecture, which is always an invariable. Thus, formula (2) tells that the code size is decided by both the length of every instruction and the number of instructions in a compiled application code segment.

4.1 Execution time estimation

In our fast evaluation model, we use the DAG (Fig. 2(b)) to evaluate the relevant performance of execution time. Two problems must be considered first. One is

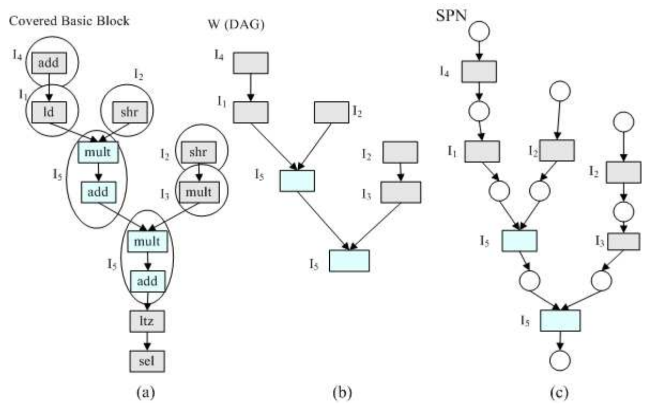

Fig. 2. An example

the iteration times of each basic block. The other one is how the structure of the graph, those concurrency and dependency, affect the pipelining of the actual instruction flow.

We use two vectors to help illustrate our evaluation method, one is θ, the other is φ. For each node t in DAG W, if $t \in B$, B is a basic block, then $\theta(t) = \theta(B)$ is the iteration times of B, and $\varphi(t) = \varphi(B)$ is the parallelability of B, which means the execution time of t can be reduced to $\frac{t}{\varphi(t)}$ by pipelining. So the whole execution time can be expressed as:

$$\sum_{B \in W} \frac{\sum_{t \in B} t_{delay}}{\varphi(B)} \times \theta(B) = \sum_{t \in W} \frac{t_{delay} \times \theta(t)}{\varphi(t)}. \tag{5}$$

For $\theta(B)$ in formula (5), we use executive without compiling optimization of the application as the input to an instruction set simulator *ISAsim* [4], and get the execution frequency θ of basic blocks with the profiling module of *ISAsim*. Although the profiling of this vector parameter based on a simulation technique, our evaluation method can still be helpful and faster because we only need the same θ when evaluating among different instruction set with the same application.

Another problem is to get the actual execution time of each basic block. This is complex due to the pipeline behavior. However, the greatest parallelism we can get from pipelining will be constrained by the dependency in the DAG (flow graph). We approximate the actual pipelined execution time of the basic block with timed Petri net. This is reasonable for two sakes: First, architecture relative optimization can't be performed before instruction set is decided and our methods just consider the maximum possibility of parallelism; Second, we just want to compare between different instruction selections, thus we don't

[4] *ISAsim* is part of *xpSIM* generated by our *xpADL*, we will briefly introduce them later.

need actual precise clock cycles, Proper approximation is enough for evaluating preference.

Definition 3 *A **Shifted Petri net (SPN)** is an extended timed Petri net[15] with following differences from* Petri net*:*

- *At any time, each place can have at most one token in it.*
- *For every transition* t*, there exist a transition delay which represents the execution time of the function associated to the transition. Formally,*

  $$\forall t \in T, \exists t_{delay} \in \mathcal{R}_0^+,$$

  with $\mathcal{R}_0^+$ being the set of non-negative real numbers.
- *Each token* k *holds a time stamp k_{time}.*
- *When a transition* t *is fired, the marking* M *will generally change by removing all the tokens from the pre-set $^\circ t$ and depositing one token into each element of the post-set t°, and*

  $$k_{time} = \max_{j \in J}\{j_{time}\} + t_{delay}, \forall k \in K,$$

  where J is the set of tokens in $^\circ t$, and K is the set of tokens in t°

Algorithm 1: *TRANSFORM*

```
Input: DAG $W = (T_{TG}, E_{TG})$; $IPT$.
Output: $SPN = (P, T; F, M_0)$ .
for each $t' \in T_{TG}$
      $count(I) = count(I) + \theta(t')$ ;
      $t = new_t$; $t = t'$;
      $t_{delay} = \theta(t') \times lookup_{IPT}(t', cc)$;
      put $t$ into $T$;
      if $^\circ t' = \emptyset$ {
         $p = new_p$; put $p$ into $inP$;
         $f = new_f$; put $f = (p, t)$ into $F$; }
      else if $t'^\circ = \emptyset$ {
         $p = new_p$; put $p$ into $outP$;
         $f = new_f$; put $f = (t, p)$ into $F$; }
end for
for each $e = (t'_1, t'_2) \in E_{TG}$
      $p = new_p$; put $p$ into $P$;
      $f = new_f$; put $f = (p, t)$ into $F$;
      $f = new_f$; put $f = (t, p)$ into $F$;
end for
$P = P \bigcup inP \bigcup ourP$;
$M_0 = P \rightarrow 0$
return $SPN = (P, T; F, M_0)$;
```

Algorithm 2: *GET* *SPN_{delay}*

```
Input: $SPN = (P, T; F)$ .
Output:
  execution time of the $SPN$ $SPN_{delay}$
for each $p \in inP$
      $k = new_{token}$; $k_{time} = 0$;
      put $k$ into $p$; $M_0(p) = 1$;
end for
while ($\exists t \in T$, $t$ is $enabled$), do
      $fire(t)$;
end while
$SPN_{delay} = \max_{p \in outP}\{k_{time}(p)\}$;
return $SPN_{delay}$;
```

Formula (3) according to Algorithm 1 and algorithm 2 is based on this approximation. Algorithm 1 describes how to transform a DAG into a Shifted Petri

Net. Vector θ is bound to each t in line 4 and 6. So line 4 keeps the accumulation for the instruction frequency *count*(I), which is used in our power evaluation; and line 6 takes the block frequency into consideration of whole execution time as the numerator on the right of formula 5. Algorithm 2 uses firing rules of timed Petri net to evaluate the execution time with instruction level parallelism to incarnate the parallelability $\varphi(t)$ on the right of formula 5.

4.2 Power consumption

Formula 4 uses *count(I)* get from Algorithm 1 and simply adds all the power consumption estimation value of each instruction used in the application to get the total power consumption of application implementation with the candidate instruction set. From the view of pipeline, this is not correct since power consumption of a single instruction are different between being *stall* and being *normal*, i.e., in pipeline, power consumption of an instruction is related to its previous instructions executed, deploy policy, even related to the memory system of the CPU. Thus, estimating the running power consumption of an application is difficult. However, the instruction level power consumption parameters we get from [17] is an average normal running values, which dissipates the power consumption of *stall* in to every instruction in the power model. The experiment in [17] also shows that the relevant error compared to the result of *SimplePower*[5] is less than 10%.

4.3 xpADL driven develop environment

Our fast evaluation method is part of our system-level develop platform for ASIP. We have explored a design environment driven by an architectural description language called *xpADL* [16]. It can generate a tool kit used for DSE including rechargeable simulators and compilers. In the method of this paper, two tools are used. One is the simulator *xpSIM* mentioned in block frequency profiling. Another tool we used from *xpADL* is *xpSYN*. During instruction set optimization, extended instructions are specified in *xpADL*, and relies on the synthesizer *xpSYN* to generate an efficient hardware implementation, often with the Verilog code. The codes will be further put through the Synplicity tools [1] to obtain instruction information such as timing and area. We add these information into our *Instruction Parameter Table (IPT).*

5 Experimental Results and Analysis

5.1 Experimental setup

We conduct a case study to demonstrate the effectiveness of our approach in the following steps:

First, use *PISA* [7] instruction set as pre-designed instruction library. Describe it using xpADL, and generate a simulator *xpSIM* as reference. Then, *IPT* construct is done by using *xpPower* in [17] with $2.5V$, $50MHz$ to get instruction-level power evaluation, and by using *Synplify Pro* map to *Xilinx xc2s200* to get instruction clock cycle (and we also use it later to get the luts number of extended instructions). Add different instructions to the original instruction set $\mathcal{W}$, get candidate instruction sets and evaluate them using *IEM*. Take *susan_smooth* program for example, after some profiling, we get three candidate instruction set $\mathcal{W}1$, $\mathcal{W}2$, and $\mathcal{W}3$ by adding fabricated instructions $a*b+c$, $a+b+c$, and $a*b+c*d$ respectively. Some evaluation parameter values are shown in Table 1. Lastly, the above steps are iterated for different instruction set extensions.

Table 1.

IS	Area (luts)	Size (bytes)	Time (cc)	Energy	Total Instr_count
$\mathcal{W}$	0	184160	42241680	326697097	61533089
$\mathcal{W}1$	543	173184	41592513	317305112	57666965
$\mathcal{W}2$	62	179584	42221870	325052831	59864311
$\mathcal{W}3$	1054	184128	42231774	326584738	61525951

5.2 Results and analysis

We apply our fast evaluation method on several different instruction set extension, then analyze the rationality of our evaluation results, also compare the time evaluation to our *xpSIM* generated from xpADL. We find that time evaluation results differ a lot from simulation. However, the trend of which instruction set can make application faster is almost the same. For other evaluation factors, there is no exercised comparative standard, however, they are consisted to the system designers' empiricism. (e.g. Fig. 3 for *susan_smooth* from Table 1).

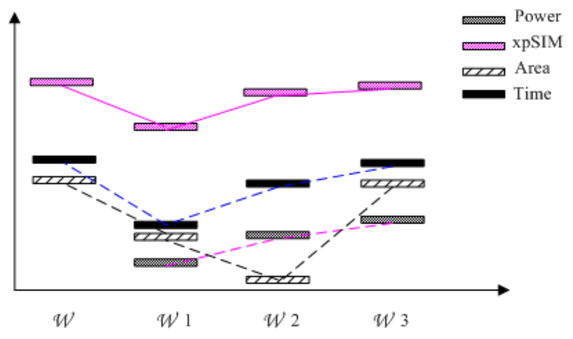

Fig. 3. Compared evaluation results

$$E_{\mathcal{W}} = \frac{1}{Power(\mathcal{W})^{\beta_P} \times Size(\mathcal{W})^{\beta_S} \times \left(C + Area(\mathcal{W})^{\beta_A}\right) \times Time(\mathcal{W})^{\beta_T}}. \tag{6}$$

Formula (6) combines the evaluation parameters in IEM and indicates when an instruction set is more superior. Fig. 4 is $E_{\mathcal{W}}$ of four code segment on four instruction-set respectively, with β all set to 1. It indicates that instruction set $\mathcal{W}1$ might be the best choice for *susan_smooth*. Actually, we noticed that, the instruction $mult + mflo$ appeared with a frequency about 6.283% in the final execute flow, while $addiu + addu$ appeared with about 2.712% and instructions in the form of $a * b + c * d$ appeared with about 0.0116%. On the other hand, instruction extension with form $a * b + c$ is not suitable for *blowfish*. This is because there is not *mult* instruction in *flowfish*.

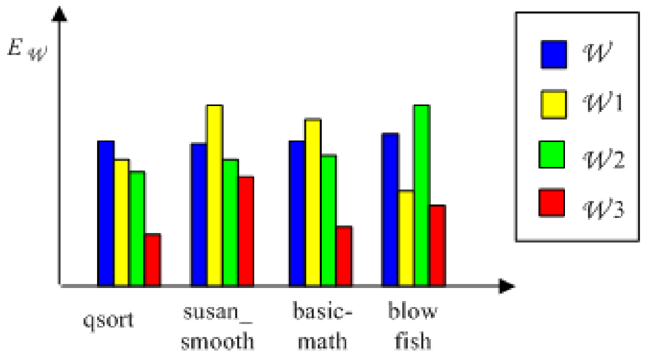

Fig. 4. Efficiency of different instruction sets

6 Conclusions

In this paper we propose a method for fast evaluating the instruction set targeted towards a certain application, with several design constraints. We use the Instruction level Parameter Table and an Instruction set Evaluation Model to compare different candidate instruction sets concerning their power consumption, code size, chip area and execution time. Our model is not as precise as the simulation techniques, but it has three important advantages. First, when handling a large number of candidate instruction sets, our method fast pre-prunes most of them, reducing the dependence on simulation (which is always the bottleneck in the design space exploration). Second, while simulators only focus on performance, our evaluation method also provides an overall view of the instruction set design. Lastly, we use Petri net in our model to get time evaluation. The flow graph representation of Petri net eases the application profiling since loops and concurrency are explicitly illustrated. This is much better than analyzing the instruction flow provided by a simulator.

For future work, we try to make our model more precise and applicable to a broader range of instruction set architectures.

References

1. *http://www.synplicity.com/ (Synplify and Synplify Pro Reference Manual).*
2. V. Zivojnovic, S. Pees, and H. Meyr. *LISA - Machine Description Language and Generic Machine Model for HW/SWCo-Design.* In Proceedings of the IEEE Workshop on VLSI Signal Processing. San Francisco, CA, USA. pp.127-136. October 1996.
3. A. Halambi and P. Grun. *EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability.* In Proceedings of Conference of Design, Automation, and Test Europe. Munich, Germany. pp. 100-104. March 1999.
4. M. Freericks. *The nML Machine Description Formalism.* Department of Computer Science, Technology University of Berlin, Berlin, Germany, Technical Report. 1991.
5. N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye. *Energy driven integrated hardware-software optimizations using simplepower.* In Proceedings of the 27th Annual International Symposium on Computer Architecture. Vancouver, British Columbia, Canada. pp. 95-106. June 2000.
6. R. Kastner et al.. *Instruction Generation for Hybrid Reconfigurable Systems.* ACM Transactions on Design Automation of Electronic Systems, vol. 7, no. 4. Apr. 2002.
7. C. Vieri. *The pendulum instruction set architecture (PISA)* MIT Reversible Computing Project Internal Memo., May 1996.
8. J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. *Instruction set definition and instruction selection for ASIPs.* In Proceedings of the 7th International Symposium on High-Level Synthesis. Ontario, Canada. pp. 11-16. May 1994.
9. M. Arnold and H. Corporaal.. *Designing domain specific processors.* In Proceedings of the 9th International Workshop on Hardware/Software Codesign. Copenhagen, Denmark. pp. 61-66. April 2001.
10. P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. *Lx: A technology platform for customizable VLIW embedded processing.* In Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-00). Vancouver, Canada. pp. 203-213. June 2000.
11. C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, *An instruction-level functionality-based energy estimation model for 32-bits microprocessors.* In Proceedings of the Annual ACM/IEEE Design Automation Conference (DAC-00). Asia and South Pacific. pp. 346-351. June 2000.
12. M. Barocci, L. Benini, A. Bogliolo, B. Ricco, and G. De Micheli. *Lookup table power macro-models for behavioral library components.* In Proceedings of the Design, Automation and Test of Europe, Paris, France. pp. 173-181. February 1998.
13. E. Macii, M. Pedram, and F. Somenzi. *High-level power modeling, evaluation, and optimization.* IEEE Transactions on CAD of Intergrated Circuits and Systems. vol. 17, pp. 1061-1079. November 1998.
14. J. Luo, L. Zhong et al. *Register binding based RTL power management for control-flow intensive designs.* IEEE Transactions on CAD of Intergrated Circuits and Systems. vol 23, No. 8, pp. 1175-1183. Augest 2004.
15. T. Murata. *Petri Nets: Properties, Analysis and Applications.* Proceedings of the IEEE. 77(4), pp. 541-580. 1989.
16. X. Li, X.H. Zhou, et al. *xp-ADL: A key issue in software and hardware codesign.* In Proceedings of 2002 Conference of Open Distributed and Parallel Computing Symposium (DPCS-02). Wuhan, China. pp. 100-104. 2002.
17. X. Li, Z.G. Wang, et al. *Study on system-level power model for low power optimization.* Acta Electronica Sinica. vol 32, no. 2, pp. 205-208. February, 2004.