System-level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-chip

Tony Givargis
Center for Embedded Computer Systems
University of California, Irvine, CA 92697
givargis@cecs.uci.edu

Frank Vahid
Department of Computer Science & Engineering
University of California, Riverside, CA 92521
vahid@cs.ucr.edu
Also with the Center for Embedded Computer Systems at UC Irvine

Jörg Henkel
C&C Research Laboratories, NEC USA
4 Independence Way, Princeton, NJ 08540
henkel@ccrl.nj.nec.com

Abstract

In this work, we provide a technique for efficiently exploring the configuration space of a parameterized system-on-a-chip (SOC) architecture to find all Pareto-optimal configurations. These configurations represent the range of meaningful power and performance tradeoffs that are obtainable by adjusting parameter values for a fixed application mapped onto the SOC architecture. Our approach extensively prunes the potentially large configuration space by taking advantage of parameter dependencies. We have successfully incorporated our technique into the parameterized SOC tuning environment (Platune) and applied it to a number of applications.

Keywords

System-on-a-chip, parameterized architectures, configurable platforms, embedded systems, system-level exploration, low-power system design, platform tuning.

1. Introduction

The growing demand for portable embedded computing devices is leading to new system-on-a-chip (SOC) architectures intended for embedded systems. Such SOC architectures must be general enough to be used across several different applications, in order to be economically viable, leading to recent attention to parameterized SOC architectures. Different applications often have very different power and performance requirements. Therefore, these parameterized SOC architectures must be optimally configured to meet varied power and performance requirements of a large class of applications.

A typical SOC architecture will have numerous cores, including a processor core, one or more caches, numerous on-chip buses, on-chip memory, and a large number of peripheral cores that provide application specific functionality such as data encoding, decoding and communication. Each of these SOC cores is likely to be parameterized, enabling a designer to *tune* a core's settings for a specific application that is to be mapped on the SOC architecture. For example, the on-chip buses may be configured to optimally use bus-invert 0 coding for low power, or the caches may be configured to use a greater or lesser degree of associativity for increased performance [2]. An assignment of a value to each of these parameters will impact the overall performance and power consumption of the SOC architecture. However, such impacts are highly dependent on the particular application running on the SOC. Therefore, a designer must have a method for finding a feasible set of parameter values, referred to as a *configuration* of the SOC, that meets the specification requirements. We outline an exploration approach that efficiently searches the entire configuration space and outputs Pareto-optimal configurations, thus providing the designer with only the interesting configurations that result in a tradeoff between power and performance.

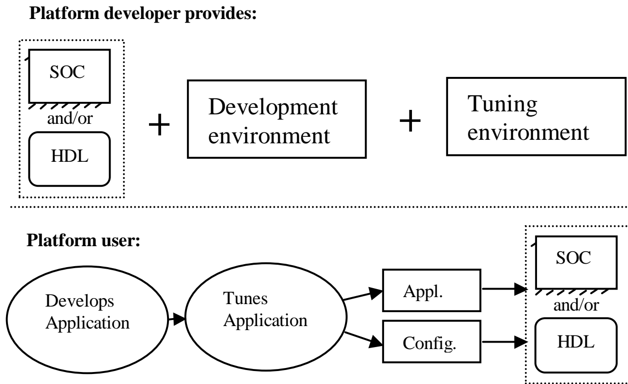
Our exploration algorithm fits in an SOC design flow as follows. As depicted in Figure 1, the SOC provider provides a parameterized architecture in HDL or configurable IC format along with all the traditional development tools such as compilers, debuggers, emulators, etc. In addition, the SOC provider provides a system-level tuning environment. This tuning environment enables the SOC user to search the parameter space of the SOC and find a configuration that meets power and performance requirements of the target application. This tuning environment is the focus of this work.

The remainder of this paper is organized as follows. In Section 2, we describe some related work. In Section 3, we state the parameterized SOC exploration problem and outline our approach for solving it. In Section 4, we give some experimental results. In Section 5, we state our conclusions.

2. Previous Work

Much previous work has focused on power evaluation of SOC architectures at various levels of abstraction. Circuit-level approaches simulate the circuit at the transistor level while monitoring supply current [3][4]. Logic-level, or gate-level, approaches simulate a gate-level design, and calculate power by considering switching activity of nodes in the design [5][6], executing orders of magnitude faster than circuit-level approaches at the expense of some accuracy. RTL (register-transfer level) power evaluation operates at an even higher-level of abstraction, modeling power consumption of more abstract circuit components, such as adders and multipliers etc. Simulation is performed at the RT-level and power is obtained by using these power models, also known as macro-models. The approach taken here can be divided into two categories, macro-modeling using table-lookup techniques and analytical modeling [7]. Lookup-tables and the coefficients of the analytical models are often derived from the gate-level circuit structure or lower-level power evaluation and simulation. RTL power evaluation, in some publications such as [8], is shown to be accurate to within 5% of actual power consumption. Behavioral-level approaches seek to estimate power of a behavioral HDL description before a synthesized design is obtained. An abstract notion of physical capacitance and switching activity is used. Switching is estimated using entropy from circuit input to circuit output by quadratic or exponential degradation [9][10].

Figure 1: SOC design flow.

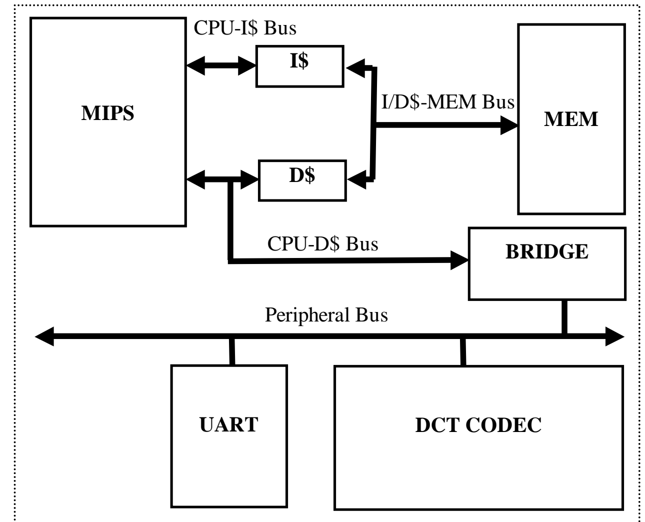


Work has been done to evaluate power consumption of a particular type of core, like microprocessors. One approach, instruction-level power modeling, is proposed by [11]. Given a program execution trace, energy is computed as the sum of the energy consumed by each instruction that is executed, circuit state energy consumed when a particular instruction is followed by another, and energy consumed by other effects such as stalls and cache misses. This approach is sped up in [12], by deriving a shorter program trace that results in equal power dissipation when compared to the original trace. In [13] a mathematical generic power model for 32-bit microprocessors is proposed. The approach classifies the instruction set into classes, like branches. The model has been applied to various 32-bit processors. Other researchers have focused on fast system-level models for cache, memory and bus power consumption [14][15], consisting mostly of simulators coupled with equations that compute power consumption as a function of usage/traffic and core parameters. Further approaches aim at estimating the power consumption of whole embedded systems. In [16], a cycle-accurate power simulation tool, for an embedded system using a strong ARM architecture as CPU, is introduced. The reported results are accurate within 5% compared to measurements conducted on a hardware board. A trace-based approach deploying a mix of analytical models (for instruction cache, data cache and main memory) and ISSs (instruction set simulators) is introduced in [17].

A closely related methodology to ours, named SPADE (System level Performance Analysis and Design space Exploration), is proposed by [18]. That work defines a general scheme for the design of programmable architectures, referred to as the Y-chart. Target applications are mapped onto the architecture, and their performance is analyzed to obtain performance numbers. (The *architecture*, *applications* and *performance numbers* represent the dimensions along the Y shaped chart, hence the name Y-chart.) After analysis, the architecture or applications are tuned and the process is repeated until a desired system is obtained. In this model, the tuning process is driven manually by the designer. In our work, we outline an approach to automate the exploration, for a restricted range of architectures.

Previous work has focused on techniques that quickly and accurately simulate SOC architectures in order to obtain power and performance metrics. Our technique combines this work with an approach for efficiently exploring the configuration space of SOC architectures by pruning configurations that are guaranteed to be inferior to others already evaluated.

Figure 2: Target SOC.



3. Target SOC

Our parameterized system-on-a-chip architecture is depicted in Figure 2. The architecture consists of the following. A MIPS R3000 processor and instruction and data caches communicate over two processor-local buses. The on-chip memory is connected to the two caches via another bus. Universal Asynchronous Receiver and Transmitter (UART) and DCT CODEC cores are connected to the peripheral bus, which is bridged to the processor's data bus. Most of the cores in this architecture are statically configurable. The MIPS can be set to run at 5 different voltage/frequencies. Caches can be set to use different associativity, line-size and total size. Each of the four busses can be configured with a different data and/or address bus width and one of binary, gray or bus-invert encoding. The UART core's transmitter and/or receiver buffer sizes can each be set to one of 5 different sizes. The DCT CODEC core's pixel resolution can be set to one of 10 or 12 bits.

To evaluate power and performance of a particular application running on our target SOC, the tuning environment utilizes an available power and performance measuring approach such as in-circuit emulation, gate-level simulation, RTL simulation or a system-level behavior approach. We have used our platform-tuning environment (Platune) which is a system-level behavior approach, as described next.

The evaluation environment, called Platune, is an executable model of the target architecture that is depicted in Figure 2. This simulation model is augmented with power models to allow for measuring the average power consumption of the chip while running an application. Platune can be broken down into two components, namely, the simulator module and power analyzer module. A detailed description of these components can be found in [19] and a brief summary is given here.

Platune is a tightly coupled collection of event driven cycle accurate simulation models of its various components, namely, processor, cache, memory, busses and peripheral cores. The processor simulator maintains detailed statistics on its internal activity, e.g., fetches, stalls, instruction execution frequency, register file access, floating-point activity, etc. Such statistics is used in a post simulation

Figure 3: Exhaustive Pareto-optimal computation.

```
list compute_Pareto_configurations(space s) {
   list all, pareto;
   float min_power = 1e100; /* infinity */
   for each configuration c in space s {
      simulate_SOC(c); all.push(c);
   }
   all.sort( /* key is execution time */ );
   while( !all.empty() ) {
      c = all.pop();
      if( c.power < min_power ) {
         min_power = c.power; pareto.push(c);
      }
   }
   return pareto;
}
```

analysis to compute power and performance metrics. The cache simulator of Platune is a fully parameterized element that operates on a stream of memory references that is output from the processor. In addition to the standard cache metrics, such as number of misses, the Platune cache simulator maintains additional activity statistics, e.g., number of tag comparisons, word-line activity and bit-line activity, etc. that is later used for power computation. Like the cache simulator, the bus simulator in Platune also operates on a stream of data and memory references that are generated by the processor, cache and memory modules and accumulates bus wire bit toggle statistics that are later used for power computation. Peripherals are handled in a scheme similar to that of the processor core.

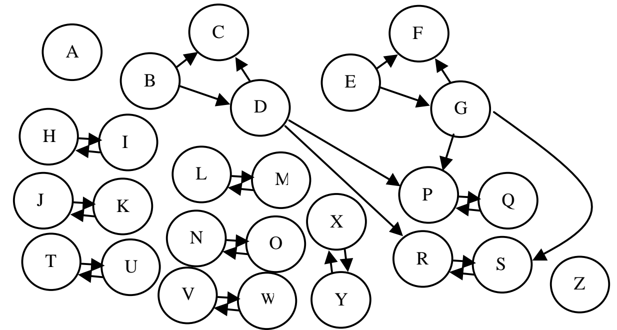
The second component of Platune, the power model and analyzers, operate on the statistics that are gathered during simulation, as described here. For the processor an instruction based power modeling approach is used [11]. For caches, first a structured (physical) model is deduced based on the cache parameter settings and technology feature size. This allows for estimation of bit-line, word-line, comparator, storage transistors, and address decoding logic capacitive loads. Then, switching activity from the simulation phase is applied to obtain average power consumption of the cache. Similarly for each bus segment, a rough layout is inferred that is based on the chip technology, chip area, bus widths, and relative size of the various cores, in order to obtain the average bus capacitance. Then, switching activity from the simulation phase is applied to obtain average power consumption of various buses.

The entire Platune environment is integrated into a single GUI application and comes with a C compiler as well as a small runtime kernel for use by the application that is being simulated. Overall accuracy of Platune is experimentally shown to be 5% to 15% of gate-level measurements [19].

4. Approach Overview

We will next state the problem and outline our solution. Our solution will be given by first looking at an exhaustive method. Then we state the key observation that makes our approach more efficient, followed by the actual modeling and solution of the problem.

Figure 4: Target SOC dependency graph.



	A	B,C,D	E,F, G	H,I,J, K	L,M, N,O	P,Q, R,S	T,U, V,W	X,Y	Z
Core	CPU	I$	D$	CPU-I$ Bus	CPU-D$ Bus	I/D$-MEM Bus	Peripheral Bus	UART	DCT CODEC
Param	V	Size line assoc	Size line assoc	D/A-width code	D/A-width code	D/A-width code	D/A-width code	T/R-Buff size	Pixel-Resl.

4.1 Problem Formulation

We are given a system-on-a-chip architecture composed of numerous interconnected parameterized computational, communication and memory elements. We enumerate each of these parameters as P_1, P_2, P_3 … P_n. Each of these parameters can be assigned a value from a finite set of values. A complete assignment of values to all the parameters is a configuration. The problem is to efficiently compute, with the aid of a system-level model, the *Pareto-optimal configurations*, with respect to power and performance, for a fixed application executing on the SOC. In our problem, a configuration is Pareto-optimal if no other configuration has better power as well as performance.

4.2 An Exhaustive Solution

An exhaustive solution to this problem can be achieved as follows. Power and performance are evaluated for all configurations. Configurations are sorted by non-increasing execution time (performance). Then, in the sorted order, a walk through the space is performed while all configurations that result in power consumption above the minimum seen thus far are eliminated. The remaining configurations are Pareto-optimal. The algorithm is given in Figure 3.

The problem with this approach is that the configuration space is likely to be very large, making the approach impractical in many cases. The exhaustive approach is practical when applied to a small subset of the solution space consisting of one or two varying parameters while all others held constant.

Fortunately, we have found that many parameters in an SOC have little dependency among each other. Two parameters are dependent if changing the value of one of them impacts the optimal parameter value of the other. For example, it may be that the associativity and line size parameters of the instruction cache are dependent. However, the associativity parameter of the instruction cache and the line size parameter of the data cache are independent.

Figure 5: Exploration algorithm.

```
list explore_Pareto_optimal_configurations(graph g) {
   list sub_graphs = compute_connected_components(g);
   list pareto;
   // part 1
   for each graph g in sub_graphs {
      pareto=compute_Pareto_configurations(g.space)
      eliminate configs. in g.space not in pareto;
   }
   // part 2
   while( !sub_graphs.size() != 1 ) {
      g1 = sub_graphs.pop_front();
      g2 = sub_graphs.pop_front();
      g = g1 union g2;
      sub_graphs.push_back(g);
      pareto=compute_Pareto_configurations(g.space);
      eliminate configs. in g.space not in pareto;
   }
   return g.space;
}
```

Our approach takes advantage of such independence of parameters to prune the configuration space.

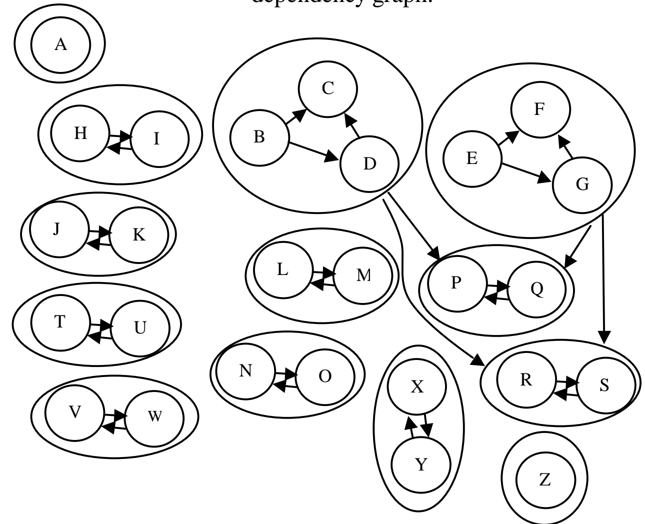
4.3 Parameter Dependency Model

Since our approach takes advantage of dependencies of parameters on each other, we need a model to formally capture the dependencies. We use a graph model. In our graph, nodes represent parameters and directed edges represent dependencies.

An edge from node (parameter) *A* to node (parameter) *B* indicates that the Pareto-optimal configurations of *B* should be calculated only after the Pareto-optimal configurations of *A* are computed. More generally, a path from *A* to *B* indicates that the Pareto-optimal configurations of *B* should be calculated once the Pareto-optimal configurations of all the nodes from *A* to *B,* residing on the path, are calculated, in that order. Meanwhile, all other parameters can be fixed to some arbitrary value. If there is an edge from *A* to *B* and an edge from *B* to *A*, then the Pareto-optimal configurations of parameters *A* and *B* must be calculated simultaneously. More generally, a path from *A* to *B* and back to *A*, which forms a cycle, indicates that the Pareto-optimal configurations of all the parameters on the cycle need to be calculated simultaneously. During that calculation, all other parameters not on the path can be temporally fixed to some arbitrary value. The Pareto-optimal configurations of an isolated node can be computed by temporally setting all other parameters to some arbitrary value.

The complete dependency graph of our target SOC of Figure 2 is given in Figure 4. We assume the designer of the SOC architecture determines the dependencies among the parameters. Often these dependencies follow from the structure of the SOC. For example, given an optimal configuration of the instruction cache, one can tune the data cache parameters without effecting the optimality of the instruction cache, since the optimally performing instruction cache will maximize instruction cache hit rate and no data cache configuration can have an effect on the instruction cache. In our graph, there are no edges going from B, C, or D to E, F, or G, stating that the

Figure 6: Initial clustering (part 1 of our algorithm) of the dependency graph.



instruction cache and data cache are independent. If the designer cannot establish the dependency of two or more parameters, than he or she should conservatively assume that they are dependent. In future work, we plan to automate this dependency determination.

4.4 Exploration Algorithm

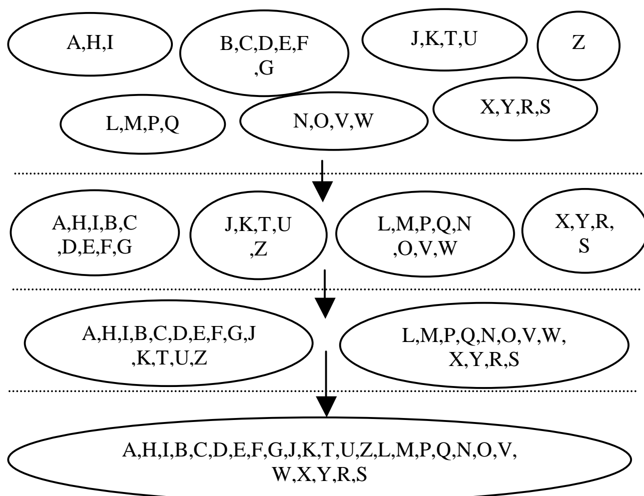
Given a dependency graph, our algorithm works as shown in Figure 5. The algorithm can be broken down into two parts. The first part performs a local search for Pareto-optimal configurations. The second part iteratively expands the local search to discover global Pareto-optimal configurations. More specifically, the first part performs a clustering of dependent nodes in our dependency graph and finds Pareto-optimal configurations within each cluster. The second part of the algorithm joins pairs of the clusters at a time and finds Pareto-optimal configurations within them until all the clusters have been combined.

The first part of our algorithm is to cluster together dependent nodes in the graph. This is the same problem as finding strongly connected components[1] of a graph and is shown in Figure 6, as performed on our sample dependency graph of Figure 4. Here, a depth first search of the graph can be used to accomplish this. In addition, if two clusters are connected (but not strongly), then they are topologically ordered. Here, each cluster represents a sub-space of the configurations of the SOC architecture. We use our exhaustive algorithm for calculating Pareto-optimal configurations, in topological order, for each of the clusters. Then, we restrict possible configurations of that cluster to the Pareto-optimal configurations only. This pruning is justified since if a configuration is not Pareto-optimal within a cluster, it cannot be part of a Pareto-optimal configuration for the entire configuration space. Conversely, if a configuration is Pareto-optimal within a cluster, it may or may not be Pareto-optimal given the entire configuration space, and thus must remain.

---

[1] A strongly connected component of a graph is a maximal set of nodes such that for any pair of nodes, u and v, in the set, we have a path from u to v and from v to u.

Figure 7: Cluster merging (part 2 of our algorithm).



Our exhaustive approach applied to clusters is usually feasible since these clusters represent only a small sub-space of the total configuration space. Nevertheless, heuristics such as probabilistic exploration techniques can be used to search within a cluster when the exhaustive method is too time-consuming.

The second part of our algorithm combines a pair of clusters into a single cluster and computes Pareto-optimal configurations within it. Then, it limits the space of this new cluster to the Pareto-optimal configurations only. This procedure is repeated until all the original clusters have been merged into pairs. Then we repeat the process until at the end a single cluster remains. The Pareto-optimal configurations within this last cluster represent Pareto-optimal configurations of the entire configuration space. This sequence of clustering, applied to our target SOC example, is depicted in Figure 7.

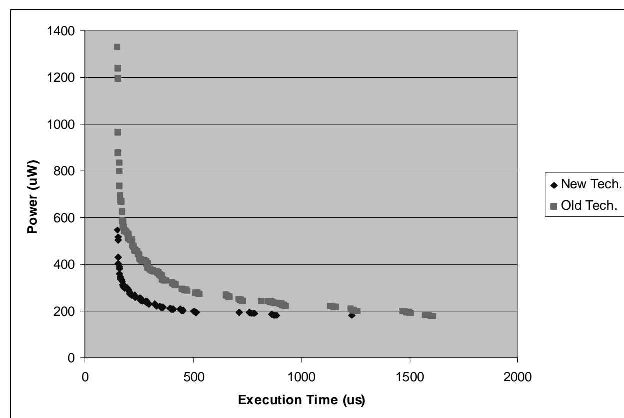
The time complexity of our algorithm is bounded by $O((K+log(K)) * 2^{N/K})$, where K denotes the number of clusters and N denotes the number of parameters. Here, the $2^{N/K}$ factor bounds the running time of the exhaustive computations of the Pareto-optimal points. The K in the first factor is a bound on the number of times that the first part of the algorithm loops, while the $log(K)$ is a bound on the number of times the second part of the algorithm loops. In the worst case, when $K=1$ (all parameters are dependent,) the running time is exponential, namely 2^N. In the best case, when $K=N$ (all parameters are independent,) the running time is linear, namely N. For most practical cases, the running time will be closer to the best case since the factor $2^{N/K}$ will decrease very rapidly as K increases.

We have outlined an exploration approach that uses a parameter dependency model to efficiently explore a large configuration space and returns the Pareto-optimal configurations. Note that the approach is exact and is not a heuristic.

5. Experiments

We have augmented our platform-tuning environment (Platune), described above, with the exploration algorithm outlined in this paper. Then, we explored the configuration space for 6 application programs, 4 different technologies, and 3 different clock-tree distribution networks, representing 72 different examples. For briefness, we give the results for one of the applications.

Figure 8: Pareto-optimal configurations of the JPEG example.



The application is named "JPEG" and implements a JPEG compression algorithm using the on-chip DCT CODEC core to perform the forward DCT transform. Quantization and Huffman encoding is performed via software running on the MIPS processor core of our architecture. A raw image is input and a compressed JPEG image is output using the on-chip UART core. Each simulation run processed a black and white picture, depicting earth from outer space, of size 32 by 32 pixels. We simulated both a version of the SOC that used power models for an older technology (0.25 micrometer) and a version that used power models for a newer technology (0.08 micrometer).

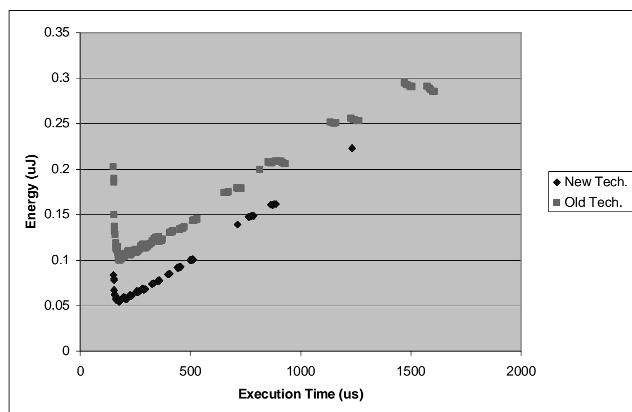
Our simulation and exploration speed results are as follows. Each simulation executed 8272 MIPS processor instructions. Our simulation model's throughput was measured to be 134,000 processor instructions per second, running on an 800 MHz Pentium III processor. The time to explore and find Pareto-optimal configurations for the newer technology was 5.7 minutes, during which our algorithm returned 77 Pareto-optimal configurations, and simulated 2474 distinct configurations. The time to explore and find Pareto-optimal configurations for the older technology was 25 minutes, during which our algorithm returned 141 Pareto-optimal configurations and simulated 9774 distinct configurations. The total number of configurations that where possible was over 10^{14}. Among the 72 different examples, we achieved an average pruning ratio of nearly 99.999997%.

The Pareto-optimal configurations of the JPEG example for the two technologies are presented in Figure 8. The performance varies by

Table 1: Summary of experimental results.

Technology(micron)	**Expl. Time (min)**	**Configs. Visited**	**Pareto-opt. configs.**
.08	5.7	2474	77
.25	25	9774	141
Technology (micron)	**Exe. Time Tradeoff**	**Power Tradeoff**	**Energy Tradeoff**
.08	8.11 times	3.02 times	4.12 times
.25	10.5 times	7.51 times	2.96 times

Figure 9: Energy tradeoffs of the JPEG example.



a factor of 8.11 for the new technology and 10.5 for the old technology. The power range varied by a factor of 3.02 for the new technology and 7.51 for the old technology. The energy range varied by a factor of 4.12 for the new technology and 2.96 for the old technology. The energy tradeoffs are given in Figure 9. In the new technology, the lowest energy is 0.0541 Joules.

Our exploration of the JPEG example revealed all the configurations of interest to a designer for two different technologies. The results for the JPEG example are summarized in Table 1. The Pareto-optimal configurations were obtained in a reasonable amount of time at the system-level.

6. Conclusions

We have presented an approach for efficiently finding all Pareto-optimal configurations of a parameterized SOC architecture. Our approach relies on our knowledge about the dependencies, in terms of execution time and power, among parameters of the SOC. We use a directed graph to capture these dependencies and give an algorithm to search the configuration space, incrementally, and prune inferior configurations. Our experiments with a JPEG example mapped onto our target SOC architecture demonstrate the feasibility of the approach. Future work includes automating the determination of dependencies, and introducing new parameters.

7. Acknowledgement

This work was supported by the National Science Foundation (CCR-9811164), (CCR-9876006), a Design Automation Conference Graduate Scholarship, and NEC USA.

8. References

[1] M.R. Stan, Wayne P. Burleson, Bus-Invert Coding for Low Power I/O, IEEE Transactions on VLSI, March 1995.

[2] A. Malik, B. Moyer, D. Cermak. A Programmable Unified Cache Architecture for Embedded Applications. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, November 2000.

[3] S.M. Kang. Accurate Simulation of Power Dissipation in VLSI Circuits. IEEE Journal of Solid-State Circuits, vol. CS21, no. 5, pp. 889-891, October 1986.

[4] G.Y Yacoub, W.H. Ku. An Accurate Simulation Technique for Short-Circuit Power Dissipation Based on Current Component Isolation. IEEE International Symposium on Circuits and Systems, pp. 1157-1161, 1989.

[5] R. Tjarnstorm. Power Dissipation Estimate by Switch Level Simulation. IEEE symposium on Circuits and Systems, pp. 881-884, 1989.

[6] T.H. Krodel. PowerPlay - Fast Dynamic Power Evaluation Based on Logic Simulation. IEEE International Conference on Computer Aided Design, pp. 96-100, Oct. 1991.

[7] A. Raghunathan, S. Dey, N.K. Jha. Register-transfer level evaluation techniques for switching activity and power consumption. International Conference on CAD Aided Design, pp. 158-165, 1996.

[8] E. Macii, M. Pedram. High-Level Power Modeling, Evaluation, and Optimization. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 17, no. 11, November 1998.

[9] D. Marculescu, R. Marculescu, M. Pedram. Information Theoretic Measures for Power Analysis. IEEE Transactions on Computer Aided Design, vol. 15, no. 6, pp. 599-610, 1996.

[10] M. Nemani, F. Najm. Toward a High Level Power Evaluation Capability. IEEE Transactions on Computer Aided Design, vol. 15, no. 6, pp. 588-598, 1996.

[11] V. Tiwari, S. Malik, A. Wolfe. Power Analysis of Embedded Software: A First Step Toward Sofware Power Minimization. IEEE Transactions on VLSI Systems, vol. 2, no. 4, pp. 437-445, 1994.

[12] C.T. Hsieh, M. Pedram, H. Mehta, F. Rastgar. Profile Driven Program Synthesis for Evaluation of System Power Dissipation. Design Automation Conference, June 1997.

[13] C. Barndolese, W. Fornaciari, F. Salice, D. Sciuto. Energy Evaluation for 32-bit Microprocessor. International Workshop on Hardware/Software Co-Design, 2000.

[14] R.J. Evans, P.D. Franzon. Energy Consumption Modeling and Optimization for SRAMs, IEEE Journal of Solid-State Circuits, Vol. 30, No. 5, pp. 571-579, 1995.

[15] T.D. Givargis, J. Henkel, F. Vahid. Interface and Cache Power Exploration for Core Based Embedded System Design. International Conference on Computer Aided Design, November 1999.

[16] T. Simunic, L. Benini, G. De Micheli. Cycle-accurate Evaluation of Energy Consumption in Embedded Systems. Design Automation Conference, pp. 876-872, 1999.

[17] Y. Li, J. Henkel. A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems. Design Automation Conference, pp. 188-193, 1998.

[18] P. Lieverse, P. van der Wolf, E. Deprettere, K. Vissers. A Mtehodology for Architecture Exploration of Heterogeneous Signal Processing Systems. IEEE Workshop on Signal Processing Systems, Taipei, October 1999.

[19] T.D. Givargis, F. Vahid, J. Henkel. Instruction based System-level Power Evaluation of System-on-a-chip Peripheral Cores. International Symposium on System Synthesis, September 2000.