Criticality-driven Superscalar Design Space Exploration

Sandeep Navada, Niket K. Choudhary, Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{ssnavada, nkchoudh, ericro}@ece.ncsu.edu

ABSTRACT

It has become increasingly difficult to perform design space exploration (DSE) of computer systems with a short turnaround time because of exploding design spaces, increasing design complexity and long-running workloads. Researchers have used classical search/optimization techniques like simulated annealing, genetic algorithms, etc., to accelerate the DSE. While these techniques are better than an exhaustive search, a substantial amount of time must still be dedicated to DSE. This is a serious bottleneck in reducing research/development time. These techniques do not perform the DSE quickly enough, primarily because they do not leverage any insight as to how the different design parameters of a computer system interact to increase or degrade performance at a design point and treat the computer system as a "black-box".

We propose using criticality analysis to guide the classical search/optimization techniques. We perform criticality analysis to find the design parameter which is most detrimental to the performance at a given design point. Criticality analysis at a given design point provides a localized view of the region around the design point without performing simulations at the neighboring points. On the other hand, a classical search/optimization technique has a global view of the design space and avoids getting stuck at a local maximum. We use this synergistic behavior between the criticality analysis (good locally) and the classical search/optimization techniques (good globally) to accelerate the DSE.

For the DSE of superscalar processors on SPEC 2000 benchmarks, on average, criticality-driven walk achieves 3.8x speedup over random walk and criticality-driven simulated annealing achieves 2.3x speedup over simulated annealing.

Categories and Subject Descriptors

C.1.0 [**Processor Architectures**]: General

General Terms

Performance, Design

Keywords

design space exploration, criticality model, bottleneck analysis, superscalar processors, simulated annealing

PACT'10, September 11–15, 2010, Vienna, Austria.

1. INTRODUCTION

Design space exploration is extensively used by computer architects to identify the subspace where the computer system performs best, while satisfying other constraints such as power, cost, area, design complexity, etc. However, it is becoming increasingly difficult to perform design space exploration with a short turnaround time because of the exploding design space, increasing design complexity and long-running workloads [16][25]. In quantitative terms, for a modest design space consisting of 2000 design points, exhaustive search through cycle-accurate simulation takes up to two months to identify the best design point for the SPEC 2000 and MiBench benchmark suites [21].

To accelerate the design space exploration, researchers have adopted two orthogonal approaches. The first approach concentrates on reducing the time taken to measure the performance at a single design point. Techniques in this class include analytical techniques, regression modeling techniques, sampling techniques, reduced input set techniques and statistical simulation techniques [20][16][17][25][37][9][31][27]. These techniques trade accuracy for speed [20]. However, loss of accuracy might lead to erroneous conclusions from the design space exploration.

The second approach recognizes that an exhaustive design space search using a cycle-accurate simulation is an insurmountable task and hence aims to reduce the number of design points that needs to be searched. Classical search/optimization techniques such as random walk, simulated annealing, genetic algorithms, evolution strategy, etc. [8][10][14][18][24][28] fall in this category. These techniques use the performance values at previously simulated design points to guide the search. Further, these techniques have a global view of the design space and avoid getting stuck in a local maximum. Even though these search techniques are much better than exhaustive design space exploration, it still takes a considerable amount of time to perform design space exploration [10][14][18]. The main reason for this is that these techniques do not leverage any insight as to how the different design parameters of a computer system interact to increase or degrade performance at a given design point and treat the computer system as a "black-box" [21].

To gain insight about the computer system, we propose using criticality analysis (performance bottleneck analysis) to guide the classical search/optimization techniques. We perform criticality analysis to find the critical design parameter(s) (the design parameter(s) which is most detrimental to the performance) along with the simulation of the design point. Subsequently, we use this knowledge within the classical search/optimization techniques to find the next design point(s) to be explored. Criticality analysis helps in finding the critical design parameter(s) at the current design point without performing simulations at the neighboring

points. That is, by just performing the criticality analysis at the current design point, we obtain a localized view of the region around the design point. Hence, criticality analysis helps classical search/optimization techniques by avoiding unnecessary simulations of design points to understand the local contour of the region.

If standalone criticality analysis (without classical search/optimization technique) is used for the design space exploration, then the performance bottlenecks will be successively eliminated at every iteration until a design point is reached which has no bottlenecks. This balanced design point is the best design point in the localized region (local maximum). However, this point may not be a good design point in the global design space. Hence, to give criticality analysis a global view, it is used in conjunction with the classical search/optimization technique which has a global view of the design space and avoids getting stuck in a local maximum. Hence, we see that there is a synergistic behavior between the criticality analysis (good locally) and the classical search/optimization techniques (good globally). Criticality-driven design space exploration exploits this synergistic behavior to accelerate the design space exploration. **To summarize, criticality-driven design space exploration uses the localized information from the criticality analysis to guide the globally-aware classical search/optimization techniques.**

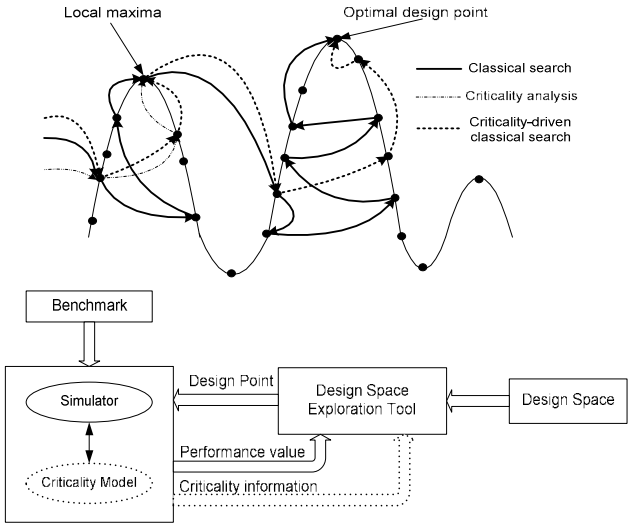

Figure 1. (a) (top) The synergistic behavior of criticality analysis and classical search/optimization techniques. (b) (bottom) Block diagram of Criticality-driven Design Space Exploration.

Figure 1(a) shows the performance landscape of a design space where the height represents the performance of the design point. The optimal design point is the highest point in the landscape. We see that standalone criticality analysis gets stuck in the local maximum. On the other hand, the globally-aware classical search technique reaches the global maximum in a large number of iterations. However, using criticality information to guide the classical search technique helps to reach the global maximum quickly.

Figure 1(b) illustrates the framework of the criticality-driven design space exploration which uses the criticality information in addition to the performance value of past design point(s) to find the next design point(s) to be explored.

While criticality-driven design space exploration is sufficiently general as to be adaptable for the design space exploration of any computer system, in this paper, we demonstrate an entire framework for fast, automated design space exploration of an out-of-order superscalar processor. To enable this, we need a high-fidelity superscalar design space and a detailed criticality model which models all the components of the superscalar processor.

Our high-fidelity superscalar design space encompasses structure sizes (reorder buffer, issue queue, load queue, store queue, instruction cache, data cache, etc.), widths of pipeline stages, and clock period. To take clock period into account, we need to measure the propagation delays of different pipeline stages in the superscalar processor. This requires physical implementation of a superscalar processor in a given technology. Therefore, we use the delay data of canonical pipeline stages of the superscalar processor from the FabScalar toolset [6] to build the design space. The high-fidelity design space unveils the delicate interplay between the instruction-level parallelism (ILP) extracting structures and the clock period [3][13] .

In addition to a high-fidelity superscalar design space, we have built a detailed criticality analysis model which is able to model all the components of an out-of-order superscalar processor. It should be noted that in the issue queue, instructions enter in-order but leave in an out-of-order fashion as data dependences get resolved and this makes modeling the issue queue more challenging than other resources such as the reorder buffer, load queue and store queue. Because of this complex out-of-order nature of issue queue, it was not modeled in previous criticality analysis models [11][34] even though it is one of the most critical components of a superscalar processor [11][21]. We have successfully incorporated it into our model. Further, we have extended Field's 3-node dispatch, execute and commit criticality model [11] to a 7-node criticality model in which there is a node corresponding to each of the canonical pipeline stages of a superscalar pipeline, to make our analysis more fine-grained.

For our evaluation, we use simulated annealing [24] and a less sophisticated technique, random walk, as the baseline search/optimization techniques. Note that simulated annealing is one of the state-of-the-art classical search/optimization techniques and has a proof for theoretical convergence [15]. For the design-space exploration of superscalar processors on SPEC 2000 benchmarks [33], on average (harmonic mean), criticality-driven walk achieves 3.8x speedup over random walk and criticality-driven simulated annealing achieves 2.3x speedup over simulated annealing. Another key finding is that even though simulated annealing performs better than random walk, criticality-driven walk performs better or comparable to simulated annealing on all SPEC 2000 benchmarks. This shows that using criticality analysis over a simple classical search/optimization technique is more effective than only using a sophisticated classical search/optimization technique.

This paper makes three main contributions.

- To the best of our knowledge, this is the first work to propose the use of criticality analysis to drive classical search/optimization techniques to accelerate the design space exploration of computer systems using the synergistic behavior between them.

- We present a detailed implementation of using criticality information to drive simulated annealing and random walk for an out-of-order superscalar processor. Further, we have performed detailed evaluation of criticality-driven simulated annealing and criticality-driven walk.
- We have built a detailed criticality analysis model of a whole out-of-order superscalar processor, including the issue queue. This detailed criticality model can also be used for other purposes like fine-grained bottleneck analysis.

The paper is organized as follows. In Section 2, we provide background on superscalar processors including the factors that affect their performance. Section 3 describes how criticality information is used to drive the design space exploration. Section 4 describes our high-fidelity superscalar design space. Criticality analysis using our detailed criticality model of a superscalar processor is explained in Section 5. Sections 6 and 7 describe our experimental methodology and results. Related work is presented in Section 8. Section 9 concludes our work.

2. BACKGROUND ON SUPERSCALAR PROCESSORS

2.1 Dimensions of a Superscalar Processor

A superscalar processor exposes and exploits instruction-level parallelism (ILP) in programs. The pipeline of a superscalar processor can be characterized along three dimensions:

- The sizes of ILP-extracting units (issue queue, load and store queues, physical register file / reorder buffer), caches, and predictors.
- The widths of pipeline stages.
- Clock frequency.

Another dimension is pipeline depth. Each high-level pipeline stage might be subdivided into multiple sub-stages. We refer to the number of sub-stages as the *depth of the pipeline stage*. In this paper, the depth of a pipeline stage is not treated as an independent parameter; rather, it is determined by frequency and the total propagation delay (logic and wire delays) of the pipeline stage.

2.2 Understanding Performance of a Superscalar Processor

When designing a superscalar processor, there is a tradeoff between accelerating the execution of *independent instructions* (*i.e.*, exposing and exploiting more ILP) and reducing the latency of *dependent instructions*, and this tradeoff plays out differently for different programs and program phases. Exploiting more ILP requires increasing the sizes of ILP-extracting units and the widths of pipeline stages, increasing their propagation delays. Deeper pipelining can help maintain the clock frequency despite the longer delays. However, the longer delays increase the latency of dependent instructions. Overall performance increases or decreases depending on whether parallelism or latency is the dominating factor. Different programs or program phases have different arrangements of independent and dependent instructions. This causes them to be characterized by different optimal pipeline designs.

2.3 Pipeline Loops

As mentioned previously, longer propagation delays increase the latency of dependent instructions irrespective of increasing pipeline depth. The reason is because dependent instructions exercise pipeline loops [3]. Three well-known pipeline loops are:

1. Control-dependence loop: This loop is exposed when a branch is mispredicted. The execute stage redirects the fetch unit in the case of a mispredicted branch. The delay of this loop is the total propagation delay between the fetch and execute stages, plus the latency of misprediction recovery.

2. Register-dependence loop: This loop is exposed for data-dependent instructions (but mainly single-cycle producer instructions). The delay of this loop is the propagation delay of the wakeup-select logic in the issue unit, and determines the minimum time to execute chains of data-dependent instructions.

3. Memory-dependence loop: This dependence is exposed when there is a store instruction followed by a load instruction to the same memory location within the pipeline. The delay of this loop is equal to the propagation delay of store-to-load forwarding.

3. CRITICALITY-DRIVEN DESIGN SPACE EXPLORATION

Simulated annealing is one of the state-of-the-art classical search/optimization techniques and has a proof for theoretical convergence [24]. Random walk can be considered as a simpler version of simulated annealing. Because both random walk and simulated annealing do not get stuck in local maxima, they are widely used in global optimization. In Section 3.1, we explain random walk and simulated annealing. In Section 3.2, we discuss how to use criticality information to drive random walk and simulated annealing. Section 3.3 describes criticality-driven design space exploration using other search/optimization techniques.

3.1 Classical Search Techniques

3.1.1 Random Walk

Random walk is one of the simplest but widely used search/optimization techniques used to perform global optimization. In this section, we describe how random walk can be used to perform design space exploration of superscalar processors [24].

A random design point is used as the starting point. Simulation is performed at this design point to obtain the performance value. A parameter is selected at random from the current design point and it is changed by the least significant unit to get a new design point. We call this a *random perturbation*. Simulation is performed again at the new design point to get its performance value. Random perturbation is then performed again at the new design point and this process is continued. During the entire process, the algorithm keeps track of the design point which generated the best performance value. If the performance value at the current design point is below 50% of the best performance value, then we restart from the best design point. Because of the restarting procedure, iterations are not wasted in searching through likely bad design points.

As random walk performs random perturbations and accepts new design points which are worse than the current design point, it does not get stuck in local maxima.

3.1.2 Simulated Annealing

Simulated annealing shares the spirit of random walk except that there is a probability associated with the acceptance of a new design point. If the new design point is not accepted, then random perturbation is performed again at the current design point to get another new design point.

The probability function is given as below:

$$P = \begin{cases} 1 & \text{if } perf_{new} \geq perf_{current} \\ e^{\left(\frac{perf_{new} - perf_{current}}{T}\right)} & \text{if } perf_{new} < perf_{current} \end{cases}$$

Where $perf_{new}$ is the performance at the new design point, $perf_{current}$ is the performance at the current design point, and T is the temperature function which is high at the start of the design space search and decreases every 20 iterations according to following equation:

$$T = T_0 \cdot (0.9)^{n/20}$$

Where T_o corresponds to the initial temperature (constant) and n is the number of design points traversed.

Note that the probability of accepting the new design point is always 1 if the new design point is better than the current design point. However, if the new design point is worse than the current design point, then the probability of accepting the design point is close to 1 at the beginning of the design space search when the temperature is high. However, as the design space search progresses, the probability function tends to zero. In other words, design space search starts as a random walk. However, when it reaches close to the optimal region, it only accepts better design points (gradient ascent). This prevents wasting unnecessary iterations when the optimal region is close. However, the initial temperature has to be carefully set. Otherwise, gradient ascent might start too early or too late. This might result in simulated annealing performing worse (taking more iterations) than random walk.

3.2 Criticality-driven Search Techniques

Although random walk and simulated annealing are much better than exhaustive design space search, using knowledge from performance bottleneck analysis can make it even faster. In criticality-driven design space search, we use criticality-driven perturbation instead of random perturbation to give random walk and simulated annealing a localized view of the region around the design point. This avoids unnecessary simulations in the localized region and, hence, accelerates the design space exploration. Section 3.2.1 explains criticality-driven perturbation. Sections 3.2.2 and 3.2.3 discusses how criticality-driven perturbation is used in criticality-driven walk and criticality-driven simulated annealing, respectively.

3.2.1 Criticality-driven Perturbation

To perform criticality-driven perturbation at the current design point, criticality analysis is performed along with the simulation of the design point. Criticality analysis assigns cycles from the total execution cycles to each of the processor resources (structure sizes and widths of pipeline stages) and program dependences (control, register and memory dependences). The processor resource/program dependence which has the most cycles attributed to it is the most critical bottleneck. Criticality analysis gives the most critical bottleneck at the given design point. Using this bottleneck information, the design parameter to be perturbed to improve the performance is found in the following way.

If the most critical bottleneck is a processor resource, the implication is that there is more instruction-level parallelism (ILP) to be extracted but doing so is prevented by the limited resource. Therefore, the next point in the design space will be based on increasing the size of that resource (structure size or width of a pipeline stage). Also, the affected pipeline stage will be increased in depth (sub-pipelined deeper) to accommodate the larger structure or width while maintaining the current clock frequency. However, if the most critical bottleneck is a program dependence, it implies that the processor resources are not critical. This means that the resources of the processor are oversized at the current design point. In this case, the clock frequency is increased while keeping the pipeline depth of each pipeline stage the same, which would decrease all the processor resources. This increases the performance as the clock frequency is increased.

Using the most critical bottleneck always leads to increasing a parameter. However, we also need to decrease the design parameters to make sure that the entire design space is reachable. In fact, to make the entire design space reachable, we need to increase and decrease the design parameters with the same probability. To decrease the design parameters, criticality analysis is also used to find the least critical bottleneck (processor resource or program dependence which has the least number of cycles attributed to it). Using the least critical bottleneck, the design parameter is decreased to improve performance in the following way.

If the least critical bottleneck is a processor resource, it implies that the processor resource is oversized. Hence, the next design point is found by decreasing the oversized resource. The affected pipeline stage will be decreased in depth to accommodate the resource while maintaining the current clock frequency. This helps to improve performance as the pipeline loop latency like branch misprediction loop latency, etc. [3], will decrease due to the decrease in pipeline depth corresponding to the processor resource. On the other hand, if the least critical bottleneck is a program dependence, it means that processor resources are critical. This implies that all the processor resources are limited and more parallelism can be extracted by increasing the processor resources. Hence, the clock frequency is decreased while keeping the pipeline depth of each pipeline stage the same, which increases all the resources.

The following table gives a summary of the above discussion on the design parameter which needs to be perturbed based on the bottleneck information from the criticality analysis.

Table 1. Criticality-driven perturbation

Bottleneck	Whether most/least critical	Design parameter to be perturbed	Comments
Processor resource	Most critical	Processor resource increased	Depth of stage increased
Program dependence	Most critical	Clock frequency increased	Decreases all resources
Processor resource	Least critical	Processor resource decreased	Depth of stage decreased
Program dependence	Least critical	Clock frequency decreased	Increases all resources

To understand if the criticality-driven perturbation yields performance improvement, we randomly picked 1,000 design points and for each performed a single criticality-driven perturbation. On average, we see that around 82% of criticality-driven perturbations resulted in a performance improvement. The remaining 18% of perturbations result in performance degradation.

One of the major reasons for performance degradation is due to the balanced design points (local maxima). A balanced design point is a design point at which all the design parameters contribute roughly equal numbers of cycles to the total execution cycles. Hence, at a balanced design point, there is no bottleneck and any perturbation at this point may result in performance degradation.

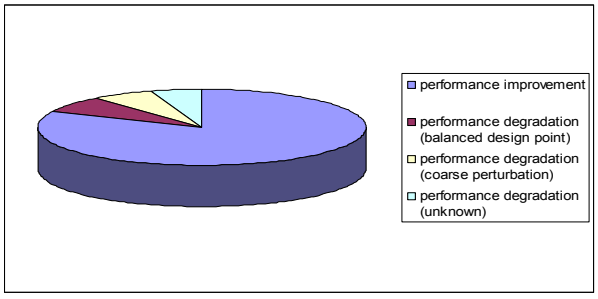

Figure 2. Breakdown of criticality-driven perturbation.

Other reasons for performance degradation could be the presence of parallel and near-critical paths. As we only model one critical path, the performance improvement is not guaranteed at these points [35]. Note that the fidelity of the criticality analysis would not affect the correctness of the design space exploration. It might just slightly slow down the design space exploration. However, our criticality model has enough fidelity to significantly accelerate the design space exploration.

To summarize, as 82% of criticality-driven perturbation results in a performance improvement, it shows that the criticality information is very useful in providing localized information to the design space exploration.

3.2.2 Criticality-driven Walk

In criticality-driven walk, to get a localized view, we perform criticality-driven perturbations instead of random perturbations. However, if a design point is revisited, then a random perturbation is performed instead, to avoid going in a loop.

At every iteration, criticality-driven walk tries to improve performance by selecting the parameter which would most likely yield a performance improvement. However, when it reaches a balanced design point (local maximum) where all parameters are almost equally critical, the parameter that does get selected might not improve performance. Once it drifts away from this balanced design point, it again tries to improve performance and ultimately reaches some other balanced design point (local maximum). After a certain number of iterations, it reaches a balanced design point which is the global maximum. Note that accepting design points which are worse than the current design point when it reaches local maxima helps criticality-driven walk avoid getting stuck at local maxima.

3.2.3 Criticality-driven Simulated Annealing

Similar to criticality-driven walk, to get a localized view, we use criticality-driven perturbation instead of random perturbation to generate a new design point. In case the new design point is not accepted, we resort to random perturbation. Also, if a design point is revisited, then a random perturbation is performed to avoid going in a loop.

During the beginning of the design space search, criticality-driven simulated annealing acts like a criticality-driven walk (high "temperature"). Hence, it reaches the optimal region in fewer iterations. Once it is close to the optimal region, it behaves like a gradient ascent method. Since criticality-driven perturbation has a localized view, the design point generated is more likely to be accepted by the probability function compared to random perturbation. Hence it takes fewer iterations even when it is doing a gradient ascent and overall, criticality-driven simulated annealing performs better than simulated annealing.

3.3 Extension to Other Search/Optimization Algorithms

As other search/optimization algorithms use random perturbations, we can easily use criticality-driven perturbation in other search/optimization algorithms [28][18]. For example, in genetic algorithms, there are two operators to generate new design points: mutation and crossover [28][18]. Mutation is a random perturbation. Hence, we can use criticality analysis to drive genetic algorithms by changing the mutation operator to a criticality-driven perturbation operator. Also, criticality analysis can be used to find good crossover sites in the design points while performing a crossover operation.

4. SUPERSCALAR DESIGN SPACE CREATION

This section explains the process of creating a high-fidelity superscalar design space. Later, it discusses a mechanism to prune the superscalar design space.

The performance of a superscalar processor is a function of both instructions per cycle (IPC) and clock frequency. To take clock frequency into account, we need to measure propagation delay. Hence, we use detailed synthesizable Verilog models and physical designs of superscalar processors from the FabScalar toolset [6]. FabScalar defines a canonical superscalar pipeline shown in Figure 3. The Standard Superscalar Library (SSL) of FabScalar

provides many different designs of each canonical superscalar pipeline stage that differ in their complexity (structure sizes and pipeline width) and depth.

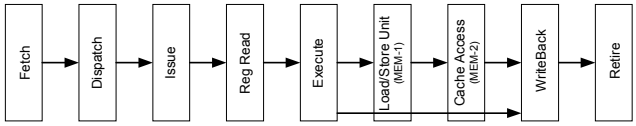

Figure 3. Canonical superscalar pipeline stages.

All possible combinations of pipeline stages that fit in a given clock period gives the design points for the given clock period. Doing this for different clock periods forms the entire design space of superscalar processors.

The following design space pruning strategy is used to decrease the size of the design space to a manageable size. Design space pruning is performed by weeding out among the design points having pipeline stages of same depth and same clock period. Among these design points having the same depth and clock period, the design point having pipeline stages with the highest complexity is retained and other design points having lower complexity are weeded out. This is because design points with lower complexity will always perform worse than the design point having highest complexity. For example, let us assume issue width of 2 and 4 for 16 entries issue queue fits into issue depth of 2 for clock period of 0.4ns. We discard the design point having issue width of 2 as it would always perform worse than the design point with issue width of 4. Sometimes it might be hard to distinguish which of them is most complex. In such cases, we don't perform pruning. For example, it is difficult to distinguish between complexity of issue queue size having 16 entries with width 2 and issue queue size having 8 entries with width 4. In case of design space exploration having fixed power/area budget, the above design space pruning is done only on those points which fit in the budget.

5. CRITICALITY ANALYSIS

The design parameter to be perturbed in the criticality-driven design space exploration is identified using the bottleneck information of the criticality analysis. This section explains the process of finding the bottlenecks for a given superscalar processor configuration using criticality analysis. Section 5.1 gives an overview of criticality analysis for a superscalar processor. Even though the issue queue lies at the heart of the superscalar processor, the issue queue was not modeled in past work on criticality analysis. Section 5.2 describes modeling of the issue queue in the criticality model. Section 5.3 illustrates the working of criticality analysis using an example. Section 5.4 gives the overhead of using criticality analysis.

5.1 Overview of Criticality Analysis

There are three main steps in finding the bottlenecks for a superscalar processor running a given application using criticality analysis.

1) Building of critical-path graph model

The critical-path graph model [11][29] of a processor is a graph-based model which helps to uncover the bottlenecks in the processor. It is a directed graph that models the resource constraints of the processor (structure sizes and pipeline widths) and control, register, and memory dependence constraints of the program currently being executed. It expresses the dependence relationships between different events occurring in the processor, which are dictated by different constraints of the processor and the program being executed. The graph is built using the trace of committed instructions of the program being executed.

Table 2. Overview of the constraints modeled by the criticality model of a superscalar processor

Constraint Modeled	Edge	Comments
Pipeline dependence	$F_i \rightarrow D_i$, $D_i \rightarrow I_i$, $I_i \rightarrow R_i$, $R_i \rightarrow E_i$, $E_i \rightarrow W_i$, $W_i \rightarrow C_i$	Applicable to non-mem. instructions
	$F_i \rightarrow D_i$, $D_i \rightarrow I_i$, $I_i \rightarrow R_i$, $R_i \rightarrow E_i$, $E_i \rightarrow M1_i$, $M1_i \rightarrow M2_i$, $M2_i \rightarrow W_i$, $W_i \rightarrow C_i$	Applicable to mem. instructions
In-order pipeline stage constraints	$F_{i-1} \rightarrow F_i$, $D_{i-1} \rightarrow D_i$,, $C_{i-1} \rightarrow C_i$	In-order constraint
	$D_{i-w} \rightarrow F_i$, $I_{i-w} \rightarrow D_i$	Width constraint, 0<w<=width
Reorder buffer	$C_{i-R} \rightarrow I_i$	R=ROB size
Load/Store Queue	$C_{K1} \rightarrow I_i$, $C_{K2} \rightarrow I_i$	K1 → bottleneck load instruction K2 → bottleneck store instruction
I-cache miss	$F_i \rightarrow D_i$, $F_i \rightarrow F_{i+1}$	Weight of the edge is i-cache miss latency
D-cache miss	$M2_i \rightarrow W_i$	Weight of the edge is d-cache miss latency
L2-cache miss	$F_i \rightarrow D_i$, $F_i \rightarrow F_{i+1}$, $M2_i \rightarrow W_i$	Weight of the edge is L2-cache miss latency
Issue width [36]	$R_b \rightarrow R_i$	b → instruction which consumed the last issue slot
Control dependence	$E_{i-1} \rightarrow F_i$	i-1 → mispredicted branch
Data dependence	$R_p \rightarrow R_c$	p → producer instruction of data c → consumer instruction of data
Memory dependence	$M2_p \rightarrow M2_c$	p → producer store instruction c → consumer load instruction

Each node in the graph represents an event when the instruction *enters* a given pipeline stage. For every non-memory instruction in the dynamic instruction stream, there is a fetch (F) node, a dispatch (D) node, an issue (I) node, a register-read (R) node, an execute (E) node, a write-back (W) node and a commit (C) node. There are two additional nodes, namely mem1 (M1) and mem2 (M2) for memory instructions. An edge in the graph corresponds to the dependence relationship between the two events (nodes) because of some resource or program constraint. If there is an edge from node A to node B and the weight on the edge is W, then the event B cannot occur earlier than W cycles after the event A has occurred. For example, if the fetch pipeline depth is 1, then

there is a directed edge with weight 1 from the fetch (F) node to the dispatch (D) node of an instruction. However, there can be multiple incoming edges to a node. In that case, the edge that dominates will decide the time of the event.

The graph is dynamically built along with the simulation of the given design point (processor configuration). As an instruction is committed in the simulator, nodes corresponding to all the pipeline stages through which the instruction passed are added. Then, the edges incoming to these nodes are added according to resource constraints of the processor and dependence constraints of the program. According to the incoming constraints to the nodes, the nodes are scheduled. That is, the time (cycle) of the event corresponding to the node is determined. It should match with the simulator timestamp of the instruction entering the corresponding pipeline stages. Table 2 gives an overview of constraints modeled by past research on criticality models [11][36].

2) Finding the critical path

The critical path is the longest path from the starting node (fetch of the first instruction in the dynamic instruction stream) to the end node (commit of the last instruction in the dynamic instruction stream). The length (sum of weights of the edges) of the critical path is the execution time of the program. Hence, any parameter affecting the critical path would affect the performance of the processor.

3) Profiling the critical path to find the bottlenecks

Each edge in the program critical path is due to some processor resource (size of a structure or width of a pipeline stage) or program dependence (control, data, or memory dependence). Accordingly, we break down the entire critical path and assign contributions to different processor resources and program dependences. The processor resource or program dependence which has the most cycles assigned to it is the most critical bottleneck and the one which has the least cycles associated is the least critical bottleneck. Note that the most critical bottleneck could be a program dependence. This would happen if the resources of the processor are oversized for the program.

5.2 Modeling of the Issue Queue in the Criticality Model

The size of a FIFO structure is straightforward to model in the graph. As an example, consider a reorder buffer of size S. The reorder buffer cannot hold more than S instructions, and instructions enter and leave the reorder buffer in program order. This implies that instruction "i" cannot enter the reorder buffer before instruction "i-S" has left it. This constraint is enforced by inserting an edge between the relevant nodes of instruction "i-S" and instruction "i".[1]

The key point here is that modeling the size S of a FIFO structure is simple because the *critical instruction* with respect to an instruction "i" is easily identified: it is always instruction "i-S".

In contrast, modeling the size of the issue queue is not straightforward because instructions leave it out-of-order. It was not modeled in previous criticality-based models [11] [21] [36]. Yet, the issue queue is one of the most performance-critical components in an out-of-order superscalar processor: the tradeoff between exposing more ILP and minimizing pipeline loop delay (wakeup-select loop) is particularly acute in the issue queue.

Consider an issue queue of size S. As with the reorder buffer, an instruction "i" cannot enter the issue queue before some prior instruction has left it. Unfortunately, because instructions leave the issue queue out-of-order, this critical instruction is not simply "i-S". The critical instruction could be any prior instruction to this point. Our approach is to maintain a list of S prior instructions that are the latest to leave the issue queue, among all prior instructions. That is, the list contains the S latest-departing instructions. The critical instruction is among these S instructions. Namely, the critical instruction is the one that leaves the issue queue first. Its departure makes room for instruction "i". Accordingly, an edge is inserted between the relevant nodes of this critical instruction and instruction "i".[2]

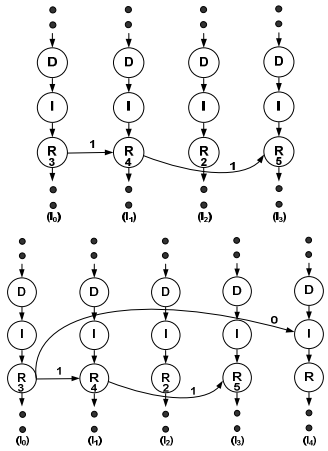

Figure 4. Modeling the issue queue in the criticality model.

The list of S instructions must be updated after each instruction "i" is added to the graph. This is simple to do: the critical instruction as identified above is removed from the list and instruction "i" is added to it. The critical instruction is the earliest-departing one and instruction "i" will necessarily have a later departure (since the critical instruction made way for it). Thus, the new list of S latest-departing instructions excludes the critical instruction and includes instruction "i".

Figure 4 (top) shows the criticality graph corresponding to instruction trace I_0, I_1, I_2, I_3. Note that the number on the edge is the weight of the edge and the number on the node is the time stamp corresponding to that node. For simplicity, we have not drawn the incoming issue queue edges for I_0, I_1, I_2 and I_3. The issue queue size is assumed to be 3. When the instruction I_4 is committed, the nodes corresponding to it are added and edges

[1] In particular, the edge is between the commit (C) node of instruction "i-S" and the issue (I) node of instruction "i": $C_{i-S} \rightarrow I_i$. Note that the commit node corresponds to an instruction leaving the reorder buffer and the issue node corresponds to an instruction entering the reorder buffer and other structures (issue queue and possibly the load or store queue).

[2] In particular, the edge is between the register-read (R) node of the critical instruction and the issue (I) node of instruction "i": $R_{critical} \rightarrow I_i$. Note that the register-read node corresponds to an instruction leaving the issue queue and the issue node corresponds to an instruction entering the issue queue.

corresponding to pipeline dependences are added. The latest three instructions to be issued are I_0, I_1 and I_3 (set S = {I_0, I_1, I_3}). I_0 has the smallest register read (R) time stamp in set S. Hence, to model the issue queue bottleneck, we have an edge from R node of I_0 to I node of I_4. Figure 4 (bottom) shows the criticality graph corresponding to instruction trace I_0, I_1, I_2, I_3 and I_4.

5.3 Illustrative Example of Criticality Analysis

Dynamic Instruction trace

I_0 :R5=0
I_1 :R3=1
L1: I_2 :R1=R3+2
I_3 :R6=R1+2
I_4 :R3=R3+1
I_5 :R5=R6+R5
I_6 :cmp R7,0
I_7 :br L1
I_8 :R5=R5+100

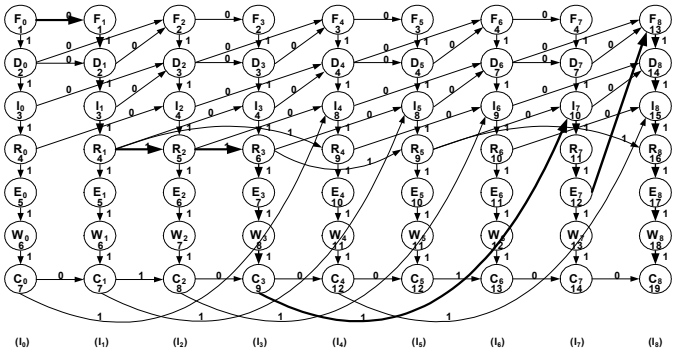

Figure 5. Criticality model of the dynamic instruction trace.

The resource constraints modeled are follows: ROB size = 4, issue queue size =2, issue width =2, fetch width =2, commit width =2. The critical path model for the dynamic instruction trace for above resource constraints is shown in Figure 5. The longest path from fetch node of I_0 (F_0) to the commit node of I_8 (F_8) is shown in bold color. This path is the critical path and has nodes F_0, F_1, D_1, I_1, R_1, R_2, R_3, E_3, W_3, C_3, I_7, R_7, E_7, F_8, D_8, I_8, R_8, E_8, W_8, C_8. The critical path is profiled and cycles are assigned to different processor resources and program dependences as follows: Register dependence = 2, Control dependence = 1, ROB = 1, unclassified = 15. Note that the unclassified cycles are the cycles attributed to pipeline dependences which cannot be attributed to any of the processor resources or program dependences. We clearly see that the register dependence is the most critical bottleneck. Note that many of the processor resources and program dependences have 0 cycles assigned to it because of the very short dynamic trace

5.4 Overhead of Criticality Analysis

Simulation time overhead

Criticality analysis requires the processing of timestamps of instructions and bottleneck events (branch mispredictions and cache misses) to build the dependence graph. Hence, criticality analysis takes lesser time compared to the time taken for cycle-level simulation. In our simulations, criticality analysis takes less than 3% of the time taken to perform cycle-accurate simulation.

Storage overhead

To perform the criticality analysis, the entire dependence graph obtained had to be built and stored. This incurs a large storage overhead. For example: a trace of 100 million instructions might take up to 10 GB of storage space. As done in [26], we remove this overhead by not storing the entire graph and performing criticality analysis as the graph is being built. This is done by realizing that the edges to the nodes corresponding to future instructions come from only a handful of nodes. We see that the storage requirement of the critical path analysis model is of the order of number of constraints modeled. Hence, the storage requirement of the graph is drastically reduced.

Design effort overhead

We need the timestamps of the instructions entering different pipeline stages and the information of different bottleneck events like branch mispredictions, icache miss, dcache miss, etc. We can then build the dependence graph from this information. Hence, it requires minimal effort to retrofit the simulator to get the critical path information.

6. EXPERIMENTAL METHODOLOGY

This section describes the design space of superscalar processors and the simulation environment used for evaluation.

Design Space

Our design space consists of design points created by varying the clock period and complexity (structure sizes and pipeline widths) of different pipeline stages of a superscalar processor. The design space is spanned by the independent design parameters as enumerated in Table 3. Note that the depths of each pipeline stage are dependent parameters as explained in Section 2. The depth of a canonical pipeline stage for a given design point is the degree of sub-pipelining required so that the given complexity of the pipeline stage may fit in the given clock period.

Our design space consists of *3.6 million design points*. By design space pruning as explained in section 4, we reduce the design space to about *104K design points*. Using design space pruning, we have eliminated nearly 97% of the design points from the original design space and this helps in fast design space exploration. Note that in our evaluation, we used the pruned design space for both criticality-driven design space exploration and baseline design space exploration.

Simulator and Benchmarks

For our evaluation, we used the SPEC 2000 benchmark suite with the given reference inputs [33]. We simulated the instruction trace using SIMPOINT [31], so that our simulation run is representative of the entire benchmark. Our simulator is a timestamp, trace-driven simulator based on SimpleScalar [2]. We use a trace-driven simulator to save time in the evaluation to make detailed evaluation of design space exploration possible. Note that using a trace-driven simulator does not affect the general conclusions of the paper as we only treat it as a "black-box" to get the performance value.

Metric

In our evaluation, the design space exploration is performed for optimizing the performance of the superscalar processor. We use instructions per time unit (IPT) to quantify the performance of the superscalar processor. Our metric for the design space exploration is the exploration time which is the amount of time elapsed before

a design point is reached whose performance value is within 0.5% of the optimal performance value [10]. The optimal design point is found by doing an exhaustive design space search. The exploration time is normalized to the time taken by a single simulation of bzip benchmark. Note that the exploration time includes the time taken to perform criticality analysis in the case of criticality-driven design space exploration.

Table 3. Microarchitectural design parameters

Parameter	Value Range	Number
Front end width	2, 4, 6, 8	4
Issue width	2, 4, 6, 8	4
Register File size	32, 64, 128, 256, 512	5
Issue Queue size	16, 32, 64, 128	4
Load Queue/Store Queue size	8/8, 16/16, 24/24, 32/32, 40/40, 48/48, 56/56, 64/64	8
Instruction cache size	8KB, 16KB, 32KB, 64KB, 128KB	5
Data cache size	8KB, 16KB, 32 KB, 64 KB, 128KB	5
L2 cache size	0.5MB, 1MB, 2MB, 4MB	4
Clock period	0.3→0.95ns (granularity=0.05ns)	14

7. RESULTS

In this section, we present the performance results and sensitivity studies with the criticality-driven design space exploration. In Section 7.1, we discuss the performance results of criticality-driven walk and criticality-driven simulated annealing. In Section 7.2, we show sensitivity studies by changing the starting point, changing the size of the design space, and adding power constraints.

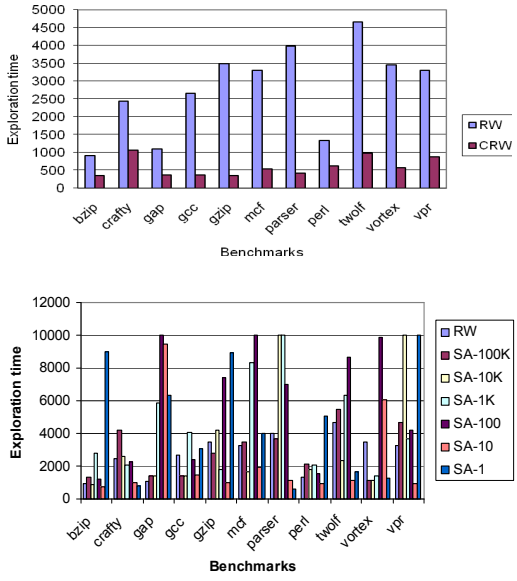

Figure 6. Comparison of criticality-driven walk (CRW) vs. random walk (RW) (top) and the effect of changing the initial temperature on simulated annealing (SA) (bottom).

7.1 Performance of Criticality-driven Design Space Exploration

7.1.1 Criticality-driven Walk

In Figure 6(top), we see that for all benchmarks, criticality-driven walk (CRW) reaches the optimal point much faster than random walk (RW). On average (harmonic mean), criticality-driven walk obtains 3.8x speedup over random walk. For some benchmarks like gzip and parser, the speedup obtained is close to 10x. This shows that the criticality information is very useful in accelerating random walk.

7.1.2 Criticality-driven Simulated Annealing

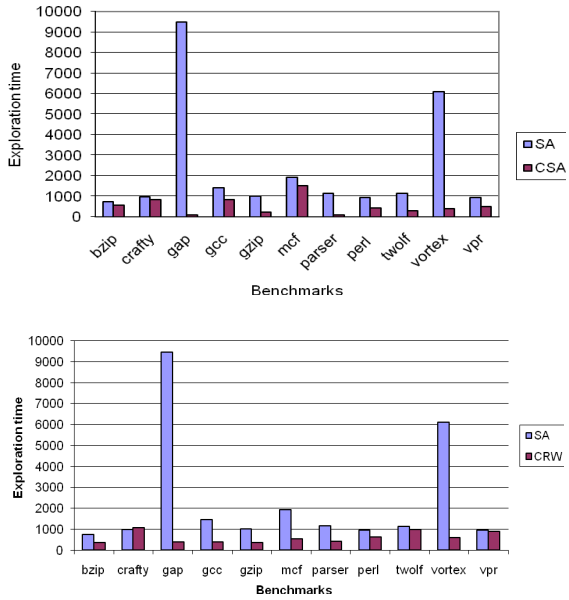

Figure 7. Comparison of criticality-driven simulated annealing (CSA) vs. simulated annealing (SA) (top) and comparison of simulated annealing (SA) and criticality-driven walk (CRW) (bottom).

For simulated annealing to be effective, the initial temperature should be carefully set. For finding a good initial temperature, we use the initial temperature of 1 to 100K. In Figure 6(bottom), we see that there is no single initial temperature which is good for all the benchmarks. However, an initial temperature of 10 is good for many benchmarks. *Hence, we use an initial temperature of 10 for all our evaluations*. Note that a significant amount of time must be invested in temperature tuning to make simulated annealing better than random walk.

In Figure 7(top), we see that using criticality information helps to speed up simulated annealing. On average (harmonic mean), we see criticality-driven simulated annealing (CSA) obtains 2.3x speedup over simulated annealing (SA). Even after removing vortex and gap, CSA obtains 1.9x speedup over SA. Using criticality analysis to guide simulated annealing helps to reach optimal point faster than conventional simulated annealing.

As pointed out before, simulated annealing requires significant temperature tuning to outperform random walk. In Figure 7(bottom), we compare criticality-driven random walk and conventional simulated annealing and see that criticality-driven

walk is better than or comparable to simulated annealing for the SPEC 2000 benchmark suite. Hence, this shows that building the criticality model is more useful than using more sophisticated algorithms like simulated annealing which are sensitive to tuning.

7.1.3 Dynamic Behavior of Criticality-driven Design Space Exploration

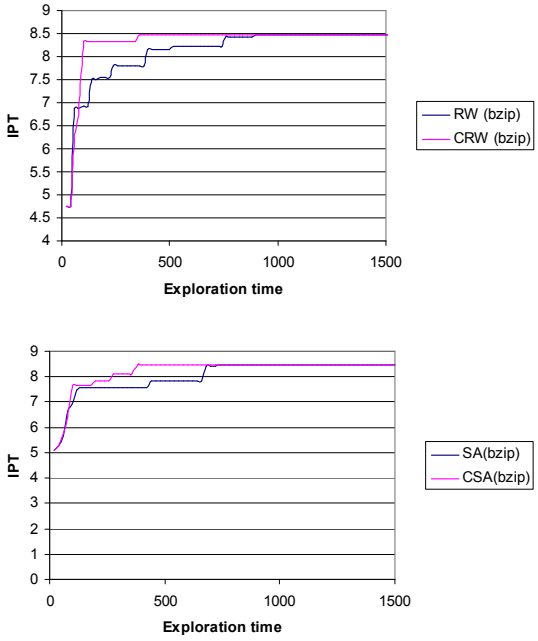

Figure 8. Variation of performance of the best design point found with exploration time, for criticality-driven walk (CRW) vs. random walk (RW) (top) and for criticality-driven simulated annealing (CSA) vs. simulated annealing (SA) (bottom), for bzip benchmark.

To understand how the performance of the best design point found changes with the exploration time for criticality-driven design space exploration, we plot the Instructions per time unit (IPT) of the best design point found with the exploration time, for criticality-driven walk and criticality-driven simulated annealing in Figure 8. Due to space constraints, we only show the graph for the bzip benchmark. However, other benchmarks also show similar trends. We see that localized information from criticality analysis helps criticality-driven design space exploration to reach better design points more quickly.

7.2 Sensitivity Studies

To understand how robust our technique is, we perform sensitivity studies by changing the starting point and the size of the design space, and adding power constraint.

7.2.1 Changing the Starting Point

We choose two other random starting points. Figure 9(top) shows the normalized exploration time needed for criticality-driven walk (CRW) and random walk (RW) for the two starting points. We see that criticality-driven walk outperforms random walk for all benchmarks except gcc in the second starting point. In Figure 9(bottom), we see that criticality-driven simulating annealing performs better than simulated annealing for all benchmarks except mcf and vpr in the first starting point. Because of non-deterministic behavior (random perturbations) of random walk/simulated annealing, in very few cases, it might go through a slightly shorter route to the optimal design point. However, in general, criticality-driven design space exploration is much better than conventional design space exploration.

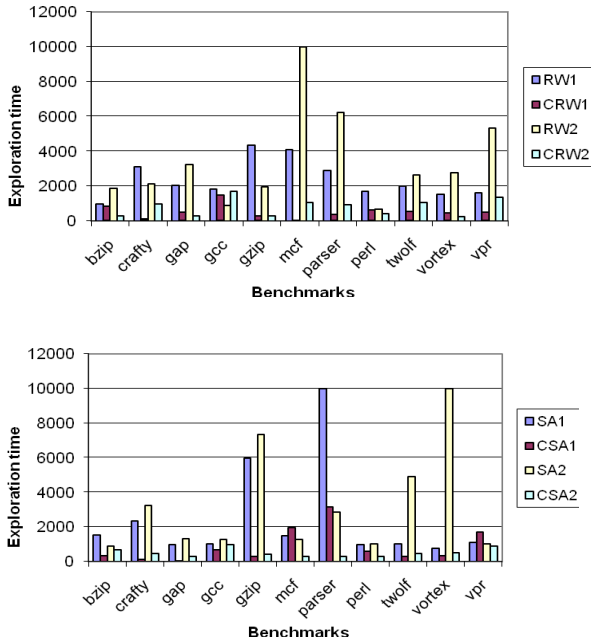

Figure 9. Effect of changing the starting point on criticality-driven walk (CRW) (top) and criticality-driven simulated annealing (CSA) (bottom).

7.2.2 Changing the Size of the Design Space

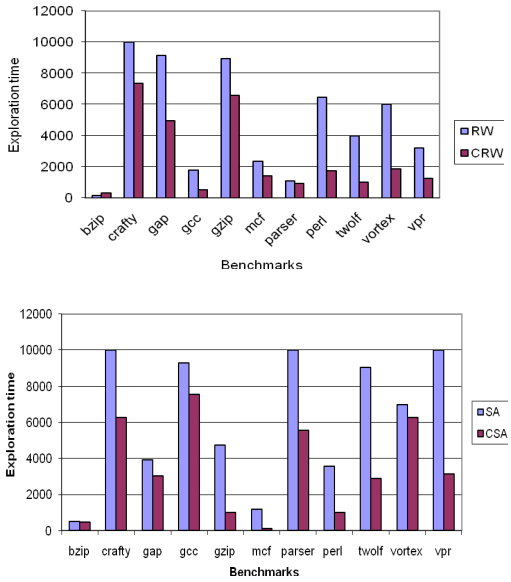

Figure 10. Effect of making the size of design space smaller on criticality-driven random walk (CRW) (top) and on criticality-driven simulated annealing (CSA) (bottom).

We decrease the size of the design space to about 60,000 design points by increasing the clock period granularity from 0.05ns to 0.1ns. In Figure 10, we see that even for the smaller design space, criticality-driven walk and criticality-driven simulated annealing perform better than random walk and simulated annealing, respectively. Further, we see that the temperature selected for simulated annealing for the large design space is not suited for the small design space, as simulated annealing performs worse than random walk on average.

7.2.3 Adding Power Constraint

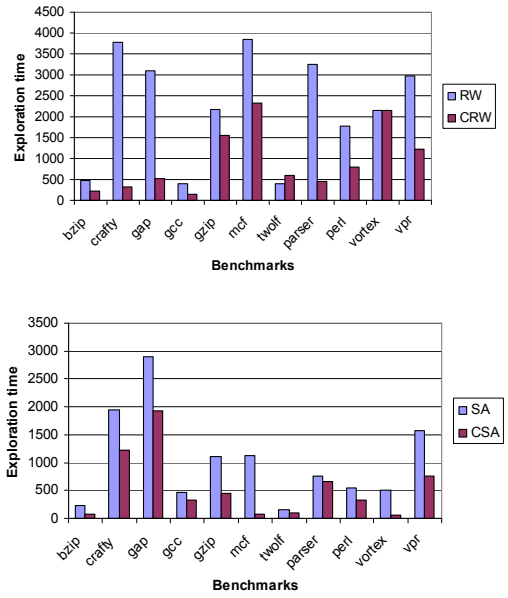

Figure 11. Effect of adding peak power budget of 50 W on criticality-driven walk (CRW) (top) and on criticality-driven simulated annealing (CSA) (bottom).

For modeling power for a given processor configuration, we used Wattch [4], an architecture-level processor power estimation tool. We modified the underlying CACTI [32] models used in Wattch, to incorporate the device parameters from FabScalar [6]. Further, we added power models of combinational logic using the FabScalar RTL model. We calculate the peak power of the processor by assuming all the components, memory structures and combinational logic elements, are switching with an activity factor of 1.

For the sensitivity analysis, we constrain the design space by keeping the peak power budget of the processor at 50 W. In Figure 11, we see that even after adding the power constraint, criticality-driven design space exploration performs better than conventional design space exploration. However, it is slightly less effective than before. This is because some of the criticality-driven perturbation leads to a design point which exceeds the power budget and hence in this case, criticality-driven design space exploration has to rely on random perturbation.

8. RELATED WORK

Design Space Exploration

Accelerating design space exploration is a well studied topic. As pointed out in Section 1, there are two orthogonal approaches for accelerating design space exploration: speeding up the simulation of a single design point and reducing the number of points to be searched.

For accelerating the simulation of a single design point, there are two methods: analytical methods and sampling methods. In analytical methods, an analytical expression is obtained for the performance. Karkhanis' analytical model [19] is a direct approach for obtaining the analytical expression. It expresses the IPC in terms of steady state IPC and penalties due to branch mispredictions and cache misses. A number of methods use an indirect method to obtain an analytical expression by sampling the design space and fitting an analytical model to the data [17][16][9][25]. Lee et al. [25] use a regression model and Ipek et al. [16] use artificial neural networks to get the analytical expression. Analytical methods trade accuracy for speed. The average error rates of these methods are between 3% and 7%. However, the worst-case error rate is a lot higher and may yield an erroneous conclusion about the optimal design point. Sampling techniques reduce the number of instructions needed to be simulated while still being representative of the benchmark [8][27][31][37]. This technique is orthogonal to our approach and in fact we use SIMPOINT [31] to accelerate the time taken to simulate a given point.

An orthogonal approach for accelerating design space exploration is to avoid exhaustive search and concentrates on reducing the number of design points that needs to be simulated [18][10][14][8]. Classical search/optimization techniques like simulated annealing, genetic algorithms, etc. belong to this category. Criticality-design space exploration builds on these classical search/optimization techniques to make them even faster.

Yi et al.[38] proposed using Plackett and Burman design to find the most critical parameters in the entire design space. Chow et al.[7] and Cai et al.[5] use principal components analysis and multivariate analysis to identify the most critical parameter in the design space. This can help pruning the design space by focusing on the most important parameters. On the other hand, our technique finds the most critical parameters at a design point (local level) which is not possible using the above methods. This fine-grained information is needed to find the bottleneck at the current design point which helps to accelerate the design space exploration.

Criticality Analysis

Criticality Analysis is very useful in improving performance in a highly concurrent system like a computer system. It has been proposed to be used inside the processor in the form of a criticality predictor which is used to identify critical instructions to improve performance [11][12][34]. It is used for better resource arbitration by giving priority to critical instructions [11][34]. It can also be used for misspeculation reduction by restricting speculation to critical instructions [11].

It can also be used offline for bottleneck analysis in the computer system [12][29][30]. It has been successfully used to identify bottlenecks in processor architectures like TRIPS, clustered architectures, etc.[1][26][35].

In a simpler design space, criticality analysis has been directly used for design space exploration [22]. However, in a more complicated superscalar design space, using only criticality analysis leads to a local maximum which is a sub-optimal design.

Our work uses criticality analysis in conjunction with the classical search/optimization techniques to avoid getting stuck in a local maximum. Our work opens up another line of research by using criticality analysis in conjunction with the classical search techniques for fast design space exploration.

9. CONCLUSIONS

Because of the exploding design space, increasing design complexity and long-running workloads, design space exploration of a computer system takes a lot of time. Criticality-driven design space exploration uses the localized information from the criticality analysis to guide the globally-aware classical search/optimization techniques. This is a promising direction of research for accelerating design space exploration.

We have just scratched the surface of criticality-driven design space exploration. Using more sophisticated critical path models such as slack and interaction cost model [12] can help to speed up the design space exploration even more. It would also be very interesting to use criticality analysis to speedup the design space exploration of multi-core processors, SMT processors and memory systems.

10. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. Devesh Tiwari, Ganesh Krishnan, Siddhartha Chhabra, Elliott Forbes, Hashem Hashemi, Abhishek Dhanotia and Rajesh Vanka also provided numerous suggestions that improved this work.

This research was supported by NSF grant no. CCF-0811707, Intel and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

11. REFERENCES

[1] M. Agarwal, N. Navale, K. Malik, M. I. Frank. Fetch-Criticality Reduction through Control Independence, In ISCA 2008.

[2] T. Austin, E. Larson, D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling, IEEE Micro, Feb. 2002.

[3] E. Borch, E. Tune, S. Manne, J. Emer. Loose Loops Sink Chips, In HPCA 2002.

[4] D. Brooks, V. Tiwari, M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, In ISCA 2000.

[5] G. Cai, K. Chow, T. Nakanishi, J. Hall, M. Barany. Multivariate Power/Performance Analysis for High Performance Mobile Microprocessor Design, In Power Driven Microarchitecture Workshop, June 1998.

[6] N. Choudhary, S. Wadhavkar, T. Shah, S. Navada, H. Hashemi, E. Rotenberg. FabScalar, In WARP-2009.

[7] K. Chow, J. Ding. Multivariate Analysis of Pentium Pro Processor, In Intel Software Developers Conference, Oct. 1997.

[8] T. Conte. Systematic Computer Architecture Prototyping, PhD. Thesis, Department of Electrical Engineering, UIUC, 1992.

[9] C. Dubach, T. M. Jones, M. O'Boyle. Microarchitectural Design Space Exploration using An Architecture-Centric Approach, In MICRO 2007.

[10] S. Eyerman, L. Eeckhout, K. De Bosschere. Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors, In DATE 2006.

[11] B. Fields, S. Rubin, R. Bodik. Focusing Processor Policies via Critical-Path Prediction, In ISCA 2001.

[12] B. Fields, R. Bodik, M. D. Hill, C. J. Newburn. Interaction Cost: For When Event Counts Just Don't Add Up, IEEE Micro, Nov. 2004.

[13] E. Grochowski, R. Ronen, J. Shen, H. Wang. Best of Both Latency and Throughput, In ICCD 2004.

[14] H. Hashemi Najaf-abadi, E. Rotenberg. Configurational Workload Characterization, In ISPASS 2008.

[15] L Ingber, B Rosen. Genetic algorithms and Very Fast Simulated Reannealing: A Comparison, Mathematical and Computer Modelling, 1992.

[16] E. İpek, S.A. McKee, B.R. de Supinski, M. Schulz, R. Caruana. Efficiently Exploring Architectural Design Spaces via Predictive Modeling, In ASPLOS 2006.

[17] P. J. Joseph, K. Vaswani, M. Thazhuthaveetil. A Predictive Perfomance Model for Superscalar Processors, In MICRO 2006.

[18] S. Kang, R. Kumar. Magellan: A Framework for Fast Muti-core Design Space Exploration and Optimization Using Search and Machine Learning, In DATE 2008.

[19] T. Karkhanis, J. E. Smith. A First-Order Superscalar Processor Model, In ISCA 2004.

[20] T. Karkhanis, J. E. Smith. Automated Design of Application-Specific Superscalar Processors, In ISCA 2007.

[21] T. Karkhanis. Automated Design of Application-Specific Processors, PhD. Thesis, Department of Electrical Engineering, University of Wisconsin-Madison, 2006.

[22] H. Kannan, M. Budiu, J. Davis, G. Venkataramani. Tuning SOCs using the Dynamic Critical Path, In SOCC 2009.

[23] R. Kumar, D. Tullsen, N. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors, In PACT 2006.

[24] P. Laarhoven, E. Aarts. Simulated Annealing: Theory and Applications, Springer, 1987.

[25] B. Lee, D. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction, In ASPLOS 2006.

[26] R. Nagarajan, X. Chen, R. G. McDonald, D. Burger, S. W. Keckler. Critical Path Analysis of the TRIPS architecture, In ISPASS 2006.

[27] S. Nussbaum, J. E. Smith. Modeling Superscalar Processors via Statistical Simulation, In PACT 2001.

[28] E. Rich, K. Knight. Artificial Intelligence, 2nd Edition. Morgan Kaufmann, 1991.

[29] A. Saidi, N. Binkert, T. N. Mudge, S. K. Reinhardt. Full System Critical Path Analysis, In ISPASS 2008.

[30] A. Saidi, N. Binkert, S. K. Reinhardt, T. N. Mudge. End-To-End Performance Forecasting: Finding Bottlenecks before They Happen, In ISCA 2009.

[31] T. Sherwood, E. Perelman, G. Hamerly, B. Calder. Automatically Characterizing Large Scale Program Behavior, In ASPLOS 2002.

[32] P. Shivakumar, N.P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power and Area model, Technical report, 2001.

[33] The Standard Performance Evaluation Corporation, http://spec.org

[34] S. Subramaniam, A. Bracy, H. Wang, G. Loh. Criticality-Based Optimizations for Efficient Load Processing, In HPCA 2009.

[35] P. Salverda, C. Zilles. A Criticality Analysis of Clustering in Superscalar Processors, In MICRO 2005.

[36] E. Tune, D. Tullsen, B. Calder. Quantifying Instruction Criticality, In PACT 2002.

[37] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling, In ISCA 2003.

[38] J. Yi, D. Lilja, D. Hawkins. A Statistically Rigorous Approach for Improving Simulation Methodology, In HPCA 2003.