Automated Design of Application Specific Superscalar Processors: An Analytical Approach

Tejas S. Karkhanis* and James E. Smith

{karkhani, jes}@ece.wisc.edu

Department of Electrical and Computer Engineering

University of Wisconsin - Madison

Abstract

Analytical modeling is applied to the automated design of application-specific superscalar processors. Using an analytical method bridges the gap between the size of the design space and the time required for detailed cycle-accurate simulations. The proposed design framework takes as inputs the design targets (upper bounds on execution time, area, and energy), design alternatives, and one or more application programs. The output is the set of out-of-order superscalar processors that are Pareto-optimal with respect to performance-energy-area. The core of the new design framework is made up of analytical performance and energy activity models, and an analytical model-based design optimization process.

For a set of benchmark programs and a design space of 2000 designs, the design framework arrives at all performance-energy-area Pareto-optimal design points within 16 minutes on a 2 GHz Pentium-4. In contrast, it is estimated that a naïve cycle-accurate simulation-based exhaustive search would require at least two months to arrive at the Pareto-optimal design points for the same design space.

Categories and Subject Descriptors
C.1.0 [**Processor Architectures**]: General

General Terms
Performance, Design.

Keywords
Application specific processors, analytical model, performance model, power model, design optimization.

1 Introduction

Widespread use of embedded microprocessors and the competitive marketplace have created the need for energy efficient microprocessors, occupying minimal silicon area, with stringent time-to-market requirements. Furthermore, the demand for increased functionality and higher performance are pushing out-of-order superscalar microarchitectures into the embedded microprocessor space; the PowerPC 440[1] is an example. The eventual widespread use of out-of-order superscalar processors in performance-intensive embedded products is inevitable. The challenge then is to design embedded out-of-order superscalar processors with short time-to-market, and that are performance-energy-area optimal for the target application program(s).

The design of an application-specific processor requires optimization at the circuit-level, the logic-level, and the microarchitecture-level. Current microarchitecture-level optimization methods are both human and computationally intensive. Consequently, current methods either evaluate a small number of designs from the design space, or they analyze a small part of the application program. This motivates an automated microarchitecture design framework based on computationally simple analytical models.

1.1 Design Framework Overview

The proposed automated design framework is composed of a number of parts. The *component database* stores pre-designed components, such as issue buffers, functional units, reorder buffers, register files, and caches, along with their silicon area and energy consumption. The *parameterized processor template* specifies the pre-designed components and interconnections between the components of a superscalar processor. *Performance* and *energy models* are based on analytical equations. The *area model* computes the total area of the superscalar processor configuration by first finding the area occupied by each pre-designed component and then summing the individual component areas.

The design optimization process employs a simple divide-and-conquer approach to arrive at optimal designs from the available components. The performance, energy, and area of each configuration are evaluated with the respective models, and a set of Pareto-optimal configurations is produced. From this set of Pareto-optimal configurations the ones that satisfy the design constraints are presented to the system designer. Or,

ISCA'07, June 9–13, 2007, San Diego, California, USA.

* Tejas Karkhanis was a student at University of Wisconsin-Madison when this research was performed. He now works at the Boston Design Center of Advanced Micro Devices.

if no designs satisfy the constraints, this information is also conveyed to the designer. For a set of benchmark programs, the design framework takes about 16 minutes to generate a set of designs that are performance-energy-area Pareto-optimal.

1.2 Related Work

A naïve method for automatically generating Pareto-optimal designs is to first generate the set of all designs from the available components, then employ cycle accurate simulations to arrive at performance and energy data for each configuration. Then, the set of Pareto-optimal designs can be selected – henceforth, this method is called the *baseline method*. The baseline method will find the Pareto-optimal designs, however it suffers from the obvious disadvantage of being very time-consuming and therefore impractical in many situations. As a result, researchers have explored methods that trade-off accuracy for reduction in design optimization time.

Heuristic-based optimization methods [2-4] choose a subset of the design space for evaluation by employing heuristics such as the Conjugate Gradient method [5] and Simulated Annealing [6]. Heuristic methods have two limitations: they are prone to getting stuck in local minima, and they do not provide insight into rationale that support the optimal choice.

Trace sampling [2,7,8] reduces the number of instructions that must be simulated by tracing and skipping instructions from the original application trace. The tracing and skipping process continues until the entire original application trace is executed. Then, the sampled trace is given as the input to a cycle accurate simulator.

Statistical simulation [9-11] also reduces the total number of instructions that the cycle accurate simulator must simulate. This method first collects program statistics through functional and trace-driven simulations. Next, a synthetic instruction trace is generated using the collected program statistics. Finally, the synthetic instruction trace is simulated with a cycle accurate processor simulator. Typically, the synthetic instruction trace is orders of magnitude smaller than the original program trace, thereby reducing the design optimization time.

Although *analytical performance models* exist [12-15,24], an automated design framework based on these models has not been explored. While analytical performance model is one aspect of an automated design framework, these models alone do not constitute an automated design framework for the combination of performance, power and area.

1.3 Paper Contribution

Cycle accurate simulation is at the center of the previous design frameworks in one way or another (even statistical simulation has a cycle-level microprocessor model). In contrast, the method proposed here is based on *analytical modeling*, and the overall design framework which incorporates the analytical model is henceforth referred to as the *analytical method*, for brevity. In the analytical method, the design optimization process arrives at Pareto-optimal design parameters while accounting for the variation in resource requirements. Results suggest that a design framework based on first-order, computationally simple analytical models is feasible, and it can successfully bridge the long-standing gap between large number of design options and the inability of cycle-accurate simulations to explore a large design space.

2 Superscalar Processor Template

The template for the parameterized superscalar processor is shown in Fig. 1. The processor core contains an instruction issue buffer, a load/store buffer, and a separate reorder buffer (ROB). Processor parameters include the L1 instruction and data cache sizes, unified L2 cache size, branch predictor size, physical registers, rename map entries, and numbers of functional units of each type. These parameterized components of the superscalar processor are pre-designed and stored in the component database along with the silicon area they will occupy and the energy modeling information.

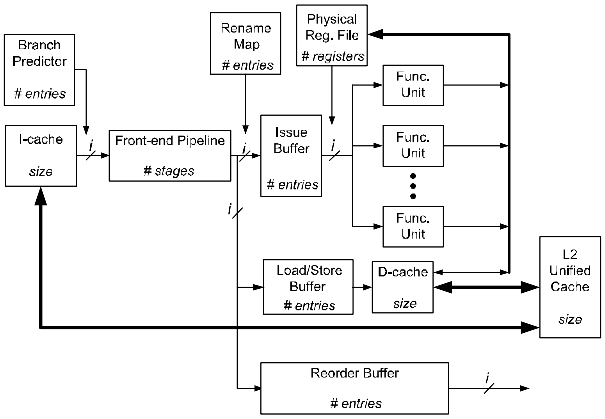


Fig. 1: Superscalar Processor Template

3 Program Statistics

The following set of application program statistics drive the automated design framework. These statistics are collected from functional or trace-driven simulations.

- **Critical-path lengths**: The average length of the longest data dependence chain for each *window* or sequence of *W* consecutive dynamic instructions. There is one average length for each window size *W*, and the maximum *W* is the maximum-sized reorder buffer in the component database. This statistic is used in determining the reorder buffer size.
- **Dependence length distribution**: The average length of the longest dependence chain leading to each instruction for each window size as defined above. This statistic used in determining the issue buffer size.
- **Functional unit mix**: the fraction of the executed instructions that use each of the functional unit types (e.g., an integer ALU or data cache port).
- **Cache miss-rate**: the number of instructions that miss in the cache divided by the number of instructions. The miss rates are determined for the set of caches in the design space; we used Tycho[16].
- **Branch miss-prediction rate**: the number of branches that are miss-predicted divided by number of executed instructions in the program. The miss-prediction rate is measured for the set of branch predictors in the design space with a trace-driven branch predictor simulator.

- **Average load independence**: for a load that misses in the L2 cache, the average number of other load misses within window *W* that are not transitively data dependent on the subject load miss. This statistic indicates the degree to which L2 misses can be overlapped.

Program statistics for the functional unit mix and cache miss rate for the L1 data cache are accumulated over intervals of one million instructions and then *mean* and *standard deviation* of the interval data is reported. As we shall see later (in Section 6), the design optimization process uses the mean and standard deviation to model the dynamic variation in resource requirements of a program. Table 1 has the time required by the trace-driven simulators to analyze a trace of 100 million instructions; when longer traces are used, these times will grow linearly.

Table 1. Time for analyzing 100M instructions.

Program Statistic	Time
Critical-path, Dependence length, Functional unit mix, Load independence	1.8 min
Cache miss rates	30 sec per config.
Branch miss-prediction rates	2 mins per config.

4 Performance Model

For the performance model, we use *interval analysis* [26], which is based on a combination of methods proposed by Michaud et al.[12], Taha and Wills[15], and Karkhanis and Smith[17].

The basis for the model is the observation that under *ideal* conditions, a superscalar processor can sustain an instruction per cycle (IPC) rate that is roughly equal to the superscalar pipeline width. However, under non-ideal conditions, the smooth flow of instructions is intermittently disrupted by miss-events such as cache misses and branch mispredictions. After a miss-event occurs, the issuing of useful instructions stops; there is then a period when no useful instructions are issued until the miss-event is resolved and instructions can once again begin flowing.

This behavior is illustrated in Fig. 2. The number of instructions per cycle that are committed is shown on the y axis, and time (in clock cycles) is on the x axis. As illustrated in the figure, the effects of miss events divide execution time into intervals. Intervals begin and end at the points where instructions just begin issuing following recovery from the preceding miss event. That is, an interval includes the time period following the particular miss event when no instructions are issued.

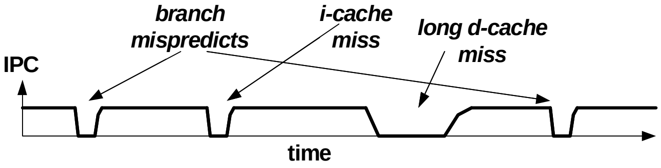


Fig. 2: Miss-events break program execution into intervals

By dividing execution time into intervals, performance behavior of an individual interval can be modeled by considering both the type of miss-event that terminates it and by the number of instructions in the interval. First, program characteristics and basic processor parameters (the reorder buffer size and instruction issue width) feed a model that tracks instruction issue rate as an interval evolves from beginning to end. Next, statistics generated through analysis (or estimation) of cache and predictor behavior yield distributions of interval lengths for each interval type. Then, given the types of the intervals and their lengths, the model provides an estimate of the total time that will be spent in each interval type. Overall processor performance is simply the aggregate performance over all the interval types [26].

The methods for computing interval timings can be derived using techniques similar to those used in an earlier model [17]. This paper employs the formula given in equation 1.

$$CPI = (1/i) + \{(1/N_{TOTAL}) \times \{[(i\text{-}1)/2] \times (m_{iL1}+m_{brm}+ m_{iL2}+m_{tbr}) + [m_{iL1}\times c_{L2}] + [m_{iL2}\times c_{mm}] + [m_{br}\times(c_{dr}+c_{fe})] + [M_{dL2}\times c_{mm}]\}\} \quad (1)$$

The first term is the ideal CPI; *i* is the peak issue rate. The remaining terms are divided by N_{TOTAL} , the total number of dynamic program instructions, to reduce them to CPI. The second term is the inefficiency in the fetch/decode unit. This assumes an aggressive front end where all the inefficiency in front-end instruction flow is due to interruptions caused by miss-events. m_k is the fraction of instructions that result in a miss-event of type *k*; m_{tbr} models the inefficiency due to correctly predicted taken branches. The third term reflects the additional cycles due to L1 instruction cache misses; c_{L2} is the L2 cache access delay in cycles. The fourth term reflects the additional cycles for instruction misses in the L2 cache; c_{mm} is the memory access time. The fifth term is the additional cycles due to branch mispredictions [25]. It is the miss rate times the ROB drain time, denoted as c_{dr}, plus the front-end pipeline length (fill time), denoted as c_{fe}. The parameter c_{dr} is cycle penalty for not issuing at the peak issue rate when the ROB is running out of useful instructions. It is a function of the critical-path characteristic (see Section 3) and the ROB size, and it is computed iteratively as explained in [12, 17]. The final term is the additional cycles due to L2 data cache misses; M_{dL2} is computed from the L2 data cache miss rate and the load independence statistic as described in the next paragraph.

Equation 1 is based on certain assumptions regarding overlapping effects of miss-events [17]. Instruction cache misses and branch mispredictions inherently do not overlap with other instruction cache misses and branch mispredictions. Long data cache misses however can overlap with each other. Therefore, M_{dL2} is computed as $(m_{dL2}/N_{ovr})/N_{TOTAL}$, where L2 data misses overlap in groups of N_{ovr}, and m_{dL2} denotes the total number of L2 data misses. N_{ovr} is a function of the L2 cache and the reorder buffer size and is obtained from the average load independence statistic (see Section 3). Given the L2 cache and the reorder buffer size, the average load independence statistic is assigned to N_{ovr}.

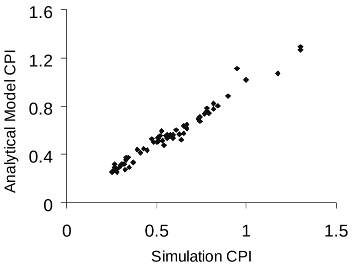
The accuracy of equation 1 is evaluated by comparing its CPI estimate with that generated from a cycle-accurate simulator. For this comparison, 32-bit PowerPC traces of SPEC-cpu2000 and MiBench workloads generated with the program

AMBER [18] are used. The baseline designs are PowerPC440-like configuration [1] and a Power4-like configuration [19].

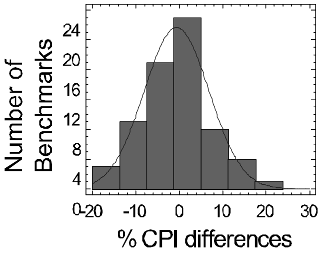
Equation 1 tracks cycle accurate simulation very well. Fig. 3a is a scatter plot for the analytical model CPI and detailed simulation CPI for a total of 36 benchmarks (24 SPEC-cpu2000 and 12 MiBench benchmarks). The correlation coefficient of analytical model and the cycle-accurate simulation CPI estimates is 0.95.

The CPI difference between the analytical model and cycle-accurate simulation is 5% averaged over absolute values of the CPI differences of all benchmarks. The benchmark *mcf* has the highest CPI difference of 13%. Fig. 3b has the distribution of the CPI differences. The distribution mean is -0.4 and the standard deviation is 6.2. The Shapiro-Wilkes goodness of fit value for the histogram is 0.97 out of a maximum possible of; when the histogram is a perfect Normal distribution the value is 1.

A Normal distribution with the same mean and standard deviation as the data is overlaid on the bars for illustration. A correlation coefficient close to one and normally distributed differences indicate a fundamentally sound mechanistic model [20].



(a)



(b)

Fig. 3. Comparison of Analytical Model and Simulated EPI. a) EPI correlation. b) distribution of EPI differences.

5 Energy Activity Model

As has been noted, the performance model just described is similar to earlier work (although not identical). The method for energy modeling is new in this work.

5.1 Quantifying Energy

Each clock cycle, a microarchitecture component is assigned one of three energy activities: *Active*, *Stalled*, and *Idle*. To be Active, the component is actively performing its intended operation during the given clock cycle. For example, an issue buffer slot is Active during a clock cycle when an instruction is actually issued from the slot. To be Stalled, the component is not performing its intended operation, but holds valid information (data and/or control) to be processed later. A component is Idle when it does not hold valid information (data and/or control). For example, an issue buffer entry is Idle if it does not hold a valid instruction. This method of categorizing energy activities is referred to as the *ASI method.*

The energy multipliers for ASI activities can be provided in a number of ways. One option is to generate energy multipliers via HSPICE simulations. For the results reported here we employed first-order estimates from the Wattch [21] library of power models.

5.2 Computing Energy Activities

The basic approach for computing energy activities relies on the same underlying model as for estimating performance. That is, the energy activities are related to the steady-state cycles and miss-event cycles. One complication, however, is that the performance model does not explicitly account for miss-speculated instructions (although it does account for their performance effects). To handle activities due to miss-speculated instructions, each of the three ASI activities are divided into two parts: *Used* and *Flushed*. The Used part is the energy activity for committed instructions. The Flushed part models the energy activity for instructions that are fetched following a mispredicted branch, but later discarded. For example, the activity for the issue buffer entries that hold miss-speculated instructions is *Issue Buffer Stalled Flushed (IB_SF).*

The overall analytical approach is: 1) compute the Used portion of each type of energy activity, 2) compute the Flushed portion of energy activity assuming the miss-speculated instructions have the same program statistics as the Used instructions, and 3) add the Used and Flushed activities of each type for each component.

5.3 Example: Issue Buffer ASI

As just defined, an issue buffer entry can be Active, Stalled, or Idle during a given clock cycle. Let B denote the number of issue buffer entries. Under ideal conditions, an average of i slots are Active every cycle to sustain the steady-state issue rate of i instructions per cycle. A processor configuration requires $CPI_{ideal} \times N_{TOTAL}$ cycles to execute the program. Because CPI_{ideal} is $1/i$, Active-Used issue buffer activity under ideal conditions is then computed with equation 2.

$$IB\_AU_{ideal} = N_{TOTAL} \qquad (2)$$

Of the B instructions in the issue buffer i issue every cycle, so on average (B-i) issue buffer entries are Stalled. Equation 3 computes the issue buffer Stalled-Used activity under ideal conditions. Idle issue buffer activity under ideal conditions is zero, because all issue buffer entries are occupied in a balanced design.

$$IB\_SU_{ideal} = \{[(B/i)\text{-}1]\times N_{TOTAL}\} \quad (3)$$

During an instruction cache miss, the issue buffer runs out of instructions and all *B* issue buffer entries are Idle until the missed instructions enter the issue buffer. Consequently, there are no Active-Used and Stalled-Used activities during this time. Idle-Used issue buffer activity due to instruction misses is modeled with equation 4.

$$IB\_IU_{imisses} = N_{TOTAL}\times[(m_{iL1}\times c_{L2})+(m_{iL2}\times c_{mm})]\times B \quad (4)$$

When one or more loads miss in the L2 cache, instruction commit eventually stops. Fetch and dispatch stop relatively quickly thereafter. The issue buffer entries have unissued instructions following the missed load. Commit resumes only after the missed load data returns. There are no Active-Used and Idle-Used activities during this time. Stalled-Used issue buffer activity because of long data cache misses is computed with equation 5.

$$IB\_SU_{L2dmisses} = N_{TOTAL}\times m_{dL2}\times B\times c_{mm} \quad (5)$$

Total Active-Used issue buffer activity is computed with equation 2. Total Stall-Used activity is computed by adding equations 4 and 5, written as equation 6.

$$IB\_SU_{total} = \{[(B/i)\text{-}1]\times N_{TOTAL}\}+[N\times m_{dL2}\times B\times c_{mm}] \quad (6)$$

The issue buffer experiences Idle-Used activity only during instruction cache misses, therefore the total Idle-Used activity is computed with equation 4.

On every branch misprediction the issue buffer experiences (c_{dr}+c_{fe}) mispeculated cycles. Out of (c_{dr}+c_{fe}) cycles (j_{dis}-1) are Idle cycles because after the pipeline flush all issue buffer entries are Idle, where j_{dis} is the depth of the dispatch stage.

For [(c_{dr}+c_{fe})-(j_{dis}-1)] cycles the issue buffer is processing mispeculated instructions. For brevity, let us define c_{brm} as (c_{dr}+c_{fe}). Equations 7, 8, and 9 compute the Active-Flushed, Stall-Flushed, and Idle-Flushed activities, respectively.

$$IB\_AF_{total} = (m_{brm}\times N_{TOTAL}\times B) \times [c_{brm}\text{-}(j_{dis}\text{-}1)]$$
$$\times [IB\_AU_{total}/(IB\_AU_{total}+IB\_SU_{total}+IB\_IU_{total})] \quad (7)$$

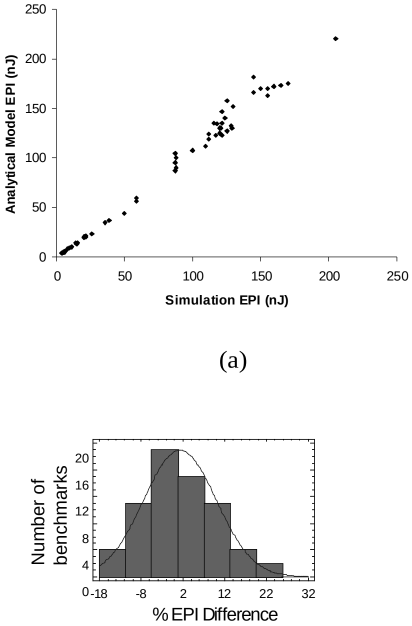
$$IB\_SF_{total} = (m_{brm}\times N_{TOTAL}\times B)\times[c_{brm}\text{-}(j_{dis}\text{-}1)]$$
$$\times [IB\_SU_{total} /(IB\_AU_{total}+IB\_SU_{total}+IB\_IU_{total})] \quad (8)$$

$$IB\_IF_{total} = \{(m_{brm}\times N_{TOTAL}\times B)\times[c_{brm}\text{-}(j_{dis}\text{-}1)]$$
$$\times [IB\_IU_{total}/(IB\_AU_{total}+IB\_SU_{total}+IB\_IU_{total})]\}$$
$$+ \{(j_{dis}\text{-}1)\times m_{brm}\times N_{TOTAL}\times B\} \quad (9)$$

5.5 Energy Activity Model Evaluation

To confirm that the analytical energy activity model is accurate, energy per instruction (EPI) computed with the analytical model is compared with the EPI generated with cycle-accurate simulation. To arrive at the EPI, energy activities are obtained from the respective models. Energy activity multipliers are taken from the models provided in power.h file of Wattch [21]. Note that the same energy activity multipliers are used for the analytical energy activity model and for the activities generated from cycle accurate simulation. The energy activities are multiplied by their corresponding multipliers and the total energy is calculated. Finally, the total energy is divided by the number of dynamic program instructions.

For all benchmarks, the analytical energy activity model tracks simulation (see Fig. 4a). The correlation coefficient is 0.99. The average difference between the analytical model and cycle accurate simulation is about *5.4*% over all benchmarks. The benchmark *mcf* has the highest difference of *15*%. The distribution of the CPI differences shown in Fig. 4b has a statistical mean of 0.8 and a standard deviation of 6.7. The Shapiro-Wilkes value for the normality test of the data is 0.93 out of a maximum of one, indicating that the histogram follows a Normal distribution.



(a)

(b)

Fig. 4. Comparison of Analytical Model and Simulation EPI. a) EPI correlation. b) distribution of EPI differences.

6 Design Optimization Process

The design optimization process employs a divide-and-conquer approach. The key insight is that the superscalar pipeline, cache, and branch predictor sub-systems can be optimized independently, in isolation of each other [2,17]. The divide-and-conquer process has several local optimization steps and a global optimization step.

The local optimization step arrives at the individually Pareto-optimal branch predictor, L1 instruction cache, L1 data cache, unified L2 cache, and superscalar pipeline by evaluating each subsystem separately. The Pareto-optimal caches and branch predictors are derived based on their miss-rates, silicon area, and energy consumption per access (component data provided in the component database and program statistics from Section 3). The Pareto-optimal superscalar pipelines are found with an analytical portion of the optimization process (Section 6.1) that models relationships among various parameters of an idealized superscalar pipeline based on the application statistics. This allows a direct method for superscalar pipeline design. Then, the performance and energy models are employed and the area information from the component database is used to find the Pareto-optimal superscalar pipelines.

The global optimization step first constructs a cross-product of the locally Pareto-optimal superscalar pipelines, branch predictors, and caches. Then, CPI and EPI values of all of these designs are evaluated with the analytical performance and energy activity models, respectively. Area is computed by summing up the areas of the individual components found in the component database. The Pareto-optimal superscalar processor designs are based on this CPI, EPI and Area data.

The design optimization process is explained in this section. First, the local optimization method for designing idealized superscalar pipeline is discussed in Section 6.1. Local optimization of branch predictor and cache is not explicitly discussed because it requires miss-rate data generated with straightforward trace-driven simulations. The global optimization algorithm is the focus of Section 6.2.

6.1 Superscalar Pipeline Design

The superscalar pipeline design process is a sequence of six steps illustrated in Fig. 5. For a given issue width, each step uses results obtained in a previous step to derive additional parameters of the superscalar pipeline. First, the sufficient number of functional units of each type is determined for a particular issue width. Next, the number of reorder buffer entries to sustain the issue rate is computed. Based on the reorder buffer size, the issue buffer size, number of physical registers, and load/store buffer size are derived using analytical model equations.

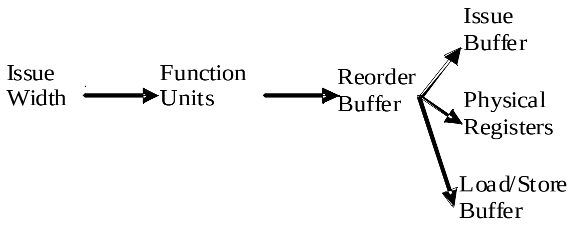


Fig. 5. Superscalar Pipeline Design Process.

Step 1: Compute Number of Functional Units

A sufficient number of functional units of type *h*, denoted as F_h, depends on three parameters: 1) the maximum issue rate *i*, 2) the fraction of instructions that require functional unit of type *h*, and 3) the issue latency of that functional unit (issue latency =1 if the functional unit is fully pipelined). Because instructions that require functional unit of type *h* may come in bursts, the mean functional unit demand denoted as $D_{h\mu}$ and the standard deviation of the demand data denoted as $D_{h\sigma}$ are used. $D_{h\mu}$ and $D_{h\sigma}$ are collected over intervals of length one million instructions during the trace-driven simulations (Section 3).

Equation 10 computes the sufficient number of functional units of type *h*. The term $[D_{h\mu}+(2\times D_{h\sigma})]$ accounts for the variation in functional unit requirements. Simulation results (not given here) show that two standard deviations added to the mean account for functional unit requirements in at least 97% of the length one million instruction intervals. A rationale for this technique is based on two well-known statistical facts: 1) the distribution of averages of “samples” taken from any underlying distribution tends to a Normal distribution by the virtue of Central Limit Theorem, and 2) adding two standard deviations to the mean of any Normal distribution accounts for 97.5% of the samples. In this case, a sample is a sequence of dynamic instructions that is one million instructions long. Finally, Little’s Law is applied and the term $[D_{h\mu}+(2\times D_{h\sigma})]$ is multiplied by the issue width *I* and the issue latency Y_h. The two standard deviation method is also used in subsequent steps where there is a need for modeling bursts in resource demands.

$$F_h = I \text{ x } [D_{h\mu}+(2\times D_{h\sigma})] \text{ x } Y_h \qquad (10)$$

Step 2: Compute Reorder Buffer Entries

The sufficient number of reorder buffer entries is computed with equation 11 [12]. In the equation, *W* is the number of reorder buffer entries, the function $K(W)$ is the average critical-path distribution measured with trace analysis (see Section 3), and c_{exe} is the average instruction execution latency computed with the functional unit mix.

$$i = W/[c_{exe} \text{ x } K(W)] \qquad (11)$$

Using equation 11 values of *i* are computed for a spectrum of *W* values. The smallest *W* that yields *i* within 3% of the peak issue rate is chosen as the reorder buffer size.

Step 3: Compute issue buffer entries

Equation 12 computes the number of issue buffer entries based on the reorder buffer entries. The function $A()$is the average instruction dependence length for a sequence of *W* instructions (see Section 3). The ratio $A(W)/K(W)$ determines the fraction of reorder buffer residence time spent waiting in the issue buffer for dependences to resolve.

$$B = W \text{ x } [A(W)/K(W)] \qquad (12)$$

Step 4: Compute Load/Store Buffer Entries

The load/store buffer entries must have sufficient entries for all in-flight load and store instructions. The number of in-flight instructions is *W* (reorder buffer size, computed in equation 11). Equation 13 computes the sufficient load/store buffer entries. In the equation, $D_{LDST\mu}$ is the mean load/store demand and $D_{LDST\sigma}$ is the standard deviation in the demand.

$$LS = W \text{ x } [D_{LDST\sigma} + (2 \text{ x } D_{LDST\mu})] \qquad (13)$$

Step 5: Compute Number of Physical Registers

The physical register file must be able to accommodate the requirements of all in-flight instructions that write to a register. The number of in-flight instructions is *W* (reorder buffer size computed in equation 11). Equation 14 computes the sufficient number of physical registers. The mean instruction demand for the physical register file is donated by $D_{WR\mu}$ and the standard deviation is denoted as $D_{WR\sigma}$.

$$PR = W \text{ x } [D_{WR\sigma} + (2 \text{ x } D_{WR\mu})] \quad (14)$$

6.2 Optimization Algorithm

The overall optimization algorithm operates in a sequence of eight steps. The first step collects program statistics. Steps 2 to 5 perform local optimization. Steps 6 to 8 are for the global optimization.

1. **Software Evaluation**: For the given application program(s), measure miss-rates for all instruction caches, data caches, unified caches, and branch predictors in the design space with application analyzers (i.e. simple trace-driven simulations).
2. **Idealized Pipeline Design**: Determine the idealized superscalar pipeline for all issue widths in the design space, as explained in Section 6.1.
3. **Cache Optimization**: Find miss-rate, energy per access, and area Pareto-optimal cache designs from the set of caches evaluated in Step 1, each independently of the others. Miss-rates are from Step 1. The area and energy per access information of every cache in the design space is obtained from the component database.
4. **Branch Predictor Optimization**: Find the miss-prediction rate, energy per access, and area Pareto-optimal branch predictors from the set of branch predictors evaluated in Step 1. The miss-prediction rate was measured in Step 1. The area and energy of every branch predictor in the design space is obtained from the component database.
5. **Superscalar Pipeline Optimization**: Find CPI, EPI, and Area Pareto-optimal idealized superscalar pipelines from the pipelines designed in Step 2. CPI of a superscalar pipeline is 1/*i* because the pipeline resources are sized to achieve the peak issue rate of *i*. EPI is computed for superscalar pipeline components with equations that model energy activity under ideal conditions (Section 5). The area of a superscalar pipeline is the sum of the areas of its individual components, as specified in the component database.
6. **Superscalar Processor Design**: Using combinations of Pareto-optimal caches (Step 3), branch predictors (Step 4), and idealized processor pipelines (Step 5) compose superscalar processor designs.
7. **Superscalar Processor Optimization**: Find CPI, EPI, and Area Pareto-optimal superscalar processors from the designs in Step 6. The performance model from Section 4 will compute the CPI. The energy activity model from Section 5 will compute the EPI. Area is computed by adding the cache, branch predictor, and idealized superscalar pipeline areas.
8. **Superscalar Processor Selection**: The processor designer chooses a superscalar processor design from the Pareto-optimal designs obtained in Step 7 that best meets his/her CPI, EPI, and Area target(s).

7 Design Framework Evaluation

Two key findings from the following evaluation are: 1) The set of Pareto-optimal designs produced with the analytical method is the same set of designs generated with the exhaustive baseline method, and 2) The Pareto-optimal CPI vs. Area, EPI vs. Area, CPI vs. EPI, and (CPI×EPI) vs. Area curves from analytical method track the same curves generated with the baseline method. This section explains the evaluation process and gives detailed results. Application software and infrastructure setup is discussed in Section 7.1. The results and the discussion are in Sections 7.2 and 7.3.

7.1 Design Space and Workloads

The superscalar processor design space used for the evaluation is in Table 2; there are about 2000 design configurations. Area of the pre-designed components is computed with the Mulder and Flynn method [22, 23], and then stored in the component database. For the evaluation, the number of front-end pipeline stages is fixed at five, although the analytical models will accommodate any pipeline length.

Table 2. Design space used for evaluation

Parameterized component	Range
L2 Unified Cache	64, 128, 256, 512 KB
L1 I and D Caches	1, 2, 4, 8, 16, 32 KB
Branch Predictor gShare	1, 2, 4, 8, 16K entries
Issue width	1 to 8
Functional units	Up to 8 of each type
Issue Buffers	8 to 80; incr. of 16
Load/Store Buffers	16 to 256; incr. of 32
Reorder Buffers	16 to 256; incr. of 32

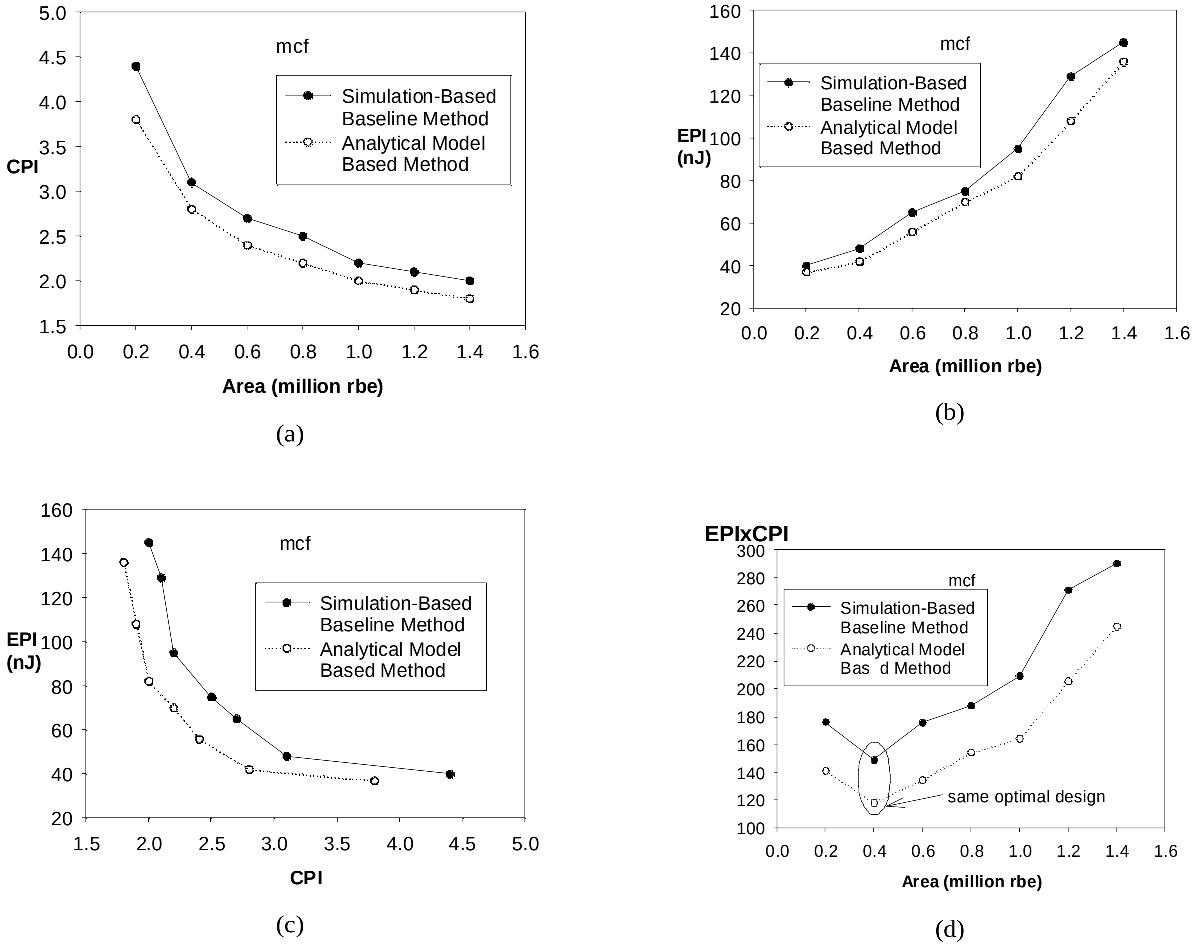


Fig. 6. Optimal CPI- EPI-area design points for *mcf*.

Programs from the MiBench and SPECcpu2000 are used as benchmark application-specific software. Each program is compiled for the 32-bit PowerPC instruction set and uses the test inputs. The programs are fast-forwarded 500 million instructions and then both methods are evaluated on the next 100 million instructions. The same instruction traces are the input to the baseline method and to the analytical method.

Traces of length 100 million instructions are used to make the baseline method, and therefore the comparison with analytical method, tractable. The embedded processor architect, in general, can use much longer trace lengths. It will become evident that analytical method will always be orders of magnitude faster than baseline method.

7.2 Comparison with Baseline Method

A set of optimal designs was generated for a spectrum of area targets with the analytical method and the baseline method. In every case, the analytical method arrived at the same Pareto-optimal designs as the baseline method. Then, CPI versus Area, EPI versus Area, CPI versus EPI, and (CPI x EPI) versus Area curves were plotted for the Pareto-optimal designs generated with both methods. Results of this comparison show that for all MiBench and SPEC-cpu2000 programs the analytical method tracks the baseline method very well. This result is not surprising. Sections 4 and 5 demonstrated CPI and energy activity analytical models track cycle-accurate simulation with the differences following the Normal distribution.

As a concrete example, consider *mcf*, essentially the worst-case benchmark of the ones we studied; it has the highest CPI and EPI differences when compared with detailed simulations. Figs. 6a-d has the CPI, EPI, and Area values of the Pareto-optimal designs generated with the baseline method (solid line) and with the analytical method (dashed line). The absolute difference between the CPI values of the two curves in Fig. 6a is around *13*% because of the approximations in the first-order performance model. However, the optimal designs identified by the analytical method track those identified by the baseline method quite well. Similarly, the absolute difference between the EPI values of the two curves in Fig. 6b is around *14*% because of the approximations in the first-order energy activity model. But, again the analytical method tracks the baseline method; the same is true for CPI vs. EPI curves in Fig. 6c. The important thing is that the (CPI×EPI) versus Area curves for both methods, in Fig. 6d, arrive at the same optimal design points regardless of the absolute CPI and EPI difference.

7.3 Performance of the Analytical Method

The combination of the performance model and the divide-and-conquer algorithm reduces the total time required to arrive at the Pareto-optimal processor designs. In this work, the analytical method arrives at all Pareto-optimal designs for a single program in about 16 minutes on a 2 GHz Pentium-4 machine. The baseline method, we estimate, will need 2 months to find the Pareto-optimal designs, for the same program, on the same machine. Given the application statistics, the analytical method optimizes processor parameters in little over a minute. The time breakdown according to different tasks within the analytical method is in Table 3. From the data it is evident that the analytical method consumes most of its time with trace-driven simulations.

The analytical method will always be more efficient than the baseline method. To see this, equations for the design times of the baseline method and the analytical method are developed.

Table 3. Time breakdown for optimization.

Task within design framework	Time (sec.)
Cache miss rates	512
Branch miss-prediction rate	250
Instr. Dep., Func. Unit Mix, Load stats	132
Design Optimization	70

Let G_i be the number of pre-designed components of type i in the design space, N_{TOTAL} the number of dynamic instructions in the application software as previously denoted, T_{SIM} the time per instruction for detailed cycle accurate simulation, T_T the time per instruction for a cache/branch predictor trace-driven simulation, and T_A the time spent for analytical modeling of one processor configuration (Sections 4, 5, and 6.1). The design time required with the baseline method denoted by T_{BM} is given by equation 15, and the design time with our analytical method denoted by T_{AM} is given by equation 16.

$$T_{BM} = N_{TOTAL} \text{ x } T_{SIM} \text{ x } \Pi_i G_i \qquad (15)$$

$$T_{AM} = [N_{TOTAL} \text{ x} T_T \text{ x } \Sigma_i G_i]+[T_A \text{x} \Pi_i G_i] \qquad (16)$$

Comparing equation 15 and equation 16, we can observe that equation 15 for T_{BM} has a product term consisting of all the G_i multiplied by the detailed simulation time for the entire application software. The same product term in equation 16 for T_{AM} is multiplied only by the time it takes to evaluate the analytical equations T_A, which is independent of the number of instructions in the application software and involves a small number of algebraic equations; T_A will be orders of magnitude smaller than T_{SIM}. The portion of T_{AM} that is a function of the benchmark length is the time required for collecting program statistics with trace-driven simulations – ($N_{TOTAL} \times T_T \times \Sigma_i G_i$).

Because the G_i terms are summed and not multiplied, their computation time is reduced considerably. Furthermore, in practice T_T will generally be much less than T_{SIM}. For instance, one trace-driven simulation T_T of a cache for trace length of 100 million instructions takes about *2* minutes on a 2 GHz Pentium-4 machine, whereas a detailed cycle accurate simulation (T_{SIM}) takes *40* minutes for the same trace length, on the same machine. Thus, analytical method will always be orders of magnitude faster than cycle-accurate simulation methods.

8 Summary/Conclusions/Future Work

Competitive marketplace demands that area and energy efficient embedded processors are designed as fast as possible, within the time-to-market that is required. Naïve methods that employ exhaustive search with cycle accurate simulations are not able to scale with respect to application software size and the design space. Analytical methods provide an attractive alternative because of their speed and the insights that analytical equations can provide. At the same time, analytical models abstract out the non-essential portions for designing superscalar processors. This paper explored the applicability of analytical methods for automatically designing out-of-order superscalar embedded processors.

The design optimization process employs a combination of local optimization and global optimization. The local optimization step reduces the number of sub-system combinations the global optimization step must evaluate. Both the local and global optimization steps employ analytical models for fast optimization. In the local optimization step an analytical portion quickly optimizes the superscalar pipeline. The set of analytical equations provide a method to balance various resources of the superscalar pipeline. Apart from their use in the design framework, these equations may be employed to gain insight into the relationship between various resources of a balanced superscalar processor.

This paper showed that analytical method for automatic design of application-specific superscalar processor is feasible and can provide the embedded processor designer with quick design feedback. This framework is attractive for gauging relative benefits of varying various superscalar processor parameters such as the issue width, functional units of various types, and level 1 and level 2 caches. The embedded processor architect can also run cycle accurate simulations on the Pareto-optimal designs found with the analytical method based framework. This will reduce the absolute difference in CPI and Energy activities (EPI) without requiring a simulation-based design space search.

The analytical method of automatic processor design and analysis opens up numerous possibilities and provides inspiration for future work. Our technique can be combined with previously proposed orthogonal automated design methods such as Simulated Annealing, and Trace Sampling. For example, Trace Sampling can be used to reduce the original program trace before the program is analyzed with trace-driven simulations. Simulated Annealing may be applied in the global optimization part of the design framework. The basic approach explored in this paper can be employed to design a single superscalar processor for a set of application programs. The same approach can be employed to design multi-core processors. In particular for multi-core processors, simulating each core in detail with cycle accurate simulations can get impractical. In future, we extend the analytical modeling method presented in this paper to intelligently and quickly optimize multi-core processors.

Acknowledgements

This work was supported by SRC contract 2000-HJ-782, NSF grants CCR-9900610, CCR-0311361, and EIA-0071924, IBM, and Intel Corporation. We also thank Lieven Eeckhout, Stijn Eyerman, and anonymous reviewers for numerous suggestions that improved this work.

References

[1] IBM, "PowerPC 440 Processor Core," available at http://www-306.ibm.com/.

[2] T. M. Conte, "Systematic Computer Architecture Prototyping," PhD Thesis: University of Illinois, 1992.

[3] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman, "PICO: Automatically designing custom computers," *IEEE Computer*, Sept. 2002, pp. 39-47.

[4] B. Kumar and E. S. Davidson, "Computer System Design Using a Hierarchical Approach to Performance Evaluation," *Communications of the ACM*, vol. 23, 1980, pp. 511-521.

[5] M. A. Bhatti, Practical Optimization Methods with Mathematica Applications: Springer Verlag, 2000.

[6] S. Kirkpatrick, C. Gellat, and M. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220-4598, 1983, pp. 671-680.

[7] E. Perelman, G. Hamerly, and B. Calder, "Picking Statistically Valid and Early Simulation Points," *International Conference on Parallel Architectures and Compilation Techniques*, 2003, pp. 244-255.

[8] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," *International Symposium on Computer Architecture*, 2003, pp. 84-97.

[9] L. Eeckhout, "Accurate Statistical Workload Modeling," PhD Thesis: University of Gent, 2002.

[10] S. Nussbaum and J. E. Smith, "Modeling Superscalar Processors via Statistical Simulation," *International Conference on Parallel Architectures and Compilation Techniques*, 2001, pp. 15-24.

[11] M. Oskin, F. T. Chong, and M. Farrens, "HLS: combining statistical and symbolic simulation to guide microprocessor designs," *International Symposium on Computer Architecture*, 2000, pp. 71-82.

[12] P. Michaud, A. Seznec, and S. Jourdan, "An Exploration of Instruction Fetch Requirement in Out-Of-Order Superscalar Processors," *International Journal of Parallel Processing*, vol. 29-1,2001, pp. 35-38.

[13] D. B. Noonburg and J. P. Shen, "Theoretical Modeling of Superscalar Processor Performance," *International Symposium on Microarchitecture*, 1994, pp. 52-62.

[14] E. Riseman and C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Trans. on Computer Architectures*, vol. C-21, 1972, pp. 1405-1411.

[15] T. Taha and D. S. Wills, "An Instruction Throughput Model of Superscalar Processors," *International Workshop on Rapid Systems Prototyping*, 2003, pp. 156-163.

[16] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, 1989, pp. 1612-1630.

[17] T. Karkhanis and J. E. Smith, "A First-Order Superscalar Processor Model," *International Symposium on Computer Architecture*, 2004, pp. 338-349.

[18] "Computer Hardware Understanding Development Tools 2.0 Reference Guide for MacOS X," July 2002.

[19] J. M. Tendler, et. al., "IBM Power 4: System Microarchitecture," *IBM Journal of Research and Development*, 2002, pp. 5-26.

[20] S. Kachigan, Statistical Analysis. New York: Radius Press, 1986.

[21] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," *International Symposium on Computer Architecture*, 2000, pp. 83-94.

[22] J. M. Mulder and M. Flynn, "An Area Model for On-Chip Memories and its Application," *IEEE Journal of Solid-State Circuits*, vol. 26, 1991, pp. 98-106.

[23] M. J. Flynn, Computer Architecture: Pipelined and Parallel Processor Design: Jones and Bartlett Publishers, 1995.

[24] E. Ipek, et al., “Efficiently Exploiting Architectural Design Spaces via Predictive Modeling,” *Architectural Support For Programming Languages and Operating Systems*, 2006, pp. 195-206.

[25] S. Eyerman, J. Smith, and L. Eeckhout, “Characterizing the Branch Misprediction Penalty”, *International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 48-58

[26] S. Eyerman, et al., “A Performance Counter Architecture for Computing Accurate CPI Components,” *Architectural Support For Programming Languages and Operating Systems*, 2006, pp. 175-174.