REDUCED OCCUPANCY FOR PROCESSOR EFFICIENCY THROUGH DOWNSIZED OUT-OF-ORDER ENGINE AND REDUCED TRAFFIC

By

Ziad Youssfi

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
of the degree of

DOCTOR OF PHILOSOPHY

Electrical and Computer Engineering

2011

ABSTRACT

REDUCED OCCUPANCY FOR PROCESSOR EFFICIENCY THROUGH DOWNSIZED OUT-OF-ORDER ENGINE AND REDUCED TRAFFIC

By

Ziad Youssfi

Reducing microprocessor power consumption can alleviate expensive thermal management, improve reliability, allow increased performance, and reduce carbon emissions. Microarchitecture techniques that reduce instruction occupancy in the out-of-order engine to ultimately create more empty segments that can be disabled provide an attractive approach to reducing power consumption. This thesis presents two guiding principles to reduce occupancy for implementing such an approach. One involves downsizing the out-of-order engine as it becomes less effective in issuing instructions out of order. The second reduces instruction traffic by dampening traffic bursts. This thesis developed two techniques that implement these principles. One downsizes the out-of-order engine based on a data dependence index (DDI), and the second moderates the instruction traffic based on the sustained dispatch rate (SDR). The SDR, in addition to reducing occupancy, reduces the i-cache's power consumption. Total processor power savings are reported. Component power savings are also reported to show their relative effects on total savings. Simulations find that occupancy was reduced from 27% to 33% in the out-of-order engine and 40% in the fetch buffer. These occupancy reductions are, to our knowledge, the largest reported to date, with a performance loss of only 2.6%. Reduced occupancy results in fewer active segments, which directly contributes to power savings. Power models estimate that the savings for a single threaded, high performance processor can total to 27% of overall dynamic power consumption, including level-1 caches.

Acknowledgments

I would like to express my deep appreciation to my wife, Maya, whose love gives me comforting perspective on life and outweighs any hardship; to my mother whose love makes me a better person; to my sister, Rasha, and her family, whose support is steadfast; to my father whose pride in me gives me confidence; and to Maya's parents, Andi and Baruch, whose supporting words always remind me that I am on the right path. I also would like to express appreciation to my advisor, Dr. Michael Shanblatt, for taking me as a Ph.D. student and for his patience over the many years it took to finish this dissertation. Finally, I express my appreciation for my committee members, Dr. Richard Enbody, Dr. Erik Goodman, and Dr. Shlomo Levental, for their helpful comments and support along the way.

Table of Contents

List of Tables

List of Figures

1 INTRODUCTION: REDUCING PROCESSOR POWER CONSUMPTION

1.1 Motivation

Steadily increasing with each technology generation, the power consumption of processor chips has become so high that it limits performance of new designs. High power consumption requires expensive heat-sinking and thermal management, and leads to high current and chip hot spots that reduce reliability [1, 2]. At the same time, data centers stacked with high power processors have costly cooling systems, high energy consumption, and greater carbon emissions. In fact, processor power consumption has been declared a grand challenge, and is a problem for which even small savings are considered important gains [3].

The processor power consumption challenge is relatively new in computing. In 1997, when billion transistor chips were becoming feasible, an IEEE Computer Magazine issue surveyed new processor challenges [4]. None of the authors in the issue predicted power consumption to be a challenge. However, by 2004, power consumption was already becoming a challenge for higher performance. A 2004 Computer Magazine retrospective issue discussed how predicting the challenge of power consumption was missed in 1997 [5]. By 2005, the International Technology Roadmap for Semiconductors (ITRS) declared processor power consumption a "grand challenge" for processor design [3].

When processor power consumption is increased, more elaborate heat-sinking and thermal management are required to remove the generated heat in order to keep the chip temperature below its operating limit. Moreover, above a certain power consumption threshold, air-cooling techniques cannot keep the chip below that temperature limit. Resorting to alternative cooling techniques, such as liquid cooling, can result in a sharp increase in the cost of chip package integration [1, 6].

Another aspect of this challenge is environmental. In 2008, the number of PCs in use (as opposed to the number shipped) in the world was estimated at more than one billion units [7]. The Hoover Dam power plant is rated at 4 Mega Horsepower [8]. If we conservatively assume a processor chip consumes on average 75 Watts, then powering the world's PCs we requires at least 33 power plants like Hoover Dam! (Actually, many more than 33 if we consider the system's power consumption and not just the processor's). Moreover, the number of PCs in the world is expected to double by the year 2014 [7]. With the rise in processor power consumption, this growth rate may not be sustainable.

This thesis details microarchitecture techniques to reduce the average power consumption of a processor chip. The microarchitecture techniques presented reduce instruction occupancy in the out-of-order engine components, creating empty segments that can be dynamically shut down to save power. Instruction occupancy is reduced by dynamically down-sizing the out-of-order engine components and by reducing instruction traffic. The same techniques used to reduce traffic are used to cluster more instructions in a single i-cache fetch, thus also reducing i-cache power consumption.

1.2 Thesis Summary

This thesis is divided into six chapters. The problem of increased processor power consumption is introduced in Chapter 1. To show the main sources of power consumption, a basic CMOS circuit design is given in Section 1.3. Factors behind the increasing trend in dynamic and static power consumption as well as the increase in power density are discussed in Sections 1.3–1.5. A summary of the proposed microarchitecture techniques to reduce processor power

consumption is given in Section 1.6. To help establish a context for the proposed techniques, a brief review and a classification of previous related research is also presented in that section. The principles for downsizing the out-of-order engine and for reducing instruction traffic are discussed. The Contribution section also presents a summary of the methods that implement those two principles.

More extensive review of related work on processor power reduction techniques is presented in Chapter 2. A general taxonomy of these techniques is presented in Section 2.1, starting at the device level all the way up to the software level. A brief review of circuit complexity of the out-of-order engine components is presented in 2.2. An in-depth review of power reduction techniques for the out-of-order engine is presented in Section 2.3, where the techniques are classified based on a new definition for occupancy reduction of either neutral or active. Active occupancy reduction techniques are further subdivided into downsizing or traffic reduction techniques. A review of complexity reduction techniques for the out-order engine is given in Section 2.4. Complexity reduction techniques are intended to increase performance rather than reduce power consumption. Nonetheless, including these techniques here offers a more complete review. Chapter 2 ends in Section 2.5 with a review of power modeling at the architecture level.

The methods used to implement the two proposed principles for reducing instruction occupancy are presented in Chapter 3. The principles for downsizing the out-of-order engine and for reducing traffic are restated in Section 3.1. The method that was developed to implement the principle of downsizing the out-of-order engine is detailed in Section 3.2. The three developed variations of the method for reducing instruction traffic are detailed in Section 3.3. One variation of the traffic reduction method exhibits significantly better clustering of fetched instructions from the i-cache. This effect and its benefit for reducing the i-cache power is detailed in Section 3.4. The framework used for architecture models, the power models, and the simulation workload is presented in Section 3.5.

Simulation results are reported in Chapter 4. First, the reductions in instruction occupancy for all out-of-order engine components using all methods are reported in Section 4.1.

The results include average occupancy as well as time distributions for the number of active segments. Traffic reduction results are reported in Section 4.2. In Section 4.3, the results for improved clustering of fetched instructions from the i-cache are presented. Dynamic power savings' results are reported in Section 4.4. In order to show the effect of components on the processor's total dynamic power savings, both component and total dynamic power savings, including level-1 caches, are reported. The effects on performance of all methods are reported in Section 4.5. Validation of the principles and their implementation is discussed in Chapter 5. The significance of the occupancy reduction results is also discussed in Chapter 5.

Finally, conclusions and future research are presented in Chapter 6. Future research discussion includes using the proposed principles and methods for more power efficiency, such as reducing static power in addition to dynamic power. They also include using the proposed methods to improve scheduling and throughput on heterogeneous multi-core processor systems.

1.3 Dynamic Power Consumption

1.3.1 CMOS VLSI Design

To help understand the rising trend in power consumption, we first examine processor circuit design. The dominant VLSI technology for microprocessor integrated circuits has been the Complementary Metal Oxide Silicon (CMOS) [9]. In CMOS two types of transistors (also called devices) are created: an n-type device (nMOS) and a p-type device (pMOS). In general, starting with a silicon wafer, CMOS fabrication involves successive steps of chemical processing to create sandwich-like layers of different conductive properties. For example, diffusion of electron-donor or -acceptor impurities is used to create positively or negatively charged silicon. Oxidation is used to create insulating layers of SiO2. Deposition and etching are used to connect devices with poly-silicon or aluminum much like circuit board components. Chemical etching is guided by mask patterns and lithography techniques.

A layout and cross section of an n-well CMOS process for an inverter gate is shown in **Figure 1**, [9]. In an n-well process, p-devices are created in a p-substrate within an n-type well.

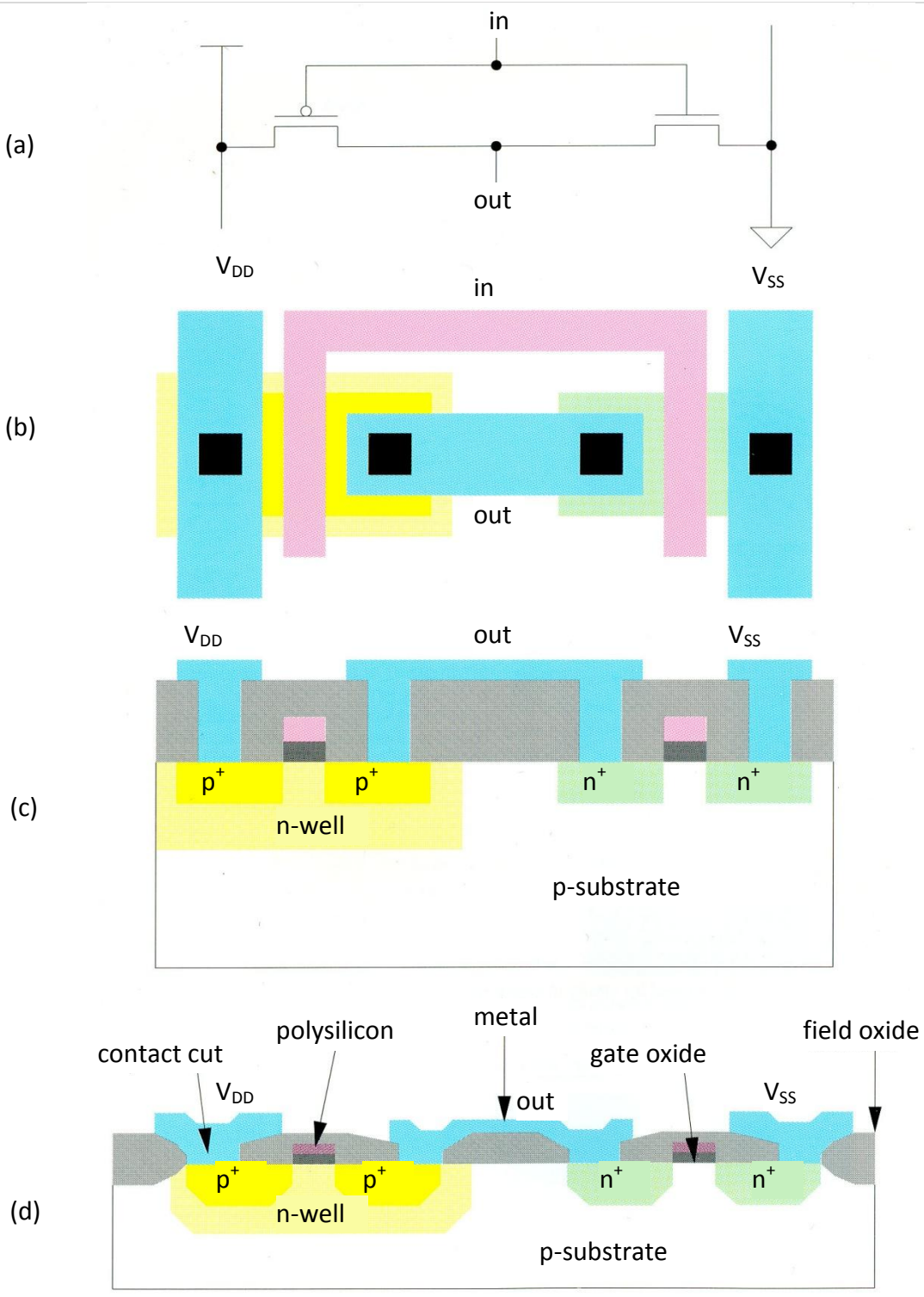


Figure 1. CMOS process for an inverter gate. (a) logic circuit diagram, (b) layout top view, (c) abstract cross-section view, and (d) actual cross section view, [9]. For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.

The n+ and p+ regions serve as the current source and drain for the n and p-devices. Their polarity is interchangeable, based on the direction of current. The applied gate voltage to the polysilicon above the gate oxide controls the conduction of current between the source and the drain. When high voltage (logic 1) is applied to the inverter input, the p-device is off, and the n-device conducts current to ground, exerting logic 0 at the output. When low voltage is applied to the input, the n-device is off, and the p-device conducts current from the voltage supply, exerting logic 1 at the output. The complementary use of n- and p-devices allows a solid sourcing and sinking of current from power supply and ground, respectively. This maintains reliable signal propagation against noise and is one advantage of CMOS.

Another advantage that CMOS had when it was introduced was lower power consumption than that of other VLSI technologies [9]. Because of CMOS high input impedance on device gates, a negligible current is drawn by the gate at the input when it is not switching, which saves power. However, CMOS circuits still consume power in two ways. One is dynamic power dissipation to charge and discharge the output capacitive load when the input changes. The second is static power dissipation due to substrate leakage current.

1.3.2 Estimating Dynamic Power

When the input logic changes from '1' to '0' for an inverter gate (**Figure 2**), the n-device is turned off, and the p-device is turned on and has to charge the capacitive load, C_L. The capacitive load

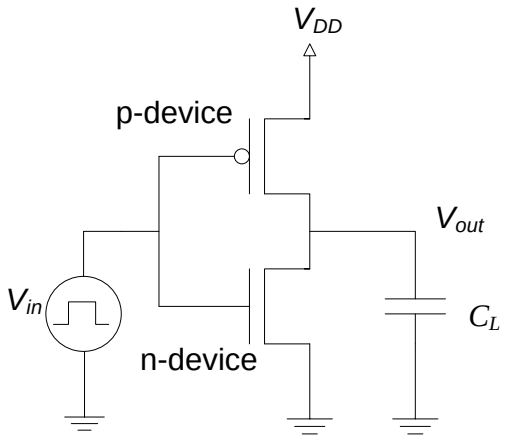


Figure 2. Logic circuit diagram for an inverter gate with a capacitive load, C_L, and repetitive step input, V_{in}.

represents the fan-out capacitive load of later logic stages. When the inverter input changes from '0' to '1', the p-device is turned off and the n-device is turned on and has to discharge C_L. This charging and discharging of current consumes dynamic power, which constitutes the largest portion of power consumption for CMOS.

To calculate average dynamic power for an inverter gate, we assume a repetitive step input, such as clock. If the input frequency $f_p = 1 / t_p$, then the average power consumed can be calculated by integrating the product of current and voltage over the period, t_p. This integral can be broken down into two integrals. The first integrates the current through the n-device and the voltage across it over a half period (0 to $t_p / 2$). And the second integrates the current through the p-device and the voltage across it over the second half period ($t_p / 2$ to t_p). This is expressed as:

$$P_d = \frac{1}{t_p}\int_0^{t_p/2} i_n(t)\, V_{out}\, dt + \frac{1}{t_p}\int_{t_p/2}^{t_p} i_p(t)(V_{DD} - V_{out})\, dt \qquad (1)$$

where

$$i_n(t) = \text{n-device transient current} = C_L\, dV_{out}/dt$$

$$i_p(t) = \text{p-device transient current} = C_L \frac{d(V_{DD} - V_{out})}{dt}.$$

Substituting the current values and changing the variable of integration from time to voltage, the following is obtained:

$$P_d = \frac{C_L}{t_p}\int_0^{V_{DD}} V_{out}\, dV_{out} + \frac{C_L}{t_p}\int_{V_{DD}}^{o} (V_{DD} - V_{out})\, d(V_{DD} - V_{out})$$

$$P_d = \frac{C_L\, V_{DD}^2}{t_p} = C_L\, V_{DD}^2\, f_p\,. \qquad (2)$$

One observation that can be made from equation (2) is that the average dynamic power scales linearly with capacitive load and input frequency, and scales quadratically with the supply voltage. Another observation is that power consumption is independent of device parameters.

The average dynamic power was calculated above for a single inverter with a repetitive step input to switch charge on a load capacitance. For simulating a system with a large number of gates (e.g., thousands or millions) with non-repetitive inputs, the above calculations are too complex and impractical to carry out for each gate. A faster estimate can be obtained by using the total capacitance, C_{total}, for all gates in the circuit and the percent of time during which they are active expressed as:

$$P_d = \frac{percent\_activity \times C_{total} V_{DD}^2}{t_p}. \quad (3)$$

The accuracy of equation (3) depends on the accuracy of the active time percentage, which is only an approximation. A more accurate power estimate can be obtained during simulation by summing up capacitances only for components that switch at any given cycle. To estimate power consumption to run a program on a processor, for example, the total switched capacitance, $C_{total\_switched}$, can be summed up for switching components during simulation. At the end of simulation, the total switched capacitance can be used along with the number of total cycles to run the program, N_{total_cycles}, to estimate power consumption:

$$P_d = \frac{C_{total\_switched} \ V_{DD}^2}{N_{total\_cycles} \ t_p} \quad (4)$$

Equation (4) can be used to show how different processor design parameters affect dynamic power consumption, P_d, as will be discussed in the next section.

1.3.3 Increasing Trend

Microarchitecture techniques that increase performance generally require physically large units (typically arrays) due to a high device count. The high device count and the long transmission lines within the arrays quickly add to $C_{total\_switched}$ in equation (4). Capacitive cross-coupling among transmission lines also quickly adds to $C_{total\_switched}$, especially with new generations of fabrication technology, due to closer physical proximity of devices. Achieving higher performance also requires decreasing the number of total cycles, N_{total_cycles}, as well as decreasing the cycle time, t_p, in order to decrease runtime, which also rapidly increases P_d. Combining all these variables changes together results in a superlinear increase in P_d. **Figure 3**

shows how clock frequency is increasing (or how t_p is decreasing) in relation to Intel processor generations [10].

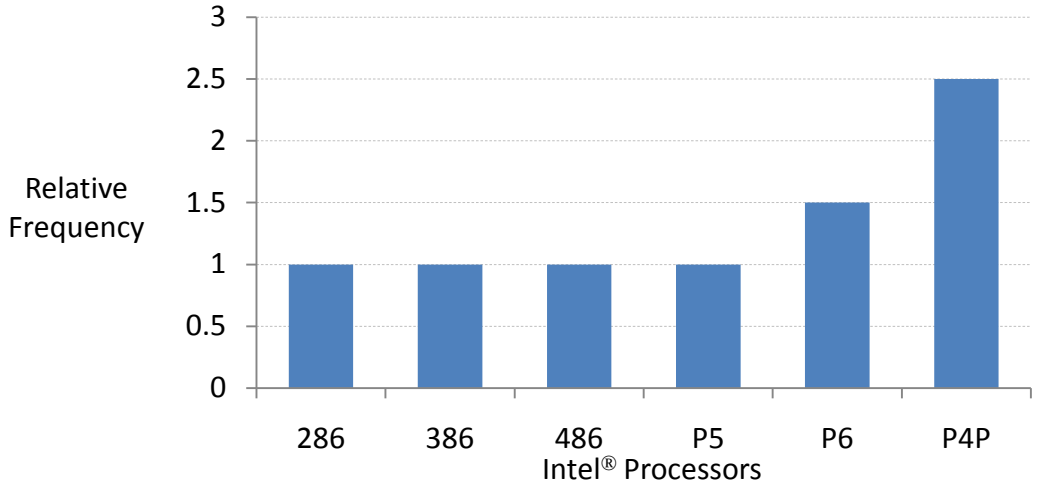


Figure 3. Relative increase in clock frequency for Intel Processors (Data source from Hinton et al. [10]).

Reducing the supply voltage, V_{DD}, can reduce dynamic power quadratically. However, reducing V_{DD} leads to an exponential increase in leakage current and static power consumption. This will offset any decrease in dynamic power.

In fact, V_{DD} had been scaling down about 30% per technology generation, but now has

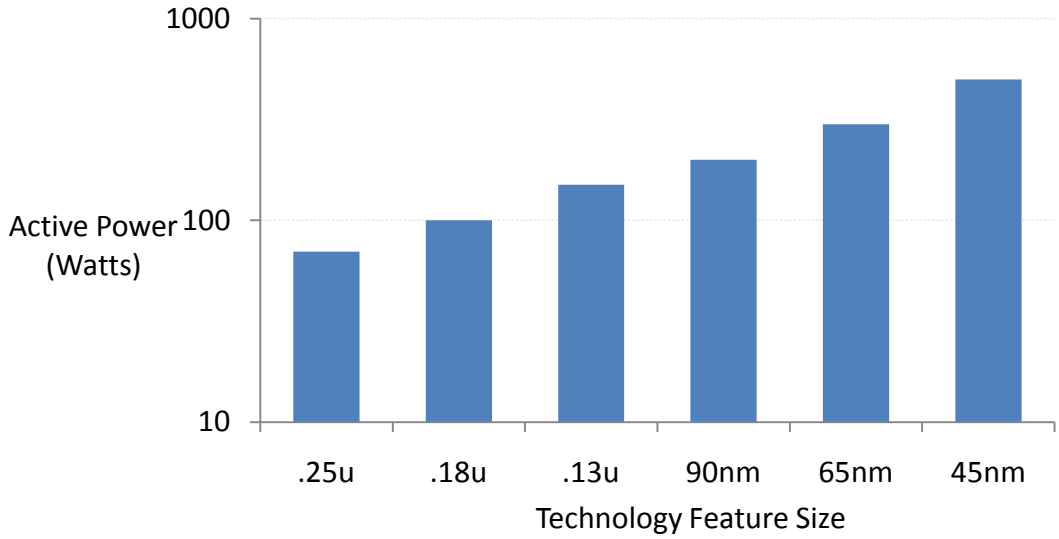


Figure 4. The rise in dynamic power consumption with technology generations for Intel® processors (Data source from Borkar [11]).

bottomed out because of this offsetting effect [11]. Even if V_{DD} continued to scale down by 15% per generation, clock frequency is increasing simultaneously by 40% [11]. At this increasing clock frequency rate, P_d can reach prohibitive levels. **Figure 4** shows the trend for dynamic power for Intel processors as a function of technology generations. Some of the recent processors fabricated in 32nm technology running at over 3 GHz clock frequency, such as some of the Intel Xeon 5000 series processors, consume well over 100 Watts [12].

1.4 Static Power Consumption

Fundamentally, static power consumption is due to leakage current at sub-threshold voltage in reverse biased parasitic diodes. These parasitic diodes, which are inherent in silicon CMOS, are formed between diffusion regions and the substrate [9]. Unlike dynamic power, static power is always dissipated whether devices are switching or not. The total static power, P_s , dissipated for a processor is independent of time, and it is summed up over all devices as:

$$P_s = \sum_{1}^{n} \text{leakage current} \times \text{supply voltage} \tag{5}$$

where

$$n = \text{number of devices.}$$

Static power consumption, just like dynamic power, has also shown an increasing trend. The reason for this increasing trend is twofold. First, scaling down V_{DD} to reduce dynamic power has the undesired side effect of increasing static power. As V_{DD} is reduced, the threshold voltage also has to be reduced. The leakage current for the parasitic reverse biased diodes increases exponentially with the decreasing threshold voltage. With Intel processors, for example, a continued decrease in supply voltage for fabrication technology of less than 90 nm feature size would actually result in static power accounting for more than 50% of total chip power dissipation. This trend is shown in the graph of **Figure 5**.

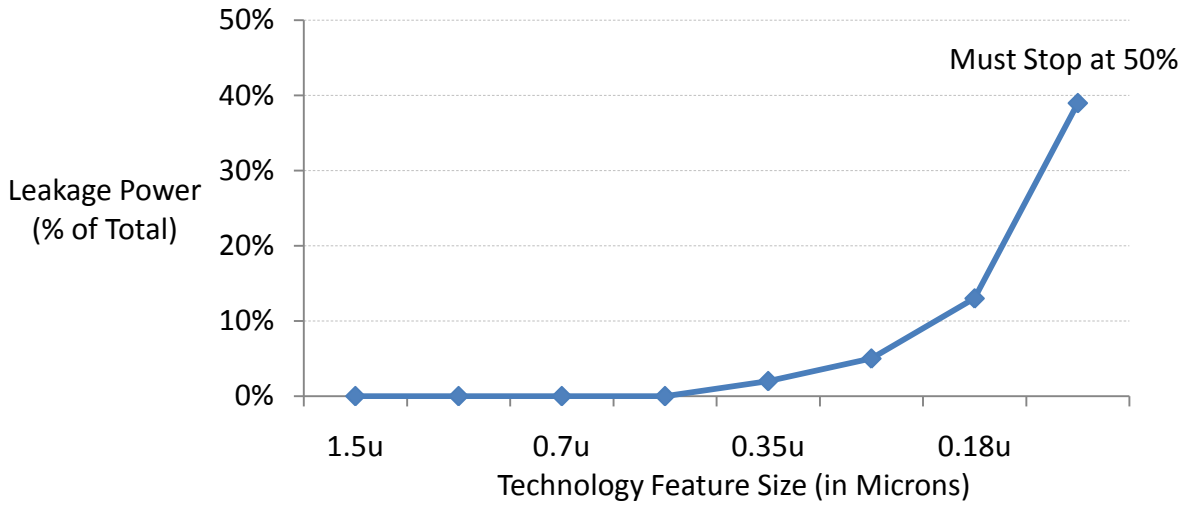


Figure 5. Percentage of total chip power consumption due to leakage current as a function of technology feature size.(Data source from Borkar [11] and Grove [13]).

The second reason for the increase of static power is the increase of the number of devices on processor chips. With the doubling of devices per technology generation, the sum of all the leakage currents for these devices, as indicated in Equation (5), is quickly reaching a critical level.

1.5 Power Density

Increasing the number of devices on a chip and increasing the clock frequency with every technology generation increases both the total chip power consumption and the power consumption density per unit area. As more heat is generated from more devices crammed per unit area, the heat can detrimentally affect the operating temperature in certain areas of the chip. **Figure 6** shows the trend over time of the power density and equivalent temperature densities for Intel processors [13]. These high heat densities, as high as a rocket nozzle, cannot be sustained on a silicon chip under normal operations.

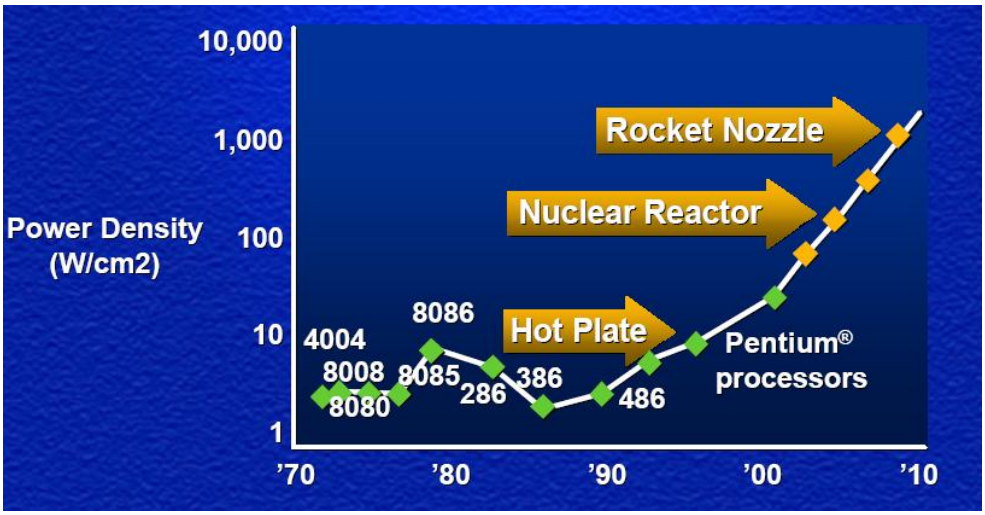


Figure 6. Power density for Intel processor generation over years and equivalent temperature density. (Data source from A. Grove [13])

Since processor chip power dissipation is not spatially or temporally uniform, localized heating areas, in the form of hotspots, occur on a shorter time scale than chip-wide heating. The time scale for these hotspots is on the order of one hundred microseconds to milliseconds. However, the rapid increase in power density is such that the temperature at these hotspots can cause timing errors or even physical damage [2, 14].

1.6 Thesis Contribution

1.6.1 Current Occupancy Reduction Techniques

The out-of-order execution engine consumes a large proportion of the power used by a chip. As a result, dynamically shutting down empty segments in its components can reduce power. Shutting down segments that are normally empty, with no additional attempt to reduce instruction occupancy, can be classified as *neutral* occupancy reduction [15, 16]. In contrast, microarchitecture techniques that cause components to reduce instruction occupancy based on dynamic thread behavior can be classified as *active* occupancy reduction. As a thread executes, its code behavior and its demands on different components change. This dynamic creates opportunities to actively reduce occupancy, creating additional empty segments. The challenge is to dynamically detect when such behavioral changes occur, so as to determine when a

component's occupancy can be reduced to save energy and when it should be increased to protect performance. Active occupancy reduction offers more attractive power savings than neutral occupancy reduction does [17]. However, current approaches have yet to fully realize that potential.

One approach to active occupancy reduction is downsizing triggered by changes in a general performance metric such as the IPC (instructions commit per cycle) [18, 19]. However, the IPC is only a symptom of performance determined by many behaviors, and it can itself be penalized by downsizing. To minimize the penalty, a downsized component must periodically be upsized to measure any effect on the IPC. This trial-and-error approach requires long "tuning" periods (on the order of hundred-thousands to millions of cycles) in order to determine the optimal size for a component, especially one that has many size configurations. With often short-lived execution behaviors, this approach can miss power saving opportunities [20].

A second approach to active occupancy reduction is reducing traffic to the out-of-order engine, for example by monitoring behavioral changes in branch prediction and throttling the fetch when the front-end encounters highly mispredicted branches [21]. However, reducing traffic alone will not always reduce occupancy. A queue's occupancy might be reduced when traffic going into it is reduced, but if the queue's service rate is also reduced, the queue steady state remains at high occupancy. Moreover, using solely one behavior, such as branch prediction, does not take advantage of other behavioral changes.

1.6.2 New Principles for Active Occupancy Reduction

Because both downsizing and reducing traffic can reduce occupancy, we propose an approach that combines them, while overcoming some of their individual limitations. Specifically, we propose improvements to downsizing and traffic reduction based on two principles:

Principle 1: Gradually downsize the out-of-order engine as it becomes less effective in issuing instructions out of program order. There are two states in which queues in the out-of-order engine become less effective. One occurs when instructions line up in a data dependence chain. The more instructions line up, the more rapidly performance becomes insensitive to

instruction window size. That insensitivity can be exploited by setting a smaller window that yields the same performance. The second ineffective state is the opposite, when instructions become virtually independent. Then they need much less (if any) out-of-order movement, making large queues energy inefficient. In both states, components can be forced to reduce their occupancy by downsizing. Components should maintain maximum size when the engine holds a roughly equal mix of independent and dependent instructions, in order to allow out-of-order issue. The sizing decision must be solely based on the state of data dependences; otherwise, the engine's ability to find and issue independent instructions might be hindered.

Principle 2: Continuously moderate the front-end pressure to reduce instruction traffic. The front end is normally designed to fetch instructions from the i-cache and push them into the out-of-order engine as fast as possible. However, the engine may not always be able to move instructions as fast as they come in. It might be slowed down, for example, by data cache misses, function unit latencies, and resource conflicts. When that happens, the engine is under constant front-end traffic pressure to raise its occupancy. By moderating front-end pressure, traffic into the out-of-order engine can be reduced.

1.6.3 Implementation of Principles

This thesis reports two methods implementing these two principles in order to reduce occupancy in all of the out-of-order engine components. The first downsizes components based on a newly developed data dependence index (DDI). The DDI is a single number that expresses how well instructions line up in a data dependence chain. In previous work, it was shown that a metric based on data dependences can be used alone to reduce occupancy in the out-of-order engine [22]. Here, a new algorithm that uses token-passing in the issue queue to measure the DDI is reported. The algorithm has the advantage of having low hardware complexity and not requiring periodic upsizing to keep components at the most efficient size.

The second method reduces traffic by using the fetch buffer to moderate the front-end pressure, matching it to the currently sustained dispatch rate (SDR). The SDR is the average dispatch rate that the engine can sustain over a short interval. This moderation dampens traffic bursts above the SDR, thereby reducing average traffic. In addition to reducing traffic, this

method provides an energy saving bonus for the i-cache. Because it drains the fetch buffer before fetching is resumed, this method can cluster significantly more instructions in each i-cache access. Because an i-cache access energy cost is the same regardless of how many instructions are fetched, this method significantly reduces i-cache power consumption.

The two methods operate independently but in tandem, matching resources to behavior changes that are both microarchitecture independent and dependent. The downsizing method responds to data dependence changes, which are microarchitecture independent. The fetch rate moderation method responds to the changes that are microarchitecture dependent.

Similar matching is desired with heterogeneous multi-core processing (HMP) where the objective is to match an inherent thread behavior to one of many dissimilar processors. HMP research has found that techniques that directly measure inherent thread behavior changes are more effective than ones relying on a general performance metric such as the IPC [23]. These findings directly support our findings in the power efficiency domain.

To evaluate the methods' power efficiency, the Wattch power models were used [24]. These power models can estimate dynamic power consumption for the SimpleScalar architecture [25]. For a realistic estimate of performance, occupancy, and power consumption, the SPEC CPU 2000 benchmark suite from the University of Minnesota was simulated as a workload [26]. This benchmark suite is derived from SPEC CPU 2000 and is recognized by SPEC as a valid simulation tool for simulation based computer architecture research. The suite uses statistical sampling to maintain function-level execution patterns, instruction mix, and cache behaviors for entire benchmarks. The available suite compatible with the “pisa” SimpleScalar architecture was used. The suite contains a mix of compression, graphics, scientific, and database benchmarks.

Power savings per component as well as total processor savings in dynamic power, including level-1 caches are reported. It is asserted that a meaningful quantitative analysis of efficiency should consider both component as well as total processor savings in dynamic power, in order to show the relative effect of components on the total savings.

2 BACKGROUND & RELATED WORK

This chapter reviews related methods and research on reducing processor power consumption. Section 1 reviews the design levels in which current methods can be grouped, from the device up to the compiler level. Since the focus of this thesis is on the out-of-order engine, Section 2 briefly reviews the function of the out-of-order issue in a typical multi-issue pipeline and its circuit complexity. Section 3 then examines power reduction techniques specific to the out-of-order engine.

The out-of-order engine techniques reviewed in Section 3 are classified based on their approach. This taxonomic classification helps emphasize common strengths that should be kept and common weaknesses that should be addressed in a new design. The classification also helps define a new method that combines the strengths of different classes.

The first level of classification of out-of-order techniques is based on whether they downsize components. Those techniques that downsize components are further classified by whether they are active or neutral regarding occupancy reduction. Active occupancy reduction techniques are further subdivided by whether they use IPC or traffic reduction. Section 3 also discusses the need for a tandem technique that draws on the strength of the active occupancy reduction techniques while addressing their weaknesses.

Section 4 reviews complexity reduction techniques for the out-of-order issue. These techniques' objective is performance rather than power efficiency. Since this thesis' work deals with out-of-order issue design, these techniques are reviewed here for completeness.

Finally, Section 5 reviews power models at the architecture levels, specifically the Wattch power models. These models are critical for a quantitative analysis of efficiency that considers both component as well as total processor power savings. Obtaining the total power consumption helps to show the relative effect of components on the total savings.

2.1 Processor Power Reduction Levels

2.1.1 Device Level

The minimum threshold and supply voltages for a CMOS device can be lowered by reducing the device size, which in turn helps reduce chip dynamic power consumption. Pursuing smaller device sizes through successive silicon fabrication technologies has helped curb the increase in power consumption, but this approach is running into limitations [27].

One such limitation is that lowering the threshold voltage leads to an exponential increase in device leakage current and hence an exponential increase in chip static power consumption [9]. For feature size less than 90 nm, the increase in static power will offset any decrease in dynamic power consumption (Section 1.3.3). Silicon on Insulator (SOI) fabrication technology was proposed to reduce leakage current; however, SOI is subject to other limitations, including higher cost of fabrication than bulk CMOS [9]. Even if leakage current is reduced for feature sizes less than 90 nm, CMOS fabrication technologies are not thought capable of producing feature sizes less than 10 nm [28].

A radical solution to reduced device size limitations is finding alternate materials and devices to those of CMOS. Advances in nanotechnology are helping find potential devices. One such device is the memristor, which is basically a resistor with memory [29]. It can be used as memory storage device; it is denser and faster than flash memory and uses much less power. While it was shown to be feasible for memory storage, a memristor device still requires much more development for use in general logic or processor circuits [28].

Another explored alternative to implement a logic switch is nano-scale germanium-silicon wires that can be used in "nano-processor" tiles [30]. Researchers have also looked at

switching a material state from crystalline to amorphous using electric current in what is called phase-change memories [31]. Using carbon nano-tubes is yet another alternative that is being pursued as alternate logic switch [28]. As no clear alternate device has emerged yet, the challenge remains not just the feasibility of these devices, but also the cost of manufacturing them.

2.1.2 Circuit Level

Circuit techniques to reduce power consumption involve reducing the total switched capacitive load of devices and their connections. This basically amounts to reducing $C_{total\_switched}$ in equation (4) given in Section 1.3.2. The capacitive load can be reduced either statically at design-time or dynamically at run-time.

To reduce the capacitive load at design time, geometries for some devices can be reduced to lower their capacitances. Some devices still must have large geometries so they can source or sink enough current to ensure fast switching speed. The challenge is finding those devices on the chip whose geometries can be reduced without sacrificing too much performance. A power-delay product curve can be used in such a case to trade off as little performance as possible for significant power savings [1]. Cell layout libraries that are used repeatedly on the chip can be targeted for this approach.

Circuit design algorithms can be used at design time to minimize the switched capacitance. During the synthesis phase in a computer-aided design (CAD) tool, these algorithms can optimize logic gate connections to have minimum switching activity. Unfortunately, only 10% of total processor power is from synthesized logic that can be optimized by CAD tools [1].

To reduce the capacitive load during run-time, empty units or segments of units can be disabled to avoid wasteful switching activities. Extra control logic can detect when units or segments become empty and disable clocking or signal propagation. One challenge for this approach is finding the optimum granular size for units or segments that can be disabled. If the granular size is too large, a unit would rarely be disabled because some part of it, even if it is a small one, would almost always be used. If the granular size is too small, the power overhead

from many logic control gates would offset the power savings. In addition, at a small granular size, the large number of control gates can create clock skew or glitches that can be difficult to mitigate [1].

The issue queue in the out-of-order engine, for example, can be segmented, and control logic can then disable segments whenever they become empty. Because the issue queue is a "power-hungry" component on the chip, this segmentation is an effective power saving scheme. Ponomarev et al. proposed such a scheme by segmenting the bit-lines of the issue queue and using a row decoder to activate only those segments that contain an accessed instructions [16].

From a power efficiency perspective, asynchronous circuits are the most efficient. Without a global clock, asynchronous units process data only when it is available, and thus circuit activity is never wasteful. However, the challenge for asynchronous circuits has been their slow performance. Novel techniques to improve their performance, such as Interlocked Pipelines CMOS [32], have been proposed. But even with these improvements, asynchronous circuits' performance still lags behind synchronous circuits.

2.1.3 Microarchitecture Level

At the microarchitecture level, the functional organization between and within processor units can be designed to maximize efficiency with no or little impact on performance. The design for efficiency can be either static or adaptive.

Static Design:

A static design is where the microarchitecture parameters are fixed at design time so that both power efficiency and performance are optimized. The design can be guided with an optimization metric that includes both performance and power. One such metric is the energy-delay product. For a processor, the delay can be the cycle per instruction (CPI) and the energy can be the power consumption in Watts. Other metrics derived from the energy-delay product can emphasize more heavily either on the delay or the energy aspect. For example, the $CPI^3 \times Power$ metric emphasizes optimizing the delay aspect over power because the CPI is raised to the cube power.

In a static design, the energy-delay product metric for superscalar processor can be used to determine the tradeoff between power consumption and performance as cache and core sizes are varied [6]. An optimal size can then be chosen that minimizes power consumption with an acceptable performance level. The energy delay product is also used to show how simultaneous multi-threading and chip multi-processor designs can scale up better in terms of throughput and power consumption than a single wide-issue processor [6].

Adaptive Design:

In an adaptive microarchitecture design, the processor's functional organization is made to adapt to certain conditions at run-time to reduce power consumption. Many of the functional components in a superscalar processor are designed to provide high performance in the long run. However, as a thread executes, its code and data behavior change, resulting in varying demands on these components. If one or more components become a performance bottleneck, other components can become temporarily underutilized, thus wasting energy. The goal of adaptive design is to take advantage of such situations and disable unused components or portions of them to save power. The design also has to detect when the thread behavior changes again and disabled components are needed back in service. The disabled components should then be quickly re-enabled for optimal performance.

The challenge for adaptive designs is finding indicators of behavioral change in executing threads to trigger adaptation. The choice of indicator also depends on the targeted component. For example, Balasubramonian et al. propose using a combination of cache miss rate, instruction commit rate, and branch frequency as an indicator to trigger reconfiguration of the cache hierarchy [33]. By dynamically reallocating a different amount of cache memory to different levels, they claim to improve cache performance and reduce cache power consumption.

This thesis proposes microarchitecture techniques to reduce power consumption in the out-of-order engine's components (issue queue, reorder buffer, load-store queue, and connecting buses). It uses two indicators to reduce instruction occupancy in the out-of-order engine. The data dependence index (DDI) is one indicator used to downsize the out-of-order

engine. The sustained dispatch rate (SDR) is a second indicator used to reduce instruction traffic bursts into the out-of-order engine. Both indicators work in tandem to adapt the out-of-order engine to behaviors that are microarchitecture dependent and independent—the data dependence index being microarchitecture independent, and the sustained dispatch rate being microarchitecture dependent. This tandem approach results in a very effective reduction in instruction occupancy, which leads to more disabled segments and greater reductions in power consumption. A brief review of other adaptive designs for the out-of-order engine was given in section 1.6.1. A more detailed review of adaptive designs for the out-of-order engine is given in the rest of this chapter.

Temperature Aware Design:

Techniques for reducing processor power consumption reduce power either on average or in the long run. They may not reduce transient spikes in power consumption that result in hotspots on the chip. These hotspots tend to vary temporally and spatially, causing steep temperature gradients that can compromise chip reliability. Microarchitecture techniques can be used to moderate occurrences of chip hotspots using workload and instruction-level-parallelism indicators [2, 14].

2.1.4 System Level

Power savings can be achieved when considering the processor and its interaction with memory, I/O devices, and peripherals. While waiting for memory to retrieve data, idle processor time can trigger transitions into one of many possible low power modes. Memory and I/O devices can transition into low power modes as well when not in use. A system power management control is responsible for detecting idle components and triggering their transition into, or out of, lower power states. Dynamic voltage and frequency scaling (DVFS) techniques can be used to transition the processor into low power modes [34]. The Advanced Configuration & Power Interface (ACPI) is a good example of a standard system power management that has been widely adopted by the industry [35].

The operating system can also be part of a system power management. For example, the OS can help in detecting process or user activity and force a system to go into a low power mode.

2.1.5 Compiler Level

A compiler can assign different power levels to a code section based on the section's behavior. A compiler technique can be either static or dynamic. A static technique uses offline profiling analysis to determine code behavior. Dynamic compilation techniques can capture program behavior dynamically by inserting an extra execution layer between the application binaries and lower OS and hardware levels. Dynamic techniques have the advantage of capturing program behavior related to the run-time environment, such as instruction and data cache performance and branch prediction that would not be possible with static techniques. Dynamic techniques, however, have the disadvantage of potentially slowing execution if code analysis is not fast or simple enough. Both static and dynamic techniques can trigger processor low power modes through dynamic voltage and frequency scaling (DVFS). Wu et al., for example, propose to reduce processor power consumption by using DVFS and dynamic compilation to detect idle CPU cycles during memory operations [34].

2.2 Complexity of the Out-of-Order Issue

In order to improve a single thread performance, microarchitecture techniques try to exploit instruction-level parallelism (ILP) to reduce execution time. The out-of-order engine helps exploit ILP by allowing instructions to bypass each other in the pipeline. If an executing instruction stalls, other independent instruction can bypass it, keeping the instruction pipeline flowing. Unfortunately, this ILP exploitation is not linear in relation to the out-of-order engine's resources. Exploiting small ILP requires a much larger increase in the number of entries in the engine [36, 37].

This non-linearity between ILP and number of entries can be compounded with high circuit complexity. As will be shown later in this section, the out-of-order issue does indeed have high circuit complexity, which impacts not only performance but also power

consumption. Out-of-order engine components tend to be among the highest power consumers on chip [15, 16, 38]. They also tend to suffer the most intense hotspots [2, 14].

This section briefly reviews the function of different out-of-order engine components within the pipeline. It then qualitatively analyzes circuit complexity of the out-of-order issue, which is critical in terms of power consumption (for a quantitative analysis of the out-of-issue complexity, refer to [39]). This analysis provides a context for reviewing previously reported techniques (presented in later sections) that reduce power in some of the engine's components. Similarly, the analysis provides a context for the adaptive techniques presented in this thesis, which reduce power consumption in all of the engine's components.

2.2.1 Out-of-Order Pipeline Flow

To analyze the circuit complexity of the out-of-order issue, a brief review of the logic function of its different components in the pipeline is given first.

Fetching, Decoding, and Renaming Stages:

In each cycle, the pipeline starts by fetching a group of instructions from the instruction cache and then decoding them. A standard pipeline is depicted in **Figure 7**. For a branch instruction,

Figure 7. Simplified processor pipeline and stages.

the branch predictor determines whether the branch is taken and what the target address is.

The compiler usually has access to only logic registers made available by the instruction set. This limitation introduces dependences among the destination registers (or write-after-write dependence) in a thread. A particular microarchitecture alleviates these dependences by offering many more physical registers, allowing multiple instructions with the same destination register to proceed in the pipeline. The rename stage, following fetch and decode, maps logic destination registers to available physical registers. Source register operands are also mapped and their status is flagged in one of two ways. If the source register already contains the desired result, then the source operand status is flagged as ready. Otherwise, the status is flagged not

ready, as the desired result will be produced by an instruction yet to be executed ("in flight instruction"). Renamed instructions are then dispatched in program order into the issue queue and the reorder buffer.

Out-of-Order Issue, Execute, and Retire Stages:

Instructions that are dependent on results from producer instruction(s) still in flight are called consumer instructions. Since the relation between a producer and consumer instruction is not determined a priori, the result tag after execution of an in-flight instruction is broadcast to all instructions in the issue queue. Every instruction in the issue queue then tries to match the result tag against its source operand tags. If matched, that source operand is marked as ready. When both operands for a consumer instruction are ready, and the function unit needed for its execution is available, the select logic can issue this instruction to execute out of program order. When an instruction is issued for execution, its entry in the issue queue is freed.

The broadcasting of result tags, matching them against operand tags, and marking of operand tags as ready is collectively referred to as the wakeup logic. The bypass logic can also pass result values directly from a producer instruction executed in the last cycle, so that a producer and a consumer instruction can execute in back-to-back cycles, avoiding pipeline bubbles. Sometime after an instruction executes, the reorder buffer commits it in program order and frees its entry, marking the end of its pipeline journey.

2.2.2 Circuit Complexity

To implement the result tag matching against all source operands, the conventional wakeup logic implements associative matching using content-addressable memory (CAM). **Figure 7** shows a block diagram for the CAM in the issue queue. When an instruction is selected for execution, its result register tag is driven on the tag lines, so that the tag is available to all instructions in the queue. Every instruction then compares the tags of its own source operands against the result tag. If matched, the source operand status is changed to ready. Once both source operands of an instruction are ready, and the needed execution unit for that instruction is available, the select logic can issue that instruction for execution.

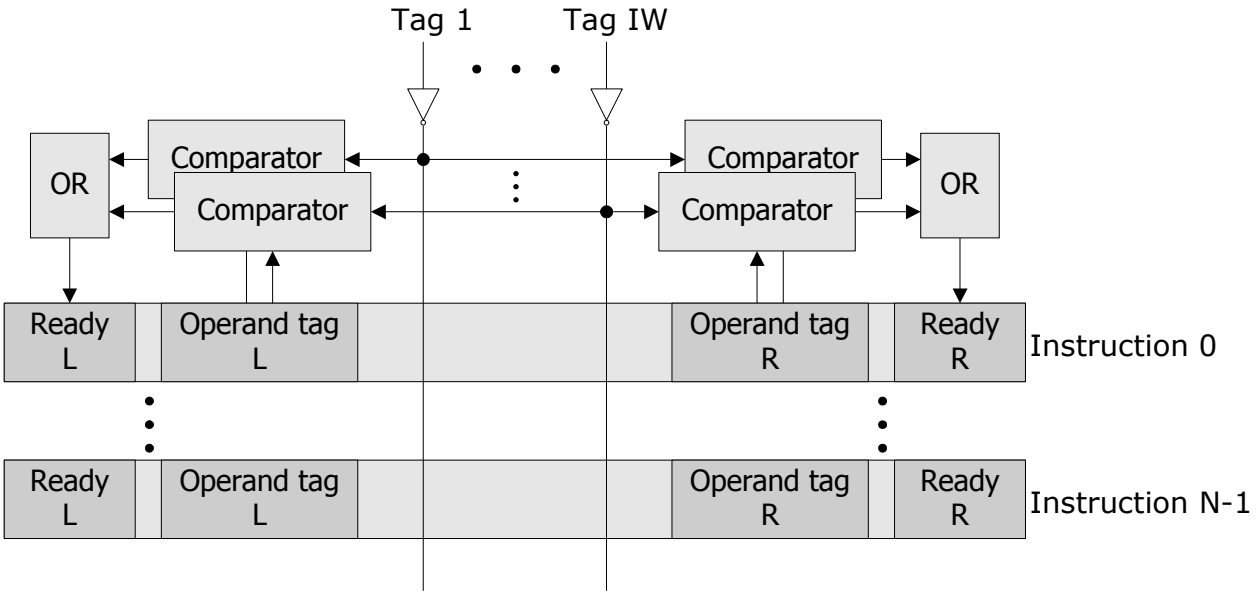


Figure 8. CAM array for the wakeup logic in the issue queue [39].

In **Figure 9**, a detailed CAM cell for one bit position of the wakeup logic is illustrated [39]. The cell compares one bit of the operand tag (DATA) with the corresponding bit position of all result tags (TAG_1–TAG_{IW}). The operand tag bit is stored in a RAM cell. Other bit positions are placed on the same row connected to the same match line. In every cycle, the horizontal match lines are pre-charged high. The pull-down stack would pull a match line low if any of the operand bits are mismatched against the result tag. Otherwise, the match line stays high, indicating all operand bits match all bits of one result tag. The horizontal match-line and its associated pull-down stack correspond to one comparator in **Figure 8**. If an operand tag is matched against any of the result tags, then the OR logic indicates the ready status for that operand.

To exploit more ILP, both the issue width (IW) and the number of entries in the issue queue need to increase. The capacitive load in the CAM array then increases superlinearly due to the following:

1. Increasing IW implies increasing the number of tag lines, which in turn results in increasing the width of the CAM array horizontally; that is, the match lines length has to increase.

2. Increasing the number of instruction entries implies increasing the height of the CAM array vertically; that is, the tag lines length has to increase.

3. The number of devices and capacitive loading also increases with increasing the CAM array tag lines and match line.

The superlinear increase in the capacitive load in the associative CAM array leads to superlinear increase of power consumption in the out-of-order issue [39].

The reorder buffer and the load-store queues are usually implemented as RAM array, rather than CAM array. However, to exploit more ILP, the capacitive loading in these units also increases superlinearly due to the following:

1. With increasing IW, the RAM arrays have to support multiple read and write ports. This leads to increased array size horizontally.

2. Increasing the number of entries leads to increased array size vertically, leading to increased bit-line length.

3. Just as with CAM array, the number of devices and their capacitive loading also increases.

With increasing the number of entries and the issue width, the result-bus, which connects all components together in the out-of-order engine, has to support more ports for simultaneous reading and writing instructions. This leads to increased capacitive loading. The result-bus would also have higher a circuit activity.

As a result, the superlinear increase in the capacitive loading in all out-of-order engine components, coupled with higher circuit activity in all of its components, leads to superlinear increase in its power consumption.

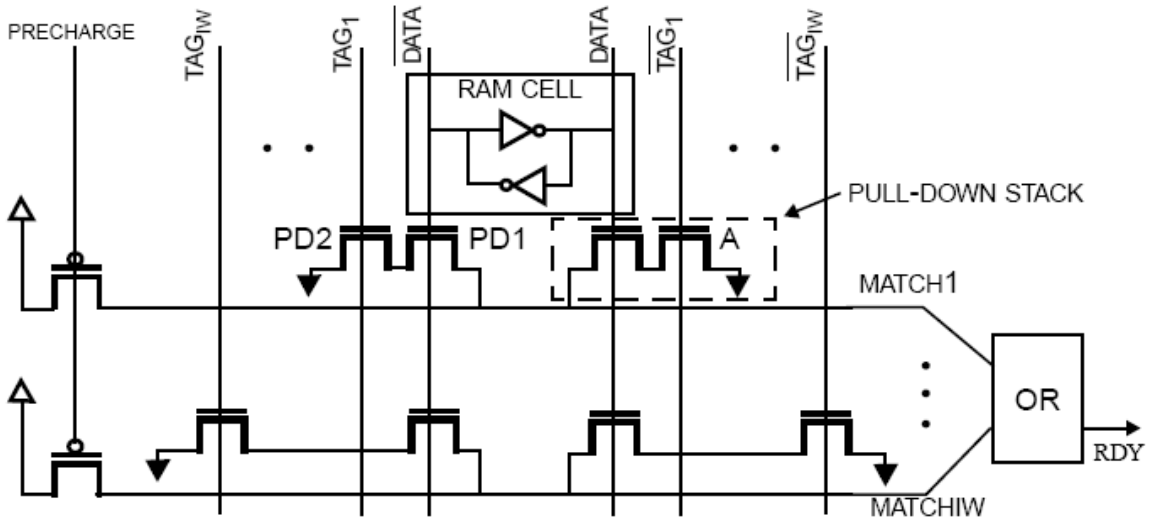


Figure 9. Circuit diagram of a CAM cell for one bit position of the wakeup logic [39].

2.3 Power Reduction for the Out-of-Order Engine

2.3.1 Classification

To reduce the high power consumption of the out-of-order engine, techniques with different approaches have been proposed. **Figure 10** shows a graphical classification of these techniques based on their approach. Techniques that keep a fixed size for components rely on dynamic reconfiguration or circuit techniques to reduce power consumption. Such techniques are outside the scope of this thesis and are briefly reviewed in this section.

Downsizing techniques can dynamically reduce the effective size of components to reduce power consumption. The trigger conditions for resizing in these techniques vary, but the variations share some common patterns. This section identifies and classifies these patterns in order to help address any limitations they might have in common. This also helps in the design of a new technique that combines the strength of different class patterns.

A common pattern for most downsizing techniques is to segment one or more components in the out-of-order engine and disable empty segments for power savings. The

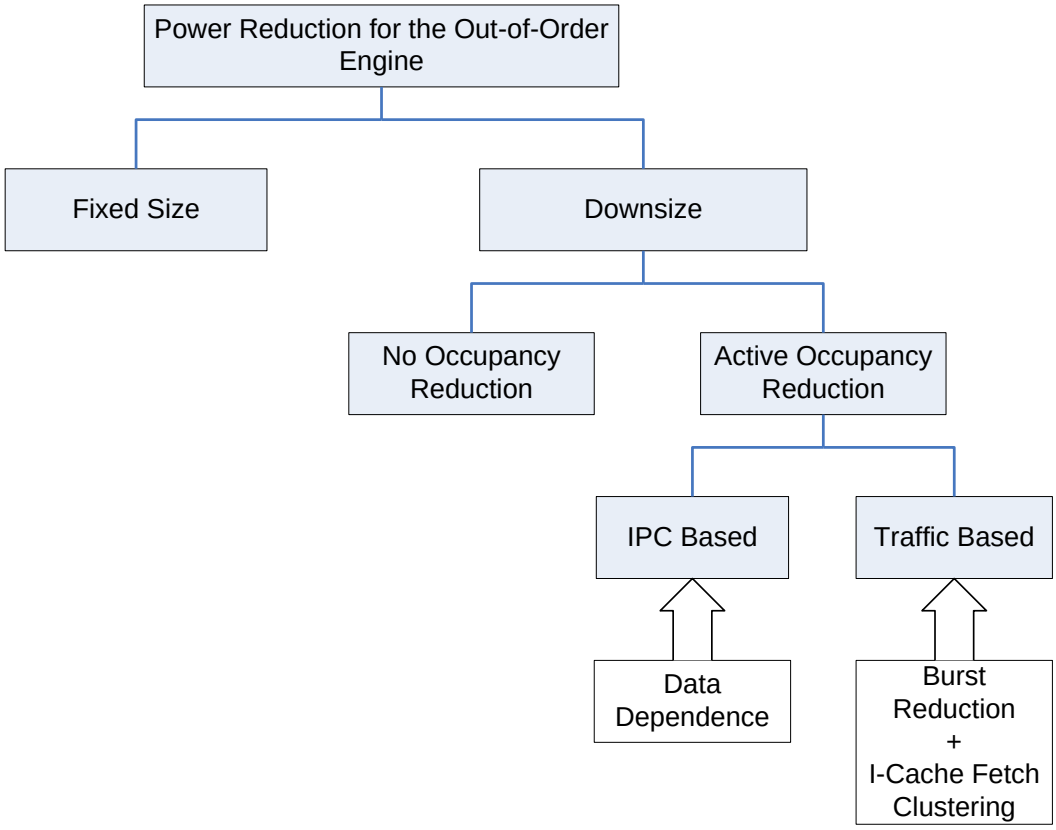


Figure 10. Classification of power reduction techniques for the out-of-order engine. The boxes on bottom right represent techniques proposed by thesis to replace the IPC based and traffic based techniques.

segmentation is implemented at the circuit level, by disabling signal propagations to a segment to reduce activity and the total capacitive load.

Beyond common downsizing, the techniques differ on whether a limit on the number of active segments is set. Shutting down empty segments that are normally empty, with no additional attempt to reduce instruction occupancy, is classified here as having no occupancy reduction. In contrast, microarchitecture techniques that cause components to reduce instruction occupancy based on dynamic thread behavior are classified here as *active* occupancy reduction. This difference is graphically illustrated in **Figure 11**.

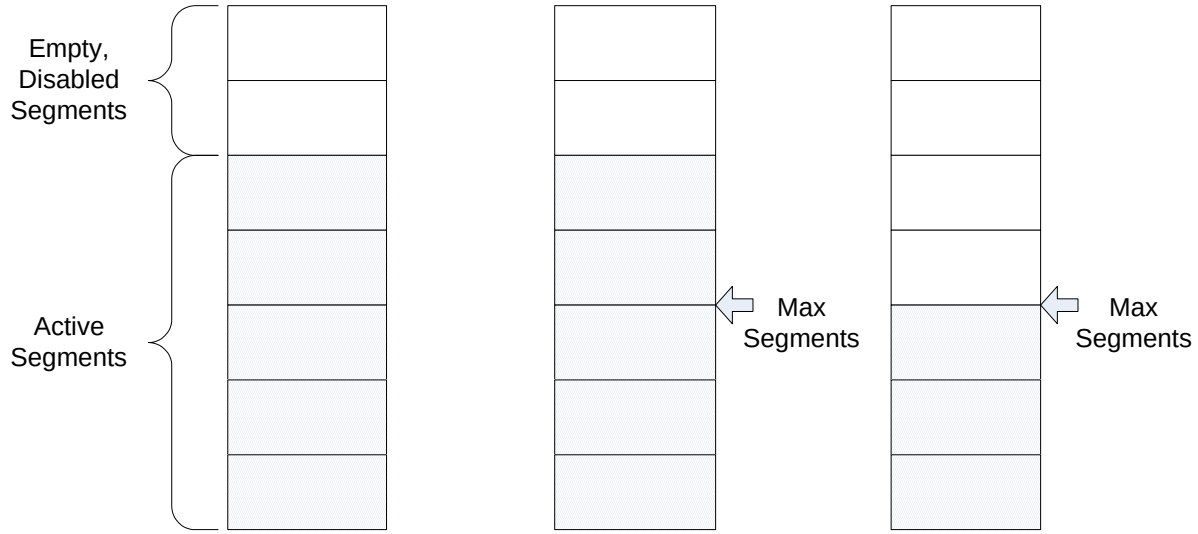


Figure 11. Active versus no occupancy reduction for an abstract queue. Without actively limiting the number of active segments, only a few segments can be disabled (left). When a limit is set based on thread behavior (middle), segments are forced to drain, creating more empty segments that can be disabled (right).

Active occupancy reduction techniques that have been proposed thus far downsize their components based on monitoring the instruction commit rate per-cycle (IPC), or based on reducing instruction traffic. Active occupancy reduction offers more attractive power savings than neutral occupancy reduction. The advantage of actively adjusting processor resources based on thread behavior was also discussed in [17]. However, current approaches for active occupancy reduction have yet to fully realize that potential. It is worth noting that all proposed active occupancy reduction techniques downsize only the issue queue. The only technique that downsizes all out-of-order engine components does not reduce occupancy. The following sections will review proposed neutral and active occupancy reduction techniques in details.

2.3.2 Downsizing with Neutral Occupancy Reduction

Only one technique proposed by Ponomarev et al. used neutral occupancy reduction, but it is also the only one that downsized all major components in the out-of-order engine (issue queue, reorder buffer, and load-store queue) [15]. In this technique, empty segments are disabled for power savings, but they are reactivated as soon as the front-end exerts pressure to dispatch more instructions. The reactivation of empty segments takes place regardless of whether they help performance or not.

Power savings for Ponomarev et al.'s technique were reported for the out-of-order engine components. However, total processor power savings were not reported due to a limited use of power modeling in their methodology. Moreover, clustering of fetched instruction to reduce the i-cache power consumption was not addressed [15].

2.3.3 Downsizing with Active Occupancy Reduction

A number of the proposed techniques to reduce power in the out-of-order engine can be classified as active occupancy reduction. They either implement IPC monitoring algorithms to trigger resizing or they limit instruction traffic based on microarchitecture indicators. However, none of the active occupancy reduction techniques proposed so far have targeted all components of the out-of-order engine for power reduction.

Using IPC:

Downsizing the issue queue based on an IPC metric was proposed by Folegnani et al. [38]. In this technique, the issue queue is segmented and the top segment is monitored for its contribution to the IPC. If the segment does not contribute, then it is disabled to save power. The monitoring then moves on to the next lower segment. Since the only way to determine whether a segment contributes to performance is by enabling it and then disabling it, the algorithm can spend a substantial time upsizing the issue queue. With often short-lived execution behaviors, this periodic upsizing can miss power savings opportunities and may not react fast enough to thread behavioral changes [20]. This is a typical adverse side effect of downsizing based on the IPC.

In Folegnani et al.'s technique, only power savings for the issue queue are reported; occupancy reduction is not reported. Moreover, a large portion of the power savings comes from disabling the wakeup logic of entries that either are empty or have ready operands; not from downsizing per se. Total processor power savings for their technique are not reported due to limitations of power modeling.

Another technique that monitors the IPC to actively reduce instruction occupancy was proposed by Karkhanis et al. [19]. In this technique, an algorithm measures the IPC over an

initial interval while setting the number of instructions in the pipeline stages from fetch through issue to the maximum possible. Over subsequent intervals, the algorithm lowers the number of instructions to different maximums. If a lower maximum is found to yield the same IPC as the initial interval, then this new maximum is maintained. Otherwise, the algorithm reverts back to the initial interval state. This process is repeated anytime the IPC or branch behavior changes.

This technique suffers from the same IPC limitation as Folegnani et al.'s. The algorithm has to go back to the maximum possible state and then "hunt" for a lower instruction limit. The duration of the initial interval as well all subsequent intervals are on the order of 100K cycles. This means that to find an optimal state, the algorithm might need on the order of a million cycles. With thread behaviors acting on a much shorter scale [20], the processor can be constantly in a "hunt" or inefficient state. Moreover, changes in IPC or branch behavior can trigger this "hunting" process, even if the original state is optimally efficient.

Segmentation of out-of-order engine components and disabling empty segments were not implemented in Karkhanis et al.'s technique. Thus, high power consumption by the tag line was not reduced, and only power savings for the issue queue comparators are reported. Reduction in instruction activities is reported, but reduction in instruction occupancy is not reported. Total processor power savings were not reported due to limited power modeling.

Using Traffic Reduction:

A traffic reduction technique based on branch behavior was reported by Manne et al. [21]. In this technique, the front-end stops fetching instructions when it encounters a branch that it cannot predict confidently. Instructions from an often-unpredicted path are then less likely to enter the pipeline, thereby reducing total traffic.

However, reducing instruction traffic alone will not always reduce occupancy. A queue's occupancy might be reduced when traffic going into it is reduced, but if the queue's service rate is also reduced, the queue steady state remains at high occupancy. Moreover, reducing traffic based on a single behavior, such as branch prediction, does not take advantage of other thread behavioral changes.

Further reduction in traffic over Manne et al. was reported by Baniasadi and Moshovos [40]. Data dependences and flow rate were used to control traffic. However, this technique shares the same traffic reduction limitation as Mann et al. Moreover, their data dependence metric only counts data dependences. This simple count does not consider how well instructions line up in a dependence chain.

Another traffic reduction technique proposed by Buyuktosunoglu et al. uses issue queue utilization and a parallelism indicator to pause instruction fetch [41]. When utilization is high and parallelism is low, fetching is paused to lower utilization. When utilization becomes low, fetching is resumed. This dynamic, however, causes the utilization to fluctuate between high and low levels during low parallelism periods, instead of remaining low. Only issue utilization is addressed in the out-of-order engine. Moreover, the fetch pausing is not timed with the fetch buffer occupancy to cluster i-cache instructions more effectively.

Need for a New Tandem Approach:

Because the IPC is only a symptom of performance that is determined by many behaviors, downsizing the out-of-order issue queue based on the IPC can affect the instruction window's ability to find and issue instructions out of order. To avoid this effect, IPC techniques have to employ periodic upsizing, which renders them too slow to react to short-lived thread behaviors [20]. Moreover, a traffic reduction technique alone will not always reduce occupancy, especially when the out-of-order engine's service rate slows down.

What is needed is a technique that resizes the out-of-order engine but does not depend on the IPC. The out-of-order engine should be resized whenever it becomes less effective in issuing instructions out of program order. This technique should be coupled with a technique that reduces traffic bursts, especially when the out-of-order engine service rate is slow. Reducing traffic bursts above the sustained dispatch rate lessens the chances of congesting the out-of-order engine.

This thesis develops both techniques. It develops a new data dependence index to resize the out-of-order engine, and it develops a burst reduction technique by using the fetch buffer to

moderate traffic. The latter technique also achieves clustering more instructions in a single i-cache access, thus reducing the i-cache power consumption.

2.3.4 Fixed-Size Techniques

These techniques do not downsize components. However, they might be applied in conjunction with microarchitecture techniques to reduce occupancy for more power savings. For example, dynamic reconfiguration of the issue queue was proposed by Bai and Bahar based on performance counters [42]. Reducing tag comparisons in the issue queue was proposed by Huang et al. [43]. A pointer-based issue queue was proposed by Ramirez at al. [44].

2.4 Complexity Reduction of the Out-of-Order Issue

The techniques discussed in this section are designed to reduce the complexity of the associative wakeup logic in the issue queue to improve performance. None of these techniques address power consumption. It is theoretically possible to combine active occupancy techniques with complexity reduction to improve both performance and efficiency. Such an approach is beyond the scope of this work and will be discussed in the future research section. These techniques are reviewed here for completeness. They are grouped into three categories: associative techniques in which the associative logic is still used, non-associative techniques in which other circuit forms are used (such as indexing and arrays), and scheduling techniques in which instruction latencies are used to reduce complexity.

2.4.1 Associative Techniques

Associative techniques try to reduce issue queue complexity by using a smaller out-of-order issue queue, and complement it with a larger simple queue or buffer. Deciding which instructions go to the small, complex queue and which ones go to the large, simple queue is critical for performance.

Lebeck et al. complement a small conventional issue queue with a large 2K-entry, simple, waiting buffer [45]. If a load instruction misses in the cache, then it is moved to the waiting buffer along with all the instructions that depend on it. The load and its dependent

instructions stay in the wait-buffer until the cache miss is serviced, as **Figure 12** (a) shows. The authors demonstrate that such a design can achieve significant speedup over a conventional, monolithic issue queue; however, they do not estimate the impact on power consumption.

Some high-performance processors allow instructions that are dependent on a load to issue speculatively before knowing whether the load will actually hit in the data cache or not. Based on decisions from a load-hit predictor, load-dependent instructions are speculatively issued early to save some pipeline cycles. When the predictor is wrong and the load does not hit in the cache, all dependent instructions that were issued speculatively must be reissued after the cache miss has been serviced. After having been issued, the speculative instructions must stay in the issue queue until their status is resolved, thereby occupying more entries and requiring a larger issue queue size.

For the purpose of having a small issue queue capable of high clock speed, Moreshet and Bahar propose using a replay buffer to store load-dependent instructions after they are issued speculatively from the main issue queue [46]. If the load hits in the cache, these instructions are removed from the replay buffer; if the load misses in the cache, then the instructions are reissued from the replay buffer when the cache miss is serviced, as **Figure 12** (b)

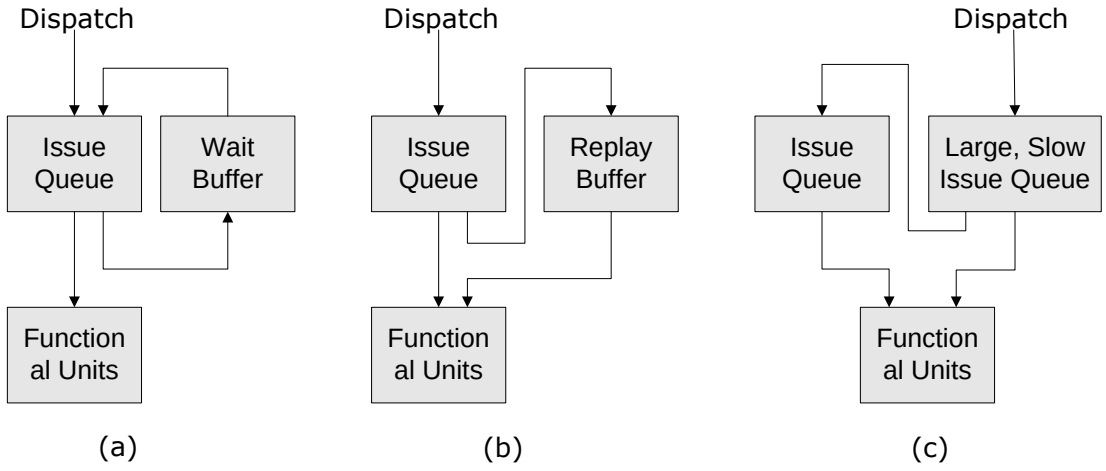


Figure 12. (a) Using CAM queue with a waiting buffer for instructions dependent on a load that missed in the cache; (b) Replay buffer for speculative loads; (c) Moving critical or oldest instructions to an associative queue.

shows. The replay buffer can be implemented with simpler, slower circuitry since loads that miss in the cache take multiple cycles to be serviced. The authors claim 18% power savings for the issue queue with no degradation of performance [46].

Brekelbaum et al. propose a simple, slow, and large issue queue to which all instructions are dispatched [47]. As non-ready instructions age and become older, they become more "critical". In every cycle, the oldest critical instructions are moved to a fast, smaller queue, from where they are issued, as **Figure 12** (c) illustrates. Since the large queue is outside the critical timing path, this hierarchical scheme yields a larger instruction window capable of extracting more ILP, but with the benefit of a fast, low-complexity issue queue. Beyond performance benefits reported in this study, no detailed power consumption tradeoffs were reported.

Ernst and Austin leverage the fact that most instructions have only one non-ready operand [48]. They propose using three queues: one non-CAM queue for instructions that are ready at dispatch, a second CAM-based queue for instructions having only one ready operand, and a third small CAM-based queue for instructions having none of their operands ready. Since the two CAM-based queues are smaller than a conventional large CAM queue, the authors show that such arrangement consumes 10-20% less power while running at clock speed 20-40% faster than the conventional design [48]. However, they do not report any power savings.

2.4.2 Non-Associative Techniques

Indexing:

Canal and Gonzalez offer indexing as an alternative to associative CAM design [49]. This technique uses RAM and an index of register tags to send each result to only one consumer instruction. Cruz et al. observed that, for a majority of instructions, a result produced by one instruction is used by only by one other instruction [50]. Based on this observation, Canal and Gonzalez proposed a queue for instructions that are the first user of a result produced by another instruction. This queue is called the first-use instruction queue, as **Figure 13** shows. Other instructions that are not "first-use" are dispatched to a small, conventional, associative queue. Instructions that have all operands ready are dispatched directly into an in-order queue.

The reduced complexity of the combined three queues allows them to be clocked faster than a conventional associative issue queue. The authors do not, however, estimate power savings for this technique.

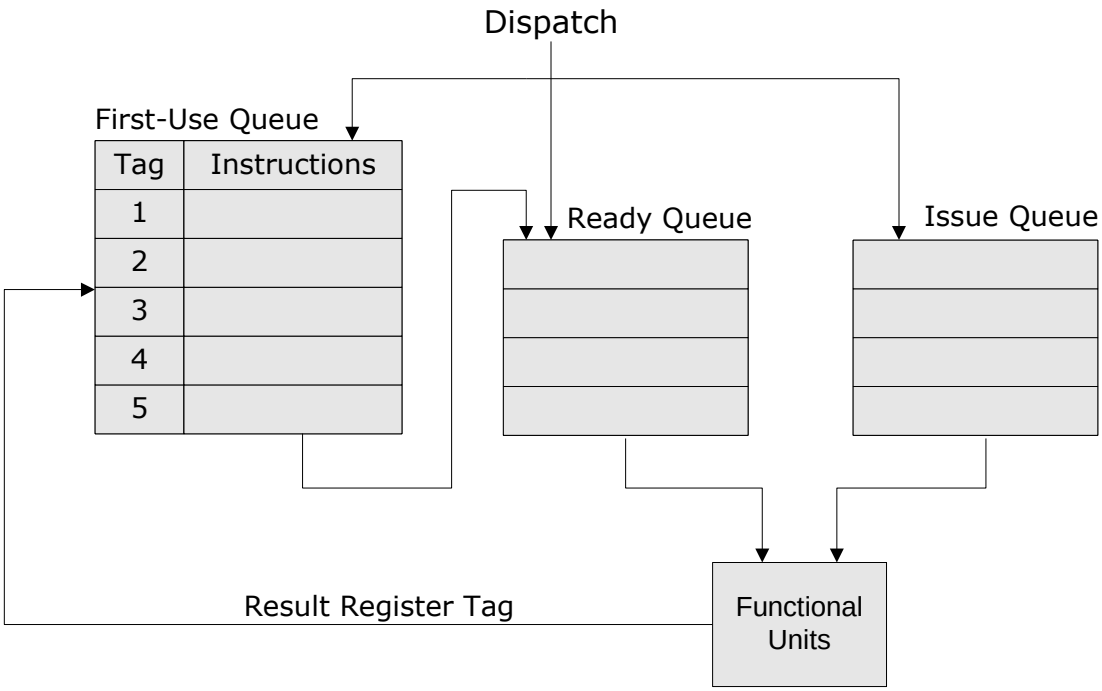


Figure 13. Block diagram for the first-use indexing [50].

Array:

Another alternative to a conventional issue queue based on associative search is to use an array structure that keeps track of instruction dependences. The array consists of rows representing instructions, and columns representing registers, as shown in **Figure 14**. A cell in the array is set to "1" when an instruction depends on a physical register whose value will be written by another instruction that is still in "flight". When a register is written by an instruction, the wakeup logic resets all cells in that register's column to "0". An instruction is determined as ready when all of its row cells are reset to "0".

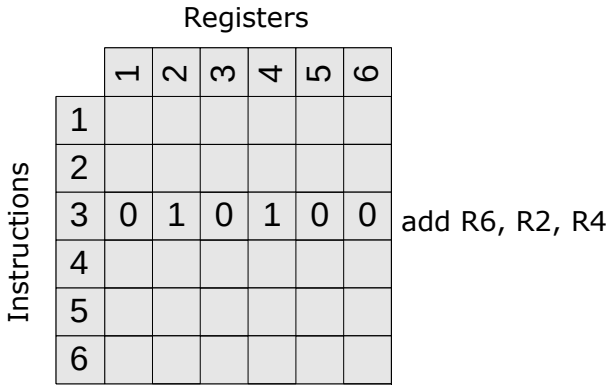


Figure 14. Array representation. The add instruction stored in row 3 depends on both registers 2 and 4.

Compared to the CAM methods, the array technique reduces complexity by eliminating the associative comparisons of register tags. However, it still has the disadvantage of requiring a large number of decoders for selecting and clearing a register column. Determining when an instruction is ready also requires a large number of comparisons to check all cells of a row.

Goshima et al. improved on the basic array method by distributing the array into three arrays for storing different types of instructions: integer, floating-point, and load-store [51]. They report improvement in clock speed due to reduced cycle time of the wakeup and select logic, but they do not study potential power reduction benefits.

Deeper pipelining has been used consistently by microprocessor designers to increase performance, as deeper pipelines allow higher exploitation of parallelism. Increasing pipeline depth also allows faster clocking, as there is smaller logic computation per stage. Over the past 20 years, for example, pipeline depths have increased from 1 for the Intel 286 processor, to 5 for the Intel 486, to 10 for the Intel Pentium Pro, to 40 (based on fast clock) for the Intel Pentium 4 [10].

Critical loops, however, pose a challenge for deeper pipelining. A critical loop is a sequence of logic operations that must produce results in one pipeline stage, so the results can be used in the next cycle; otherwise, a pipeline "bubble" is propagated, reducing IPC performance.

The wakeup and select logic form such a critical loop. When an instruction is selected for execution, its dependent instructions have to wake up in the same cycle so that they can execute in the next cycle; otherwise, the instruction and its dependent instructions cannot execute in consecutive cycles; i.e., a pipeline bubble would be produced. To allow pipelining the wakeup and select logic, Brown et al. have proposed breaking down the critical loop of wakeup and select logic into two loops: a critical, single-cycle loop for wakeup logic and a non-critical loop for select logic [52]. They accomplish breakdown by speculating that all waking instructions are immediately selected for execution. They use an array to keep track of instruction dependences over the two loops. The authors claim potential power savings but they do not report power savings data for their technique.

2.4.3 Scheduling Techniques

Scheduling techniques try to take advantage of complete or partial knowledge about instruction latencies to reduce the wakeup's high wire-latency in the issue queue. The challenge for these techniques is how to accurately estimate latency or issue-time.

Generally, the issue-time of an instruction is based on the issue-time of all instructions it depends on. If all load instructions hit in the cache, this estimation would be simple. The difficulty arises due to variable latency of memory access instructions.

Canal and Gonzalez have proposed two similar but different techniques to deal with the variable latency of loads [49]. In one technique, they propose using a wait-buffer to hold load instructions, and all instructions dependent on them, until it is known whether the loads hit in the cache or not. Once cache latency for a load is determined, the load and all dependent instructions are scheduled with proper priority into an in-order queue, as **Figure 15** (a) illustrates.

The second technique proposed by Canal and Gonzalez also uses a wait-buffer, but with a different arrangement [53]. All load instructions are assumed to have cache-hit latency as they enter the in-order queue. If they suffer a cache miss and can't issue when they reach the bottom of the queue, they are relocated to the wait-buffer along with dependent instructions. Once the

cache miss is serviced, instructions can then issue directly from the wait-buffer, as **Figure 15** (b) illustrates.

For both examples above, the authors assert that these techniques reduce the complexity in comparison to using only one large out-of-order issue queue and, therefore, are amenable to faster clocking and scalability. The authors, however, do not conduct a study for potential power savings benefits. Other scheduling techniques have been reported that report innovative circuit design with similar results [54-56].

From a power consumption perspective, scheduling instructions is not power efficient in two cases. One is when most instructions line up in a data dependence chain and cannot be scheduled. The second is when most instructions are independent and performance does not benefit from scheduling them. It is theoretically possible that active occupancy reduction

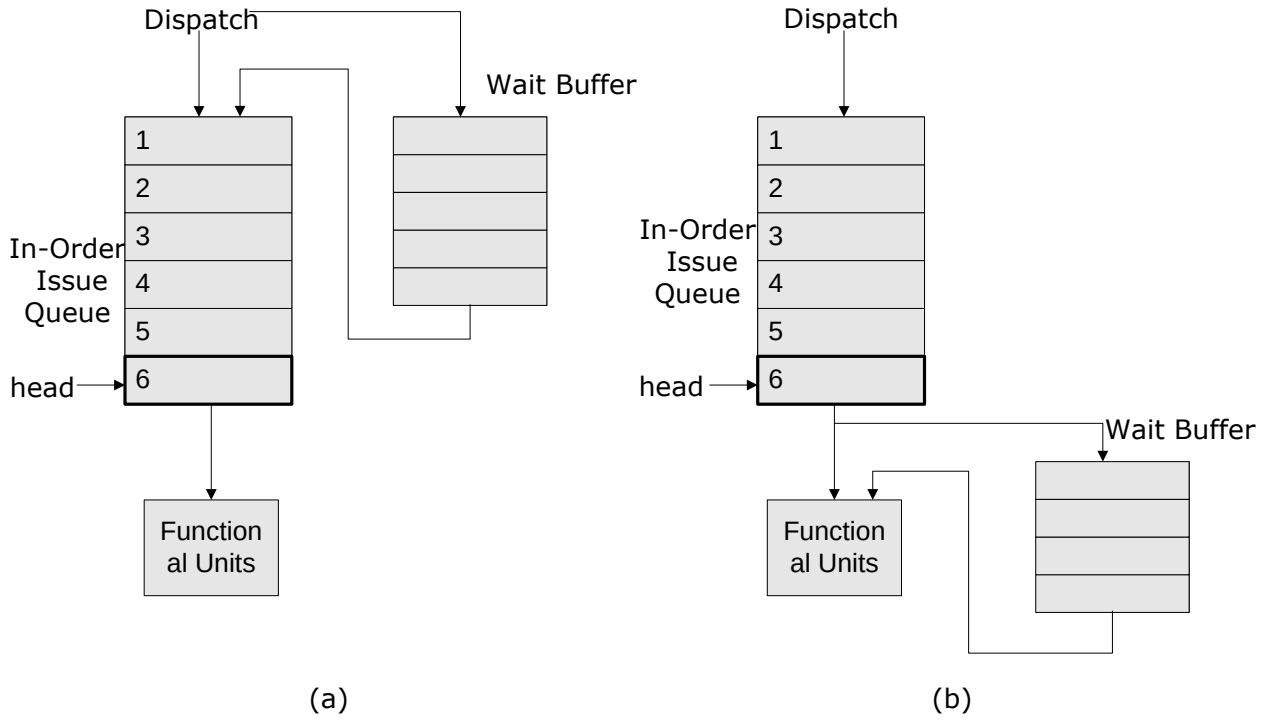


Figure 15. Using an in-order issue queue with latency scheduling. Only instructions at the head of the queue can issue. In (a), loads and their dependent instructions are dispatched to a wait-buffer, and in (b), loads are moved to a wait-buffer when they can't issue due a cache miss.

techniques developed here could be combined with scheduling techniques to achieve additional optimization in performance and power consumption.

2.5 Power Models at the Microarchitecture Level

Since power consumption has become a critical design constraint, microprocessors designers have to evaluate power consumption early on in the design process. Traditional circuit or device level power estimation tools, however, are not practical at the architectural definition stage. Traditional power estimation tools rely on circuit parameters obtained after the chip layout and floor-planning stages. It is usually too late after these stages to make architecture modifications for power reduction. Moreover, the dynamic power component is dependent on actual workloads. Architecture simulations with workloads using detailed layout circuit parameters entail time-consuming simulations that are impractical for designers to quickly explore the tradeoff space of power and performance.

Recently, researchers have developed new techniques capable of quick dynamic power estimation at the architecture level. The key feature of these techniques is the estimation of the total, switched capacitive loading variable (C_{tot}) in the processor dynamic power equation (3), in Chapter 1. Recall from that equation that dynamic power is dependent on the variables C_{tot}, V_{DD}, N_c, and t_p. All of these variables are dependent on the technology process, but the total switched capacitance, C_{tot}, and the number of cycles, N_c, are, in addition, dependent on architecture configurations. N_c can be easily obtained from an architecture, cycle-level simulator; however, C_{tot} is the key variable that needs to be estimated efficiently at the architecture level to estimate dynamic power consumption. Brooks et al. [24], Vijaykrishnan et al. [57], and Cai [58] have implemented dynamic power models that estimate C_{tot} based on architecture parameters, such as pipeline width, issue queue size, cache sizes and configuration, number of registers, and other architecture parameters.

2.5.1 The Wattch Approach

To estimate the switched capacitance efficiently, Brooks et al. [24] in their Wattch power models simplify the capacitance calculations by grouping typical processor circuits into four major circuit categories:

1. *Memory Array Structures:* used for instruction and data caches, register files, branch predictor, and buffers.

2. *Fully Associative CAM:* used for the issue queue wakeup logic, the reorder buffer and the translation look-aside buffer.

3. *Combinational Logic and Wires:* used for functional units, the issue queue selection logic and result buses.

4. *Clock Structures:* for distributing clock signals to all units on the die.

For each of the above circuit categories, Brooks et al. then derive formulas for the capacitive load of all essential circuit stages based on a typical physical layout. To illustrate how they derive the capacitance equations for the word-lines and bit-lines activation stages for the register file structure, the schematic circuit diagram in **Figure 16** is used. The register file is a memory array type. Its word-lines are laid out horizontally and its bit-lines are laid out vertically in the schematic. The word data bits are stored in static memory cells across the array. The array access has four stages: in the first stage, the array activates decoding the requested address (left in the schematic); in the second, it activates the resulting word-line; in the third, it activates the bit-lines discharge from cell transistors after the initial pre-charge; and in the fourth, it activates the output data from the sense amplifiers connected to each pair of bit-lines complements .

The capacitance of a single word-line is composed of the capacitance of the diffusion capacitance of the word-line driver (left in **Figure 16**), plus the gate capacitance of the cell access transistors, plus the capacitance of the line metal wire. Similarly, the capacitance of a single bit-line is composed of diffusion capacitance of the pre-charge driver (top in **Figure 16**), plus the

diffusion capacitance of the cell access transistors, plus the capacitance of the line metal wire. **Table 1** lists the total capacitance equations for all the word-lines and the bit-lines for a register file structure.

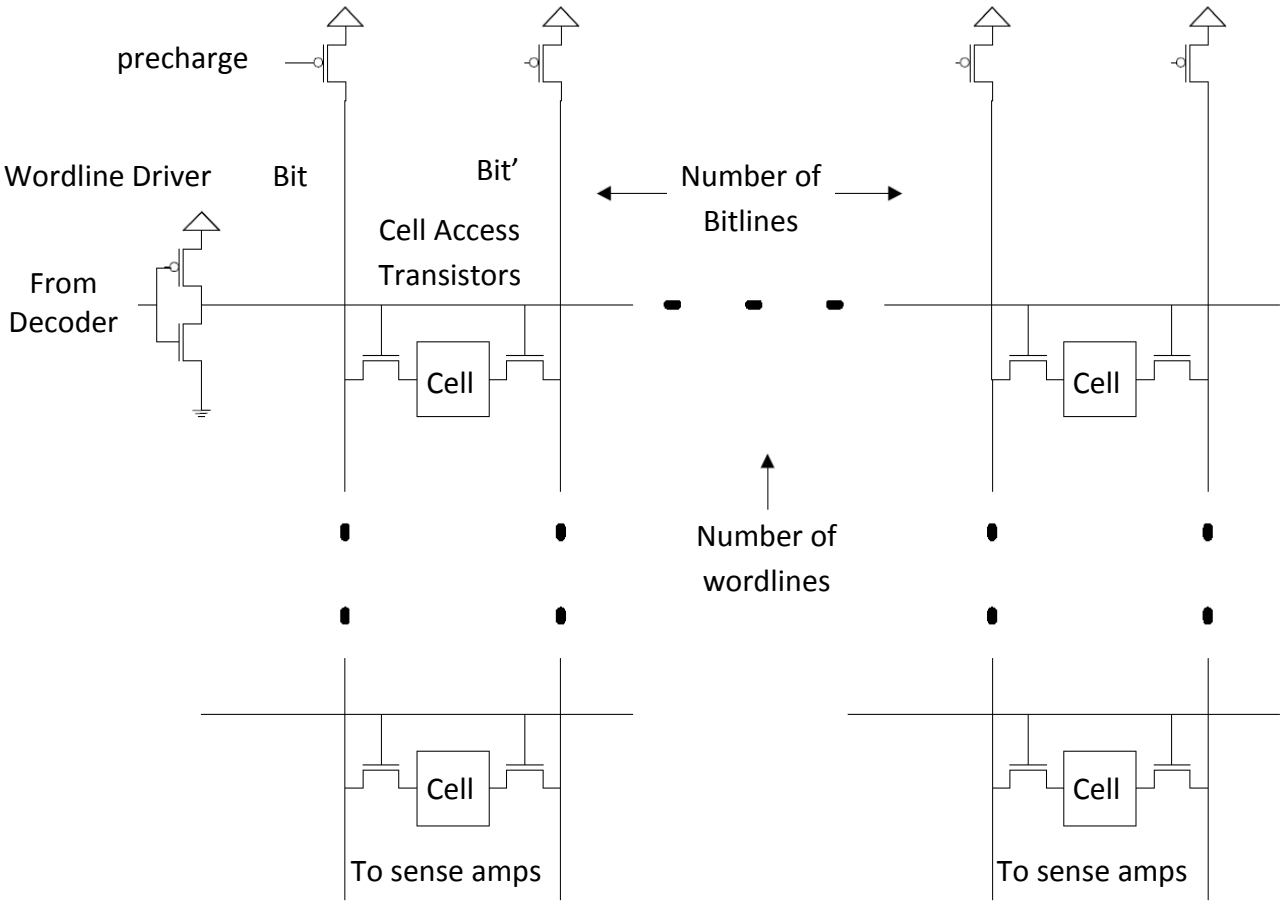


Figure 16. Schematic diagram of word-lines and bit-lines for a circuit array structure (adapted from Brooks et al. [24])

The capacitance equations in **Table 1** factor in the register file size, so they are a function of architecture parameters, which is a key feature for these equations. The number of word-lines, the number of bit-lines, the diffusion capacitance for the word-line and the pre-charge drivers, the word-line length, and the bit-line length all are a function of the register file size. The register file size is an architecture parameter dependent on the register file width, the number of entries, and the number of read-write ports. **Table 1** lists circuit types whose capacitance load is scaled based on the respective architecture structure. Complete details for

other structures' capacitance equations can be found in [24] and the Wattch source code that is available from the authors.

Table 1 Capacitive load equations used for bit-lines and word-lines stages for the circuit of type memory array for a register file structure.

Node	Capacitance Equation
Register File Wordline Capacitance =	C_{diff} (Wordline Driver) + C_{gate} (Cell Access) × Num Bitlines + C_{metal} × Wordline Length.
Register File Bitline Capacitance =	C_{diff} (Precharge) + C_{diff} (Cell Access) × Num Wordlines + C_{metal} × Bitline Length.

The Wattch power models integrate with the sim-outorder SimpleScalar cycle-level simulator, and Wattch uses the architecture parameters given to sim-outorder. When simulating a program, and before sim-outorder gets into its main cycle-level loop, the Wattch models are invoked to calculate the capacitance load for each architecture structure. The capacitance load values are then stored in memory for fast dynamic access. Each time a unit is accessed during the cycle-level loop, that unit's capacitance is added to the running sum for $C_{tot.}$. When the simulated program ends, Wattch uses C_{tot} and the total number of cycles, N_c , obtained from sim-outorder to calculate the average dynamic processor power (equation (3), Chapter 1). Wattch uses constant V_{DD} and t_p values for a given technology process.

Clock gating:

For multi-ported units, clock gating becomes a more common approach in processor designs to curb power consumption. When a multi-ported unit is accessed on some but not all ports, only the accessed ports are usually activated to reduce power consumption. Wattch authors have developed three models of clock gating for multi-ported units:

1. A unit consumes 100% of its full power whenever it is accessed on any particular port. The unit consumes zero power when none of its ports are accessed.

2. The power that a unit consumes increases linearly with the number of accessed ports. The unit consumes zero power when none of its ports are accessed.

3. The power that a unit consumes increases linearly with the number of accessed ports. The unit, however, consumes 10% of its full power when none of its ports are accessed. This option reflects the power overhead for control logic to disable part or all of a unit.

In our work, we used the above option (3), and our reported data for power savings reflect the 10% overhead.

3 METHODS

This Chapter presents the methods used to reduce instruction occupancy and ultimately increase processor efficiency. Section 3.1 states the guiding two principles for the methods. The principles were introduced in Chapter 1. They are elaborated here in more depth. Principle 1 states two cases for downsizing the out-of-order engine based on data dependences. Principle 2 states the case for reducing instruction traffic bursts. The tandem principle explains why both principles must be applied to achieve significant occupancy reduction.

Section 3.2 describes the methods used to implement Principle 1. It first develops the data dependence index (DDI) that is needed to quantify data dependences. The token passing algorithm shows how to measure the DDI in hardware. An alternative measuring algorithm to measure the DDI by extending register renaming is also proposed. Validation of the algorithms is then presented. The downsizing policy shows the rationale for when and how much to downsize the engine as a function of the DDI.

Section 3.3 describes the methods used to implement Principle 2. Three methods are investigated to evaluate the most effective one in moderating traffic bursts: The first uses the DDI as a condition to pause fetching; the second uses the sustained dispatch rate (SDR) and the fetch buffer occupancy to pause fetching; and the third uses both SDR and the issue occupancy to pause fetching. Section 3.4 then briefly shows that the second and third methods have the advantage of clustering fetched instructions, which can reduce i-cache power consumption.

Finally, Section 3.5 describes the simulation framework in detail. Both the architecture and the power models are described. The workload benchmarks and the simulations methods are then described.

3.1 Efficiency Principles for the Out-of-Order Engine

3.1.1 Principle 1

Gradually downsize the out-of-order engine as it becomes less effective in issuing instructions out of program order. This principle can be applied to queues in the out-of-order engine in two cases.

Case 1: Upper levels of data dependence. If window instructions exhibit long chains of data dependence, the data dependence level is said to be high. The more instructions line up, the higher the dependence level. When this happens, the window's performance rapidly becomes insensitive to window size. This insensitivity effect can be observed in Wall's study on available instruction level parallelism [36]. **Figure 17** shows this effect for two benchmarks using Wall's data. For an ideal processor, no parameter affects the issue rate performance other than data dependence and window size. Although the figure shows data dependence for entire benchmarks, data dependence levels vary dynamically, as will be seen later in the chapter.

This effect is also observed for an ideal processor with an unlimited issue width. However, it can still be applied for a non-ideal processor. The high data dependence levels naturally limit the issue rate, so that the limited issue width of a non-ideal processor does come into play. Thus, this trade-off between window size and performance at upper levels of data dependence can still be exploited for efficiency for a non-ideal processor. **Figure 18** graphically shows this exploitation.

Case 2: Lower levels of data dependence. This case exploits the limited issue width of a non-ideal processor at low levels of data dependence. In Case 1, it was asserted that performance is sensitive to window size at low levels of data dependence for an ideal processor. However, the limited issue-width of a non-ideal processor renders performance insensitive to window size at very low levels of data dependence. In this case, a small window would be able to find enough independent instructions to issue and saturate the issue-width, even in the event of other microarchitecture latencies, such data cache misses. This effect can then be exploited for efficiency.

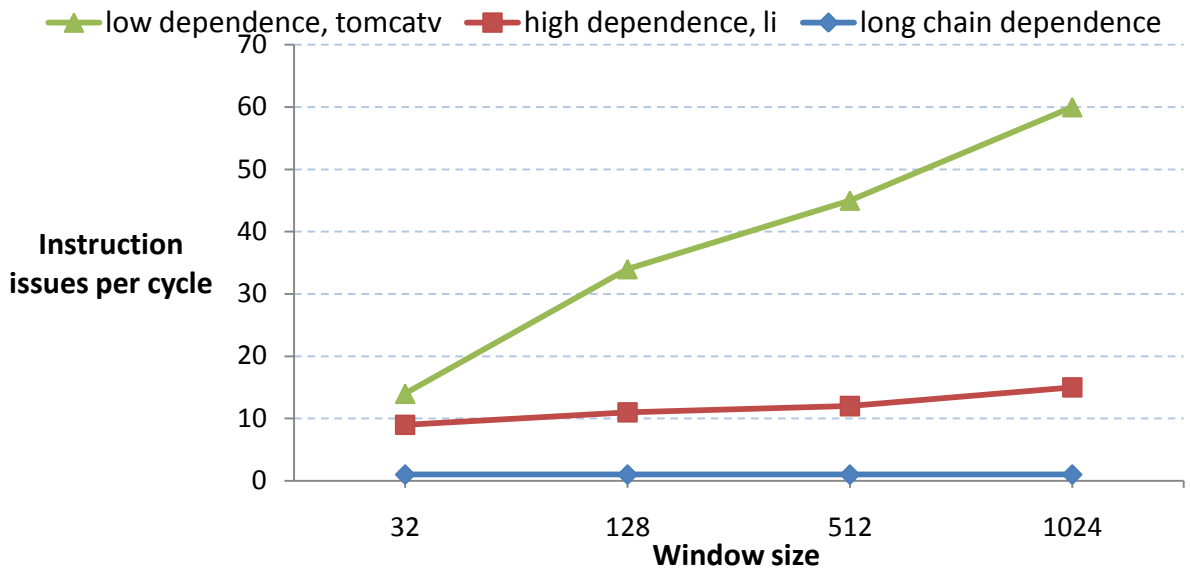


Figure 17. Ideal processor performance as a function of window size for different data dependence levels. The data dependence level is higher in li than in tomcatv. Thus, li's performance is less "sensitive" to window size. If the window size is reduced from 128 to 32 entries, for example, performance drops much less significantly for li than does for tomcatv. For a long chain of data dependence, performance is completely insensitive Data from Hennessy and Patterson [37].

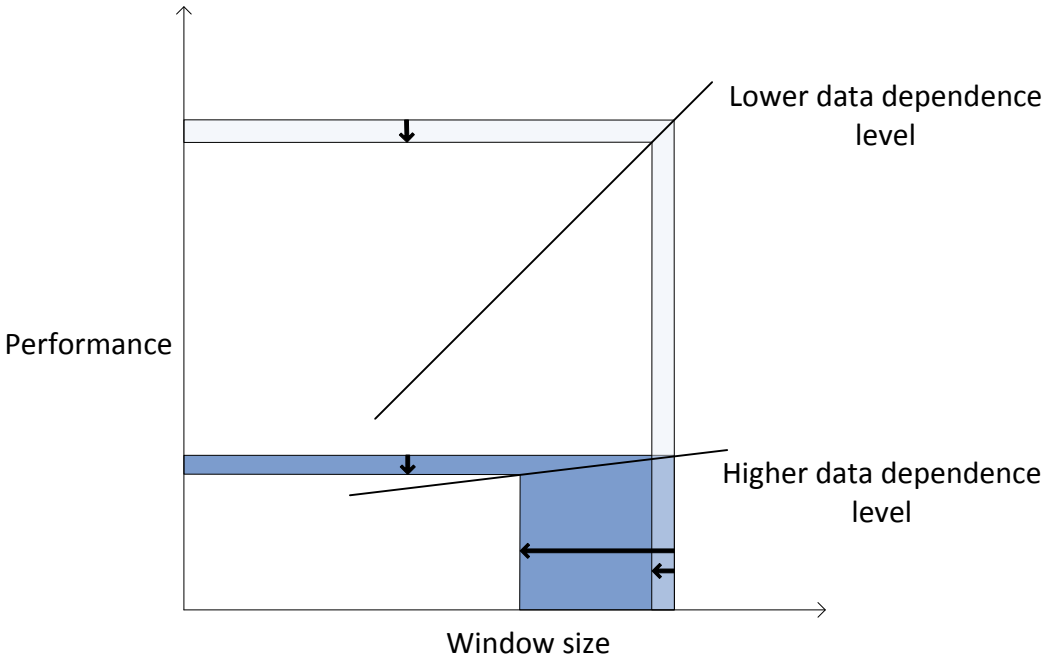


Figure 18. Ideal processor trading off window size for performance at different data dependence levels. At higher data dependence levels, a more significant decrease in window size can be traded off for smaller performance loss than can be traded at lower data dependence levels.

In both cases, components can be forced to reduce their occupancy by downsizing. This concept is illustrated in **Figure 19**. Components should maintain maximum size when data dependence exhibits neither high nor low levels; that is, when the engine holds a roughly equal mix of independent and dependent instructions. In other words, maximum size should be maintained when neither Case 1 nor 2 applies. The sizing decision must be solely based on the state of data dependences; otherwise, the engine's ability to find and issue independent instructions might be hindered.

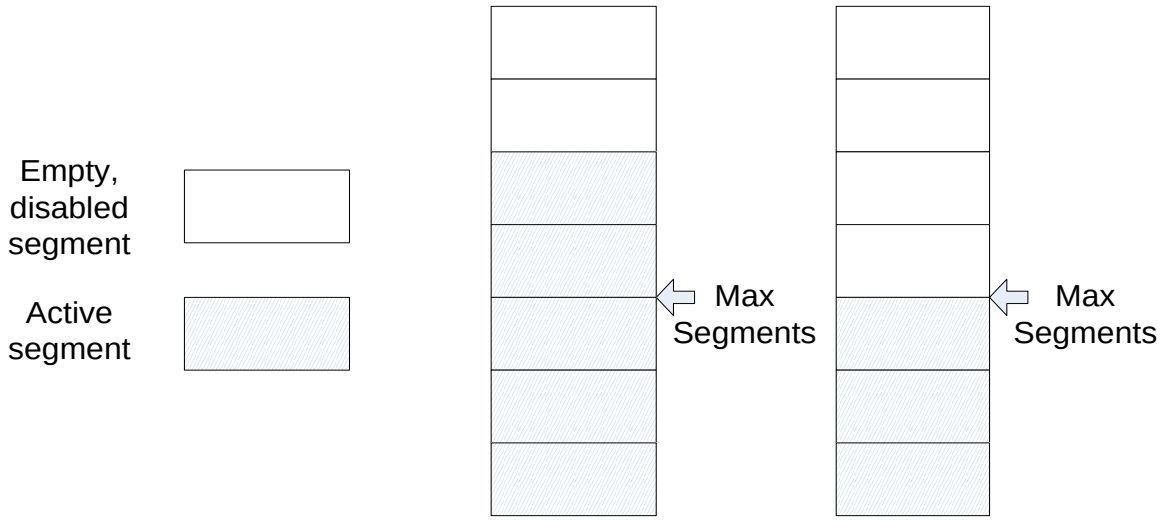
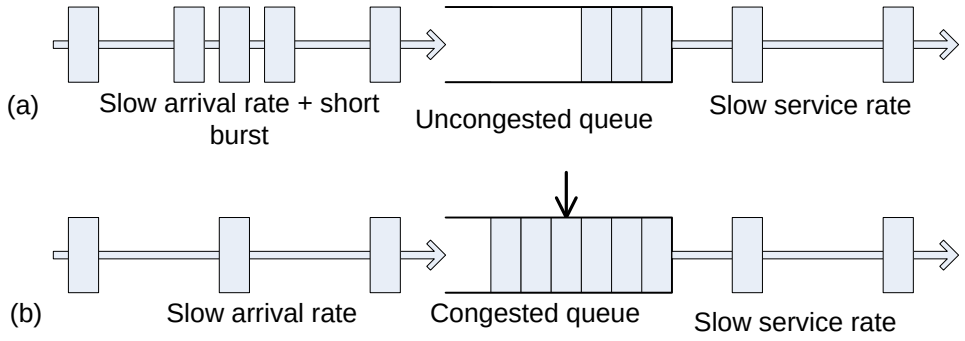


Figure 19. Downsizing with active occupancy reduction. Segments above the max segments pointer are allowed to drain so they can be disabled.

3.1.2 Principle 2

Continuously moderate the front-end pressure to reduce traffic bursts. The front end is normally designed to fetch instructions from the i-cache and push them into the out-of-order engine as fast as possible. However, the engine may not always be able to move instructions as fast as they come in. It might be slowed down, for example, by data cache misses, function unit latencies, or resource conflicts. Because of this constant front-end pressure, any empty entries in the engine are quickly filled up by short dispatch bursts. Even if traffic is reduced, engine queues can still maintain high occupancy. This scenario is graphically illustrated in **Figure 20**. To help maintain a lower occupancy, it is important to not only reduce traffic, but also to dampen short bursts above the currently sustained dispatch rate.



3.1.1 Tandem Principle

Figure 20. A short traffic burst as a result of front-end pressure, in (a), can raise occupancy in the queue in (b). The queue can then stay congested even if traffic is reduced.

For a significant occupancy reduction, both principles one and two must be applied. Applying Principle 1 alone does not cover cases when the engine is slowed down by microarchitecture-dependent latencies. For such cases, applying Principle 2 helps reduce occupancy when Principle 1 alone cannot. Conversely, applying Principle 2 alone does not cover cases when the engine can be downsized based on microarchitecture-independent latencies (namely data dependences). In such cases, applying Principle 1 helps reduce occupancy when Principle 2 alone cannot. For example, when data dependences are very few, traffic can be high so Principle 2 would not apply, yet the engine can still be downsized by Case 2 of Principle 1.

3.2 Downsizing the Out-of-Oder Engine

3.2.1 The Data Dependence Index

In order to downsize the out-of-order engine's components based on data dependences, as outlined in Principle 1, a metric for data dependences is needed. The latency metric proposed by out-of-order scheduling techniques often involves a data dependence order [54-56]. However, downsizing components based on such metric results in a large performance loss. This is because this metric includes microarchitecture latencies unrelated to data dependences.

As Principle 1 states, downsizing on any condition other than data dependences hinders the instruction window's ability to find and issue independent instructions.

Dynamically building an entire dependence graph in the issue queue is impractical due to high hardware complexity. Measuring the longest data dependence path is more practical. However, the longest path alone does not account for instructions that are not part of it, and thus does not fully capture data dependence parallelism. Therefore, we propose the Data Dependence Index (DDI) as the ratio of the number of instructions in the longest path, Nmax, to the total number of instructions in the graph, N, expressed as:

$$DDI = \frac{Nmax}{N} .$$

Nmax = Pmax + 1, where Pmax is the number of edges in the longest path. If all instructions are independent, then Nmax = 0. The range of the DDI is a fixed scale from zero to one ($0 \leq DDI \leq 1$). As more instructions become independent, the DDI approaches zero, because the longest path becomes shorter, containing fewer instructions. On the other end, as more instructions become dependent, the DDI value approaches one because the longest path becomes longer, containing more instructions. If the DDI = 1, then all instructions form one long dependence chain. **Figure 21** gives examples of the DDI. In (a) the DDI is greater than in (b), indicating greater data dependence. **Figure 22** gives examples of the DDI that are counter intuitive: the graph in (a) appears to have more data dependence than (b); however, both graphs in (a) and (b) have the same DDI.

The DDI can be theoretically justified using the ideal processor concept, such as the one defined by Wall [36]. In such a processor, the issue rate is a function of only data dependences. To derive an expression for this issue rate, we can realize that all instructions, N, in a graph can issue in the time it takes to issue the longest path's instructions. Thus the ideal issue rate can be expressed as N/(Pmax + 1), where Pmax is the number of edges in the longest path. The ideal issue rate is almost the inverse of the DDI, except that its range is not bounded between zero and one as with the DDI.

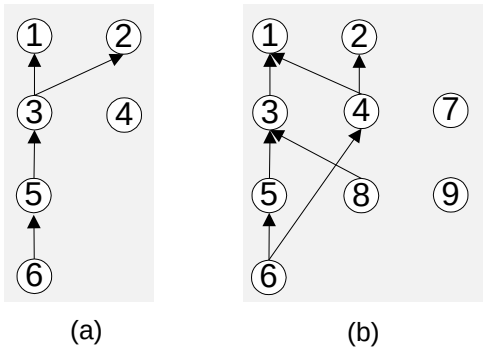


Figure 21. Data dependence graphs with instructions numbered in their program order. In (a) Nmax = 4 and N = 6, giving DDI = 4/6. In (b) the DDI = 4/9.

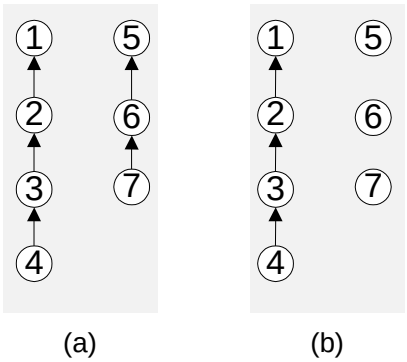


Figure 22. The graph in (a) appear to have more data dependence than the graph in (b); however, counter intuitively, both graphs have the same DDI = 4/7.

Applying the ideal issue rate directly instead of the DDI is impractical because of its range. As data dependences decrease and become very few, the ideal issue rate becomes very large and falls outside the useable dynamic range of size adjustment for the out-of-order engine's components. The DDI, on the other hand, quickly goes to zero, then equals zero when all instructions are independent. In other words, the fixed DDI scale between zero and one is more practical for size adjustment.

The DDI measures data dependences for all instructions, including those from a predicted path. This implies that the window size is adjusted to data dependences from whichever path is currently taken. If the prediction turns out to be correct, then the window size was properly adjusted. If the path was mispredicted, the window will adjust when the correct path instructions are fetched and dispatched.

Modern processors cannot determine data dependences through memory at the issue stage. Likewise, the DDI does not measure dependences through memory; the DDI measures only data dependences through registers. However, modern processors still issue loads and stores out of order by relying on special disambiguation techniques and memory address speculation [59, 60]. These disambiguation techniques verify that data dependences are not violated at the retirement stage. So as far as adjusting the window is concerned, loads and stores are considered to be independent (at least at the issue stage) and do not affect the DDI.

3.2.2 Token Passing Algorithm

To dynamically measure the longest data dependence path, we use a token-passing algorithm. The algorithm was inspired by Fields et al.'s algorithm that tracks the critical path for instruction scheduling [54]. The algorithm tracks only data dependences as instructions issue and does not need a predictor, resulting in a simpler hardware implementation. The proposed algorithm is for issue queue instructions. An alternative algorithm for the reorder buffer is proposed in the Future Directions chapter.

Each instruction entry in the issue queue has a token field that represents the instruction depth in the dependence chain. When instructions are dispatched into the issue queue, their token is initially set to zero, which means that they do not yet have a token. Independent (or ready) instructions, whose operands are available in registers, are never part of a chain, and thus their token is never inherited. Dependent instructions will eventually inherit a token from previous instructions before they issue.

Token creation and increment: When an instruction issues for execution, its token is incremented by one. If it is incremented from zero to one, then a token is said to be "created". This could be the head of a new chain.

Token passing: Normally when an instruction issues to execute, it passes its result tag to the issue queue so that waiting instructions can compare the result tag against their operand tags. If there is a match, the corresponding operand is marked as ready.

Token passing can happen at the same time. When an instruction executes, it passes its incremented token along with its result tag. Waiting instructions can then inherit this token when they compare their operand tags to the result tag. To ensure that the token from a longer path is propagated when a chain merges, each waiting instruction has to compare two tokens (assuming dyadic instructions) and store the larger one as follows:

- If the waiting instruction matches two result tags (A and B), then compare token A to token B and store the larger.
- If the waiting instruction matches one result tag, then compare the corresponding token (A or B depending upon which tag matches) to the instruction's existing token and store the larger.

Figure 23 (a) summarizes the token passing algorithm in a graph, and **Figure 23** (b) shows the logic needed to compare and store instruction tokens.

Token termination: If an instruction executes and no waiting instruction inherits its

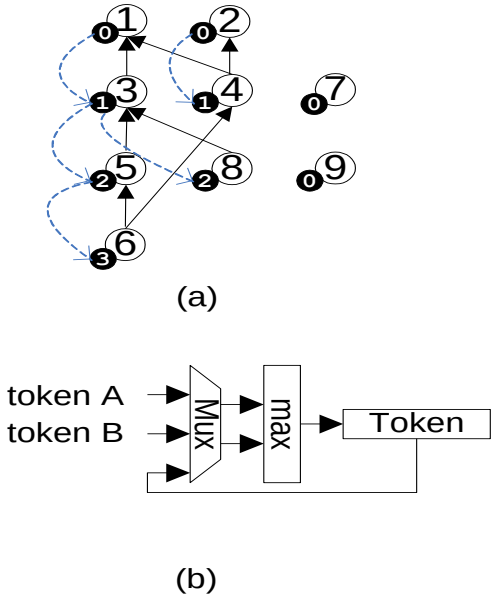


Figure 23. Token passing is shown in (a). As instructions issue, instruction No. 1's token (small black circle) is passed all the way to instruction No 6. The logic for comparing and storing the largest token is shown in (b). The Mux either passes both token A and B, or passes token A or B and the existing token. This Mux is controlled by the matching of result tags with operand tags.

token, that token is said to "terminate", indicating the end of a chain; that is, the window is not large enough to pass this token to future instructions.

Token termination: If an instruction executes and no waiting instruction inherits its token, that token is said to "terminate", indicating the end of a chain; that is, the window is not large enough to pass this token to future instructions.

The DDI: As instructions issue during each cycle, the maximum token among the issue group is compared to a global maximum. The global maximum stores the issue group maximum token if it is greater than its existing value. The global maximum then holds the longest path, Pmax, that the window "sees" over any time period until Pmax is reset again. A sample DDI is computed using Pmax over the time period it takes to issue N = 8 instructions, then Pmax is reset again. Once 128 DDI samples are accumulated, a DDI interval average is taken, which is used to adjust the window size for the next interval. On average, this interval lasts about 512 clock cycles.

Complexity: Because Pmax is computed over 8 instructions, a 3-bit token is sufficient to hold the largest value. Compared to other issue queue entries, such as instruction opcode, operands, operand flags, and operand tags and matching, the 3-bit token adds negligible storage to the issue queue. The token does affect the tag-drive width because the token is passed along the results tag. It also slightly increases the number of bit-lines for reading tokens as instructions issue. In the worst case scenario, if the issue queue is completely full all the time (empty segments are not turned off) and all devices are switching, the power models used in this work estimate about 6% power overhead in the issue queue and 0.5% in the total processor. However, with segmentation of the bit-lines, reducing instruction occupancy, and turning off empty segments, this overhead becomes practically undetectable, especially for the total processor.

Moreover, computing Pmax requires eight comparators (8 issue width), each 3-bits wide, which we assert requires a negligible power increase. Pmax also requires 3-4 logic levels, which are well within the optimum cycle of 6-8 fan-out-of-four (FO4) logic levels [61]. The DDI

sample can be computed with a small ROM lookup table. Also, both the sample DDI and the interval DDI computations are not on the critical path, so they can overlap with Pmax and the token passing activities and can take more than one cycle if necessary.

3.2.3 Extended Register-Renaming Algorithm

This is an alternative method to the token-passing algorithm to measure the DDI. Measuring the DDI can be accomplished at the rename stage by extending the dependence-check-logic. As instructions are decoded, logical register operands are renamed to physical registers to remove any name dependences introduced by the compiler. When the dependence-check-logic determines that an instruction's source operand is dependent on a previous instruction's destination operand, it renames the producer and consumer to the same physical register.

The check-logic can be extended so that each time data dependence is detected for an instruction, its depth in the dependence chain is incremented by one and propagated to the next instruction in the chain. A comparator is needed to propagate the maximum dependence for two source operands for the same instruction. In other words, if an instruction is the merge point of two dependence chains, only the longest branch of the chain is propagated to the consumer instruction. A final comparator determines the maximum of all dependence chains lengths to find the DDI. An average DDI is then computed over 512 clock cycles. This averaging helps to smooth out any irregular DDI samples.

Complexity. Since the DDI is measured every time over the same number of instructions, *e.g.*, over 8 instructions at dispatch, the ratio is not needed and the longest path itself is equivalent to the DDI. **Figure 24** illustrates a simple example for obtaining the LDPR over 4 instructions. The magnitude of the measurement of the longest path is on the order of $log_2 n$ where n is the number of instructions. Moreover, since n, the dispatch width, is small, the complexity and overhead to obtain the DDI at rename is minimal compared to the overall dependence-check-logic. For instance, if the dispatch width is 4 instructions, the hardware needed to measure the DDI is 4 comparators and 3 increment-by-1 counters, all of which are two bits wide.

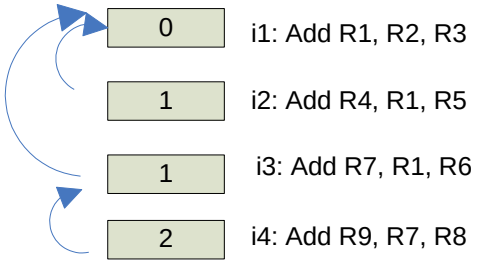


Figure 24. An example of four instructions at rename. The shaded boxes represent dependence depth counters. The maximum dependence value for all chains is 2 and the DDI = 2/4.

3.2.4 Validation

To validate the implementation of the DDI algorithm, the measured DDI must be compared to a known data dependence condition. Two loops were hand-coded in assembly language so that their performance was a function only of data dependences. The first loop had most of its instructions dependent on each other in a long chain (DDI near one), whereas the second loop had most of its instructions independent of each other (DDI near zero). The loops were coded so that instruction latency was least dependent on the microarchitecture. This implies that instructions hit in the caches, loop branches were predicted correctly and instructions did not contend for functional units. The two loops were then run several hundred times. The measured issue rate, which is near ideal, is then compared to the measured DDI for validation.

During the run of the first loop (with long chain dependences), the measured DDI was observed to be near one while the issue rate was also near one. During the run of the second loop (no data dependences), the measured DDI was observed to be near zero while the issue rate was near the maximum issue width. These observations confirmed correct implementation and allowed proceeding with the window adjustment using the DDI.

3.2.5 Downsizing Policy

Data dependences tend to change in phases during program execution because they inherently relate to program structures. **Figure 25** shows DDI graphs during the execution of randomly selected periods of two SPEC_2000 benchmarks. A long running loop, for example, might have many instructions dependent on each other. Even though different iterations of the loop might be independent, the processor window may not be large enough to "see" beyond one loop

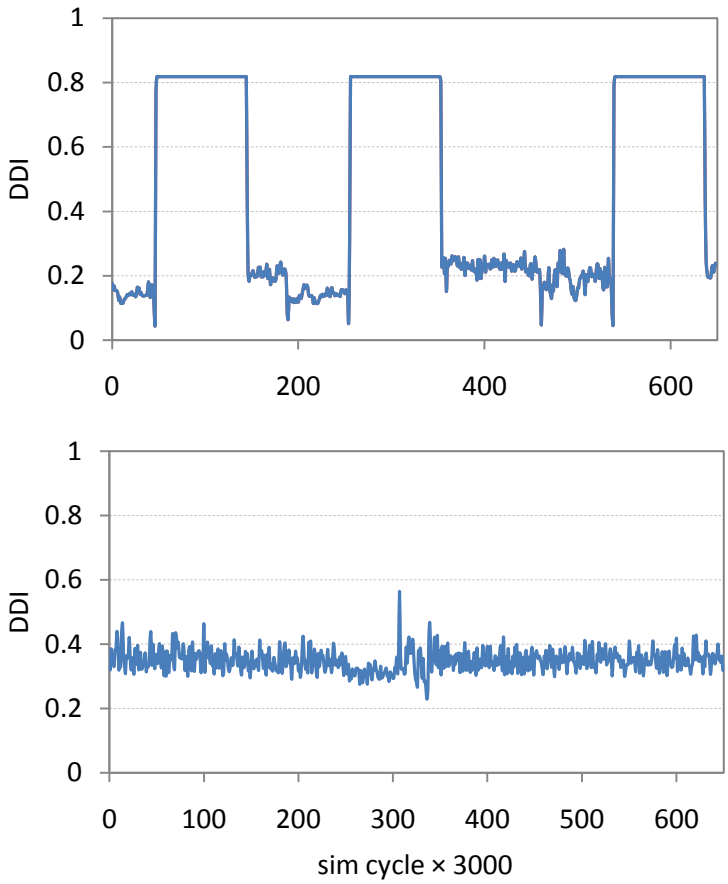


Figure 25. During two randomly selected periods, the DDI for the 164.gzip.log benchmark (top) alternates between high and low periods. The DDI for the 175.gcc benchmark (bottom) remains fairly constant around a value of 0.36.

iteration. A different loop, on the other hand, might have many independent instructions. Because of this phase pattern, the DDI should be measured over short intervals so that an efficient window size can be set for the duration of a phase.

To adjust the instruction-window size, the architecture and the power models were segmented for the issue queue, the load-store queue, and the reorder buffer. Past studies, for example Ponomarev et al. [16], showed power saving benefits from turning off issue queue segments whenever they become empty. In this work, we proactively set a maximum number of useable segments based on the interval DDI. This reduces the number of instructions and forces more segments to become empty and thus be turned off. In this work, this mapping from DDI values to the maximum useable segments is called the DDI policy.

An initial DDI policy was first derived based on the two cases of Principle 1; that is, the maximum number of active segments was limited as the DDI value increased above a threshold

and as it deceased below another threshold. The threshold DDI values and the rate of increase and decrease were then empirically optimized with initial simulation tests. These values were then applied to all subsequent simulations. **Figure 26** shows a graphical representation of the DDI policy.

When the maximum window size was adjusted, the issue queue, the load-store queue, and the reorder buffer were adjusted together and kept to their original proportions: the reorder buffer was set to hold twice as many entries as the issue queue or the load-store queue. However, since the reorder buffer segment size (16 entries) is twice as large as the issue queue or the load-store queue segment size (8 entries), the maximum useable segments remain the same for all three units. So, for example, when the maximum number of useable segments is set at 4, the reorder buffer can have up to 4×16 = 64 entries, and the issue and load-store queues can have up to 4×8 = 32 entries each.

As **Figure 26** shows, the DDI policy assigns the maximum useable segments to the physical window size of 8 segments for DDI values between 0.17 and 0.55. In this range, the window contains a mix of data-dependent and independent instructions, so that the window's performance is sensitive to its size. In other words, in this range a larger window can effectively find more independent instructions to issue.

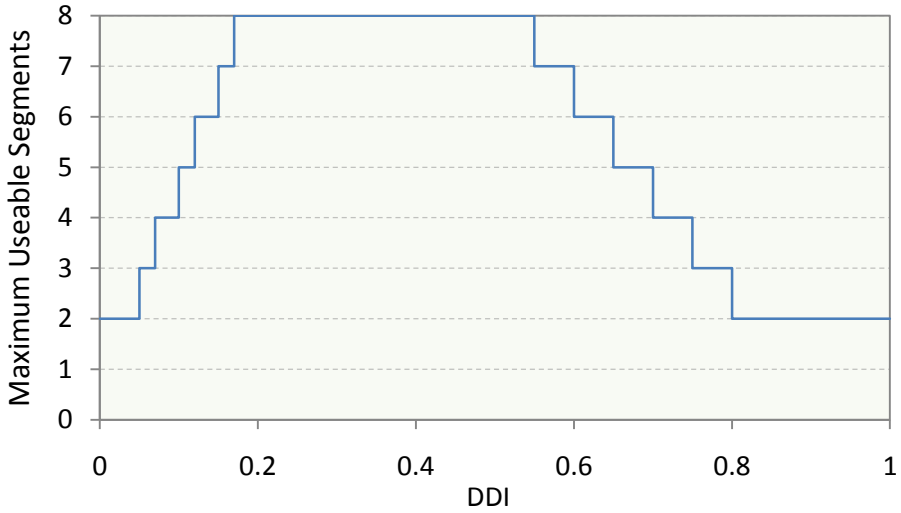


Figure 26. A graphical representation for the DDI policy.

As the DDI value increases above 0.55, the window size is gradually decreased as fewer independent instructions are found and the window's performance gradually becomes less sensitive to size. Less sensitivity means that a smaller, more efficient window can perform just as well. When the DDI reaches 0.8, most instructions essentially become dependent and the window becomes completely insensitive to size. Two active segments can then perform just as well.

As the DDI value decreases below 0.17, independent instructions dominate. Because of limited microarchitectural resources (such as limited issue width), the window can find enough independent instructions in a smaller instruction pool to sustain the maximum issue rate. Thus, the window size is gradually decreased. Below a 0.05 DDI value, sufficient independent instructions are available so that only two segments can sustain maximum issue rate. Because of the nature of the DDI ratio, the DDI value goes to zero faster when independent instructions dominate than it goes to one when dependent instructions dominate, as shown in **Figure 26**. This explains why the DDI policy curve is steeper on the left and slightly asymmetric.

To control the window size, three counters were assigned to keep track of the number of non-empty segments in each of the reorder buffer, the issue queue, and the load-store queue. These counters increment and decrement each time a segment is turned on and off, respectively, in each unit. If incrementing one of the units' counters results in exceeding the maximum number of useable segments set by the DDI policy, instruction dispatch is momentarily halted. When entries are freed up by instructions issuing or committing, instruction dispatch resumes.

3.3 Moderating Front-End Traffic Pressure

To find the most effective method for reducing traffic, three variations on fetch pausing aimed at moderating the front-end pressure were implemented.

3.3.1 Fetch Pause Based on Data Dependence

The first variation pauses fetching on any cycle if the out-of-order engine has a high level of data dependences:

```
FetchPause-DDI:
if (highDDI && fBufferNum > 0) {
            fetchPause = true;
} else {
            fetchPause = false;
}
```

Here the fBufferNum > 0 condition ensures that fetching is not paused if the fetch buffer is empty so that the engine does not "starve". The highDDI is a flag set if the interval DDI is greater than 0.4. Although this variation can slow down traffic when the engine slows down due to a high level of data dependences, it does not consider other microarchitecture latencies.

3.3.2 Fetch Pause Based on Sustained Dispatch Rate

To consider microarchitecture latencies, a second variation on fetch pausing was implemented. Fetching is paused on any cycle if the fetch buffer occupancy is higher than the sustained dispatch rate (SDR). In this case the SDR is the dispatch rate averaged over intervals of 512 cycles:

```
FetchPause-SDR:
if (fBufferNum > SustainedDispatchRate + delta) {
            fetchPause = true;
} else {
            fetchPause = false;
}
```

The delta parameter is a safety margin to "cushion" fast deviations in the SDR and avoid the risk of starving the engine. A delta = 2 was empirically found to be optimal. The SDR is determined by all microarchitecture latencies. The advantage of this fetch pause variation is that it can moderate traffic bursts above the SDR. This concept is illustrated in **Figure 27**. If the engine has many empty entries and dispatch bursts by the front-end try to fill it up, the bursts are dampened because the fetch buffer would be frequently drained by the fetch pause.

3.3.3 Fetch Pause Based on Sustained Dispatch Rate and Occupancy

The third variation is the same as the second except that an issue queue occupancy condition is added. Fetching is paused not only when the fetch buffer occupancy is higher than the dispatch rate but also when the issue queue has more than four full segments:

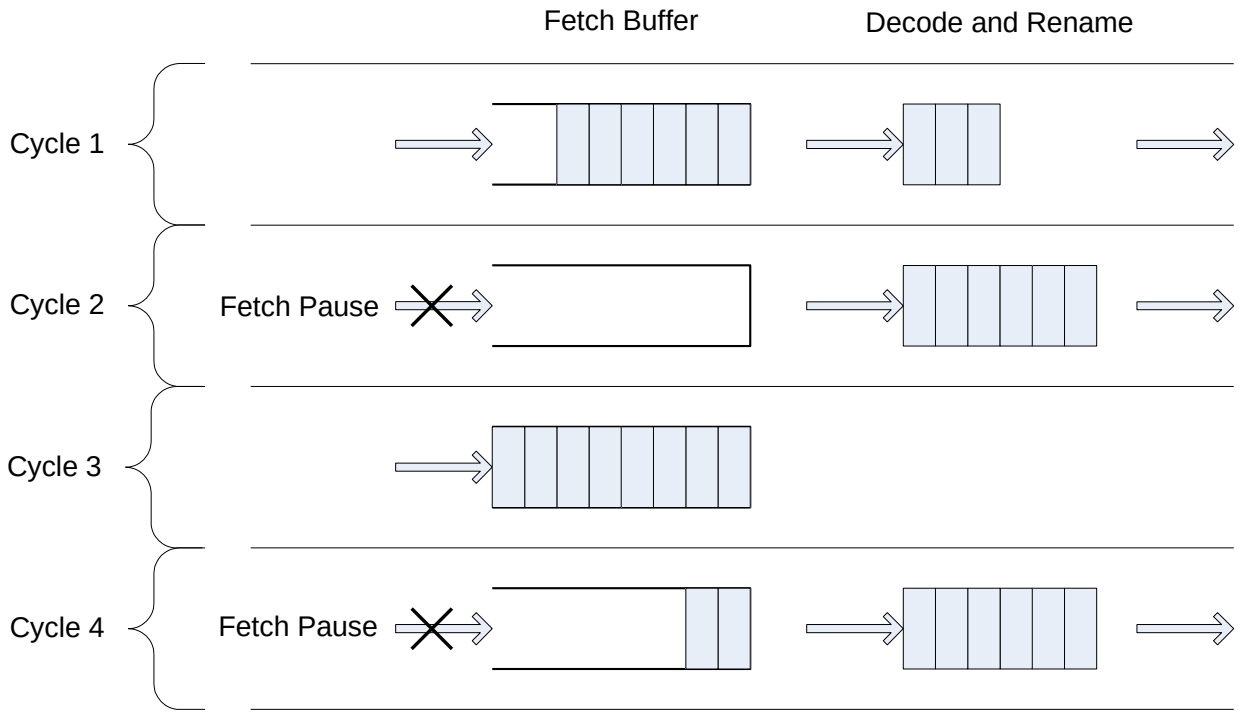


Figure 27. Fetch pause based on the fetch buffer occupancy and the sustained dispatch rate (SDR). In cycle 2, fetch is paused because the fetch buffer holds more instructions than the SDR. This can cause the fetch buffer to empty. In cycle 3, fetch is resumed, but no instructions move downstream because the fetch buffer was empty cycle 2. In cycle 4 fetch is paused again. The behavior can repeat as long as the front-end is trying to push instruction faster the SDR, in effect dampening traffic bursts above the SDR that can raise occupancy in the out-of-order engine.

```
FetchPause-SDR+IQ:
if ((fBufferNum > SustainedDispatchRate + delta) &&
      (IQ_num >= 4 * segSize)) {
            fetchPause = true;
} else {
            fetchPause = false;
}
```

IQ_num is the instruction occupancy in the issue queue and segSize is the segment size (8 for the issue queue). The rationale for adding the issue queue occupancy condition was to avoid the possibility of "starving" the out-of-order engine.

3.4 Clustering Fetched Instructions

The FetchPause-SDR and FetchPause-SDR+IQ methods are designed to take advantage of an energy saving bonus in the i-cache. Because an i-cache access energy cost is the same however many instructions are fetched, clustering as many instructions as possible into a single fetch is more energy efficient. The second and third variations let the fetch buffer drain first before fetching is resumed, creating ample space to cluster instructions. Instruction clustering was addressed by Buyuktosunoglu et al. [41]. However, their technique does not consider the fetch buffer occupancy for effective clustering. Other techniques proposed to reduce traffic did not consider instruction clustering [19, 21, 40].

3.5 Evaluation Framework

3.5.1 Architecture Models

The architecture models were simulated using SimpleScalar [25]. For the base model, SimpleScalar was modified to have a separate issue queue and reorder buffer. An 8-way, performance-driven model with parameters comparable to those used by similar studies, such as in [38, 41, 42], was chosen to help compare the results. **Table 2** lists the architecture parameters. Also, the issue queue and the reorder buffer were sized just large enough so that performance was not degraded (i.e., simulations for this work showed that a smaller 32-entry issue queue and a 64-entry reorder buffer degraded performance for some benchmarks by as much as 15%).

3.5.2 Power Models

Power consumption was modeled using the Wattch framework [24]. Wattch provides mathematical models for the capacitive loading for different types of processor circuits (RAM, CAM, clock, or complex logic) as a function of architectural unit sizes. The models can estimate dynamic power consumption in the early architectural definitions before committing designs to circuit floor-planning and layout.

The Wattch models were modified first to fit the base model with a separate issue queue and reorder buffer, both with appropriate read and write ports. The models were then decomposed to represent structural segmentation. This allowed only occupied segments to contribute toward total power during any given cycle. These segmentation models were then used for window adjustment and fetch clustering. We also used Wattch power models that are sensitive to data changes and that apply aggressive conditional clocking (cc3 mode). The goal was to use a base model that is already power efficient to which the occupancy reduction techniques could be applied to further reduce power.

Table 2 Processor parameters used for simulations.

Fetch Buffer	16 instructions, up to 2 branches
Decode/Issue/Commit Width	8 instructions
Issue Queue	64 entries
Load-Store Queue	64 entries
Reorder Buffer	128 entries
ALU	8 integer, 2 integer multiply/divide, 4 floating, 2 floating multiply/divide
I-cache-L1	2048 sets, direct, 32 bytes per block, LRU, 2 cycle hit latency
D-cache-L1	512 sets, 4 way, 32 bytes per block, LRU, 2 cycle hit latency
D/I-cache-L2	16K sets, 4 way, 64 bytes per block, LRU, 16 cycle hit latency

3.5.3 Simulation Methods

For a realistic estimate of performance, occupancy, and power consumption, the SPEC CPU 2000 benchmark suite from the University of Minnesota was simulated as a workload [26]. This benchmark suite is derived from SPEC CPU 2000 and is recognized by SPEC as a valid simulation tool for simulation based computer architecture research. The suite uses statistical sampling to maintain function-level execution patterns, instruction mix, and cache behaviors for entire benchmarks. The available suite compatible with “pisa” SimpleScalar architecture was used. The suite contains a mix of compression, graphics, scientific, and database benchmarks

[26] For each of the benchmarks, large input data sets were chosen for simulation, resulting in 700 million to 5 billion instructions per method per benchmark simulation.

To each benchmark, the following six methods were applied, which are designated as M1–M6 as follows:

M1: Base model was used to establish performance with no architectural or power model modifications.

M2: Empty segments were turned off whenever they became empty. No active restriction on window size or fetching was applied.

M3: The out-of-order engine was dynamically adjusted using the DDI policy. This implied simultaneously adjusting the sizes for the issue queue, reorder buffer, and the load-store queue.

M4: Fetch pause based on the window DDI (FetchPause-DDI) was added to M3.

M5: Fetch pause based on the sustained dispatch rate (FetchPause-SDR) was added to M3.

M6: Fetch pause based on the sustained dispatch rate and the issue queue occupancy (FetchPause-SDR+IQ) was added to M3.

In addition to performance and power statistics reported by SimpleScalar and Wattch, the simulator was instrumented to collect distribution data about the number of active segments and instruction fetches. Simulation data was then post-processed to obtain averages for all benchmarks for each of the M1–M6 methods.

4 RESULTS

This chapter reports the simulation results for the methods M1 through M6 that were described in Chapter 3. Section 1 reports instruction occupancy statistics for the out-of-order engine's components. The occupancy averaged over all benchmarks for each method is reported. Moreover, the percentage time distribution for the number of active segments in each of the out-of-order engine components is reported for each method.

The aim of the fetch pausing methods in M4–M6 is to reduce instruction traffic as outlined in Principle 2. To compare the methods' efficacy in reducing traffic, Section 2 reports instruction traffic statistics for all methods. Since two of the fetch pause techniques (FetchPause-SDR in M5 and FetchPause-SDR+IQ in M6) are also designed to take advantage of instruction clustering for extra power savings in the i-cache, Section 3 reports statistics for instruction clustering for all methods. The clustering statistics include the average number of instructions per single fetch, the distribution of the number of instructions per fetch, and the average fetch event rate.

Section 4 reports power consumption statistics generated by the power models for all methods. The power statistics are reported in two formats: the first is power consumption by each component broken down by method, and the second is total processor power consumption for each method broken down by component. The first format helps indentify how effective each method is in reducing power consumption for a specific component. The second format shows total processor savings for each method and the relative effect of each component on the total power savings.

Finally, Section 5 reports the methods' effects on the benchmarks' performance. The percent change in IPC performance is tabulated for each benchmark and for each method. The average performance effect for each method is also given in that section.

4.1 Instruction Occupancy Reduction

The average instruction occupancy for each of the six methods (M1–M6) was calculated for the reorder buffer (ROB), the issue queue (IQ), the fetch buffer (FB), and the load-store queue (LSQ). The instruction occupancy statistics are based on architecture models and are independent of the power models. **Figure 28** shows the average occupancy reduction for the M2–M6 methods. Turning off empty segments in M2 does not reduce instruction occupancy, since no size or flow restrictions were applied.

When the out-of-order engine was downsized in M3 with the DDI policy, instruction occupancy dropped 16-17% in the issue queue, the reorder buffer, and the load-store queue. At the same time, the fetch buffer occupancy remained constant in M3 because no fetch pausing restriction was applied. The fetch buffer occupancy dropped by 15% in M4 when the FetchPause-DDI condition was added (**Figure 28**).

However, when the sustained dispatch rate condition (FetchPause-SDR) was added in M5, the occupancy dropped the most in all components: 40% in the FB, 33% in the IQ, 27% in the ROB, and 27% in the LSQ (**Figure 28**). The decrease in the fetch buffer occupancy is a strong indication that the fetch buffer is getting drained at a much faster rate. This implies that traffic bursts are being dampened successfully, which is the aim of Principle 2.

When the IQ minimum occupancy condition was added to the fetch pause condition (FetchPause-SDR+IQ) in M6, the fetch buffer and the window occupancies substantially dropped as well, but not as much as they did in M5. The FB dropped 21%, the IQ 25%, and the ROB and LSQ 22% (**Figure 28**).

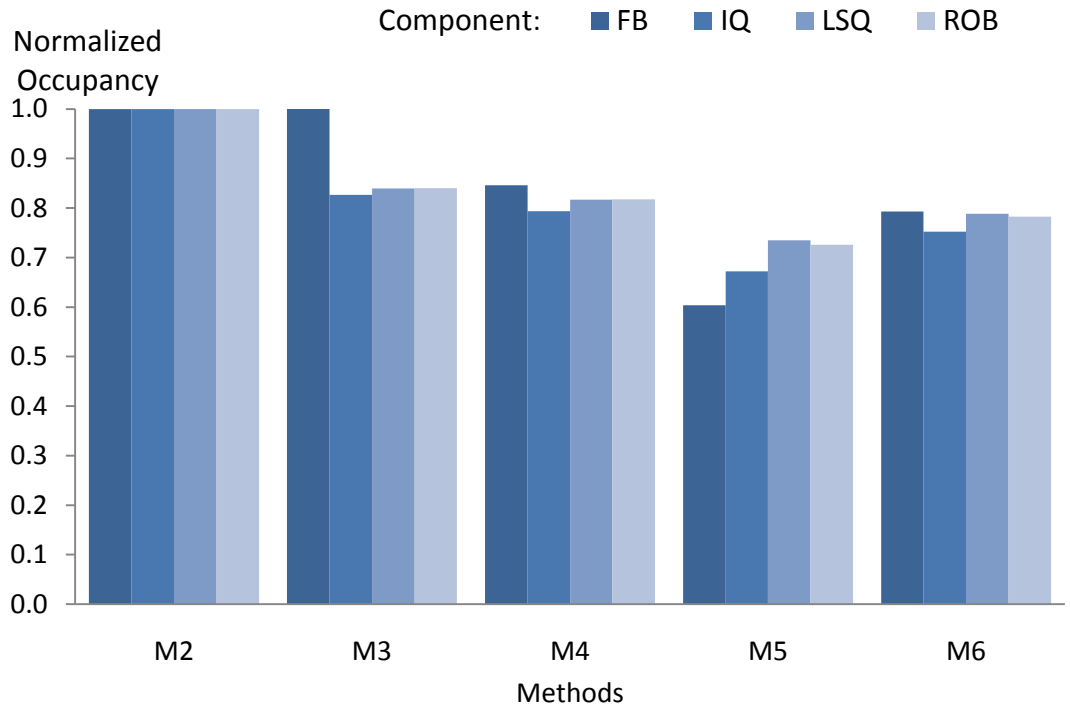


Figure 28. Reduction in instruction occupancy for the FB, the IQ, the LSQ, and the ROB. Occupancies are averaged over all benchmarks and normalized to M1.

As instruction occupancy is reduced, the number of active segments is also reduced. **Figure 29** shows the proportions of time (averaged over all benchmarks) of the number of active segments for the IQ, the ROB, and the LSQ. In M2, the IQ spends a large proportion of time having 8 active segments, but in M3 this time proportion is reduced in favor of smaller (2–5) active segments. This trend toward reducing large number active segments reaches a maximum in M5 where the IQ spends significantly more time having a smaller number of active segments, which helps reduce power consumption. This trend toward reducing the number of active segments is exhibited in the ROB and the LSQ as well. The number of active segments was computed based only on the instruction occupancy, which means it was based on the architectural model and is unrelated to the power models.

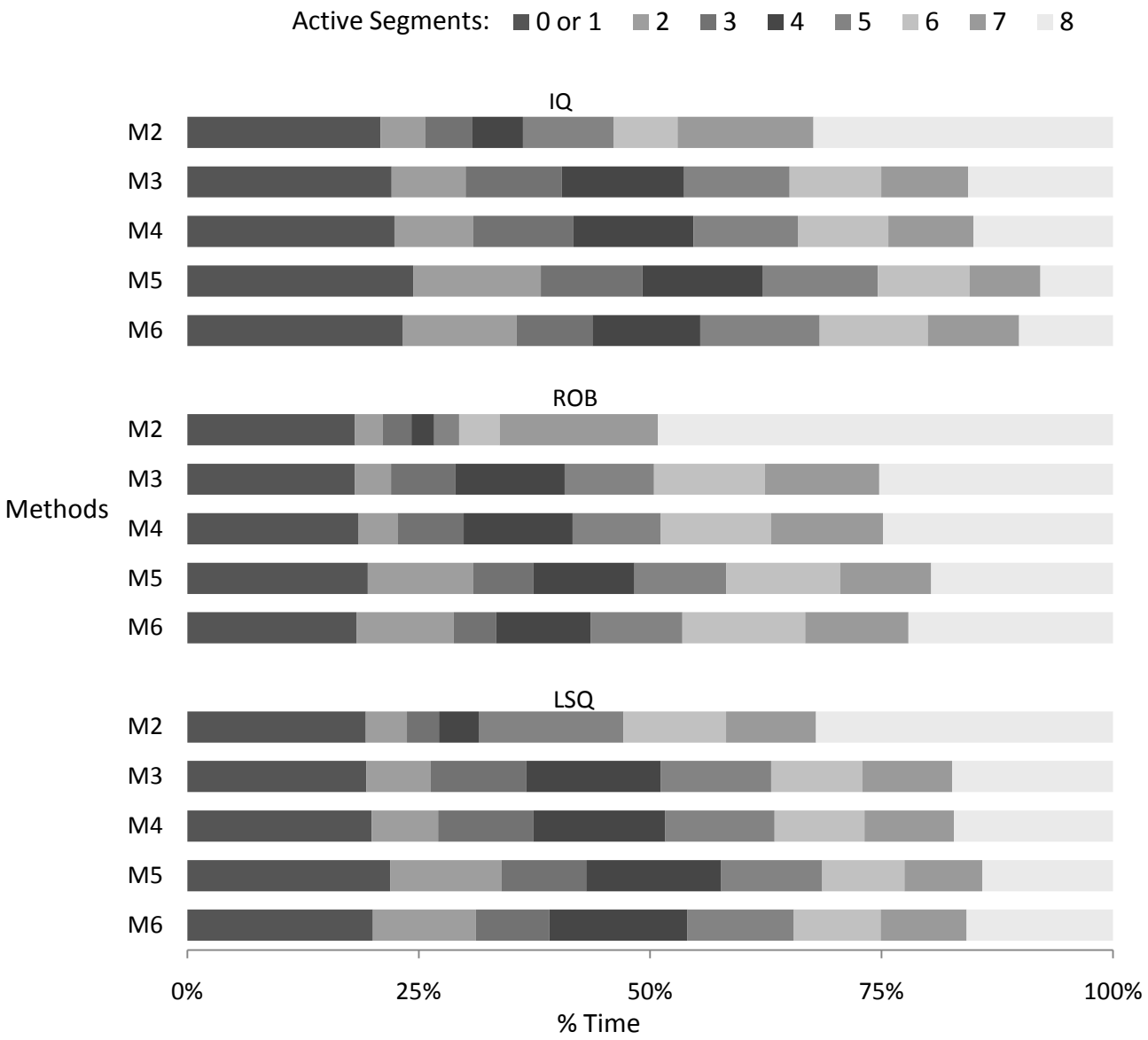


Figure 29. Percentage of time for the number of active segments for all methods. M1 is omitted as it does not support segmentation.

4.2 Traffic Reduction

To illustrate effects on reducing the front-end instruction traffic, **Figure 30** shows the average instruction fetch rate (instruction per cycle) for all methods. M5, with its FetchPause-SDR, reduced the instruction fetch rate the most, from 4 instructions per cycle to 3.5, or about 13%. The next best method in reducing fetch rate was M6, which is also based on the SDR.

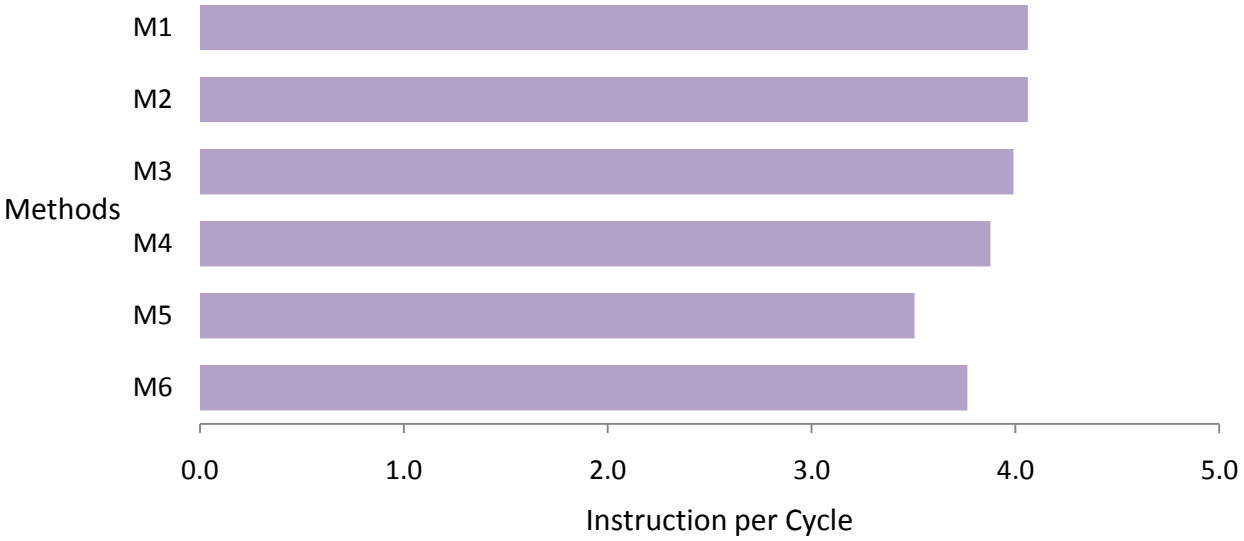


Figure 30. The average instruction fetch rate, which is reduced the most in M5.

4.3 Fetch Clustering

In addition to reducing traffic, the goal of methods M4–M6 was to increase the clustering of fetched instructions. To show the effect on clustering, **Figure 31** displays the average number of instructions in a single fetch or i-cache access for all methods. Pausing fetch based on the window DDI in M4 leads to a small increase in instruction clustering. However, when pausing fetch based on the SDR was applied in M5, the average instructions fetched from a single cache line almost doubled compared to M1, from about 7 to 14 instructions.

This doubling is significant, especially if we consider that the fetch buffer size is 16 instructions; that is, the number of instructions fetched in a single access went from less than half to almost the full fetch buffer size. This increase of clustering is also shown in **Figure 32**, which plots the percentage proportion of fetches by the instructions they carry; it shows that in M5 over 80% of fetches carry 13–16 instructions.

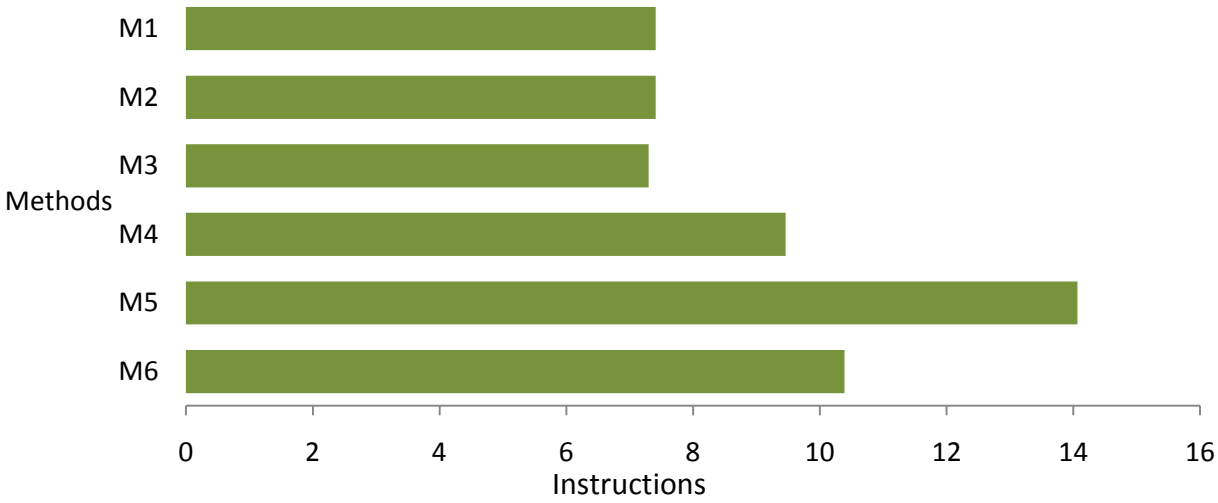


Figure 31. Average number of instructions per fetch, which is highest in M5.

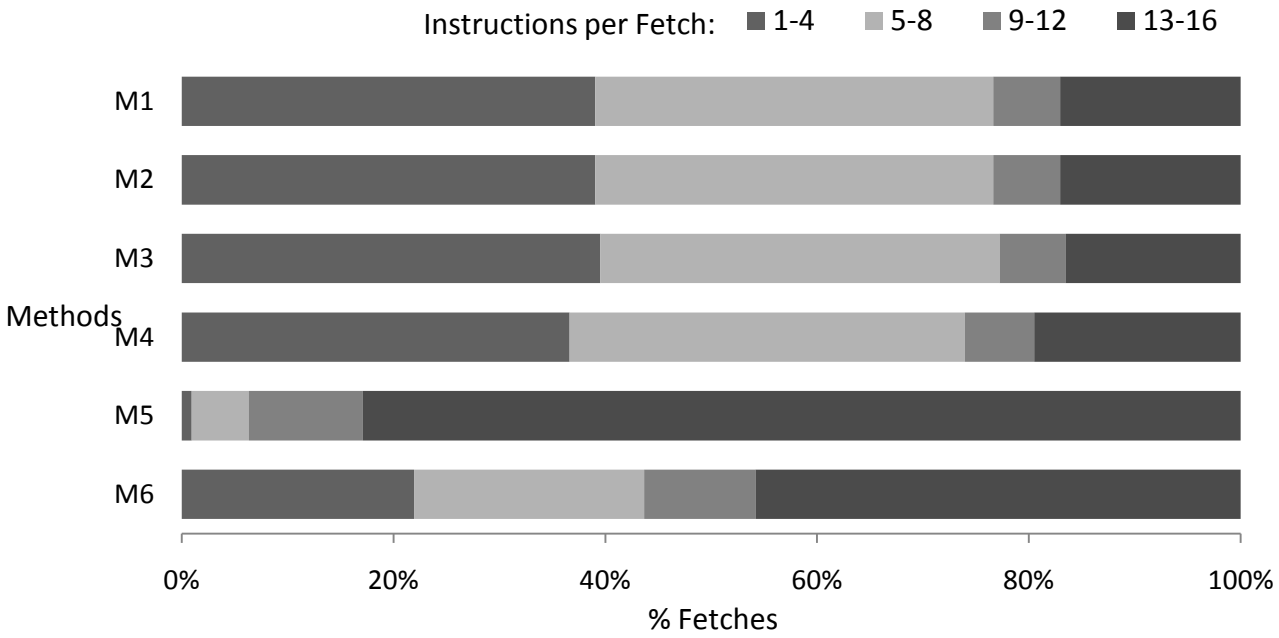


Figure 32. Percentage of fetches by the number of instructions they carry. In M5, over 80% of fetches cluster between 13–16 instructions in one i-cache access.

Another important variable to consider is the fetch activity on the bus that connects to the i-cache. **Figure 33** shows this activity in the fetch event rate (fetch per cycle). With effective clustering, bus traffic is reduced. M5 exhibits the lowest fetch activity, which is reduced by 56% compared to M1.

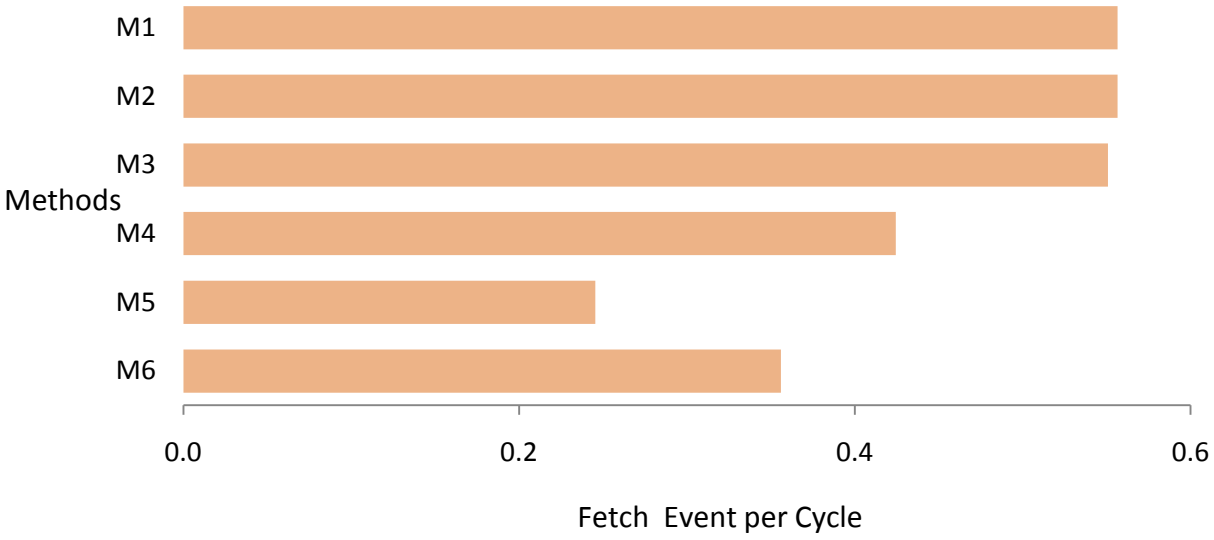


Figure 33. Average fetch event rate, which is also lowest in M5.

4.4 Power Savings

Along with occupancy and traffic statistics, dynamic power consumption in Watts was collected during the same simulations. Power savings are illustrated in **Figure 34**, which shows the average power consumption breakdown for the window, the load-store queue, and the i-cache for all methods. The window power component covers both issue queue and reorder buffer activities, which include wakeup and select-logic accesses, dispatching and issuing instructions, physical register accesses and allocating-deallocating reorder buffer entries. The i-cache power component reflects energy usage for cache line accesses. The result bus and the clock are also included in the figure, because their power consumption is related and was reduced. The result-bus component reflects energy spent transferring instructions and data between the issue queue, the reorder buffer, and the execution units. The clock component is proportional to global chip activities. The "other" component in the figure did not have much power change and includes the power for the branch predictor, the TLB, the rename and decode, and the d-cache activities.

Figure 35 shows the same data as in **Figure 34** but grouped by method in a stacked bar graph to highlight savings trends in total processor power consumption.

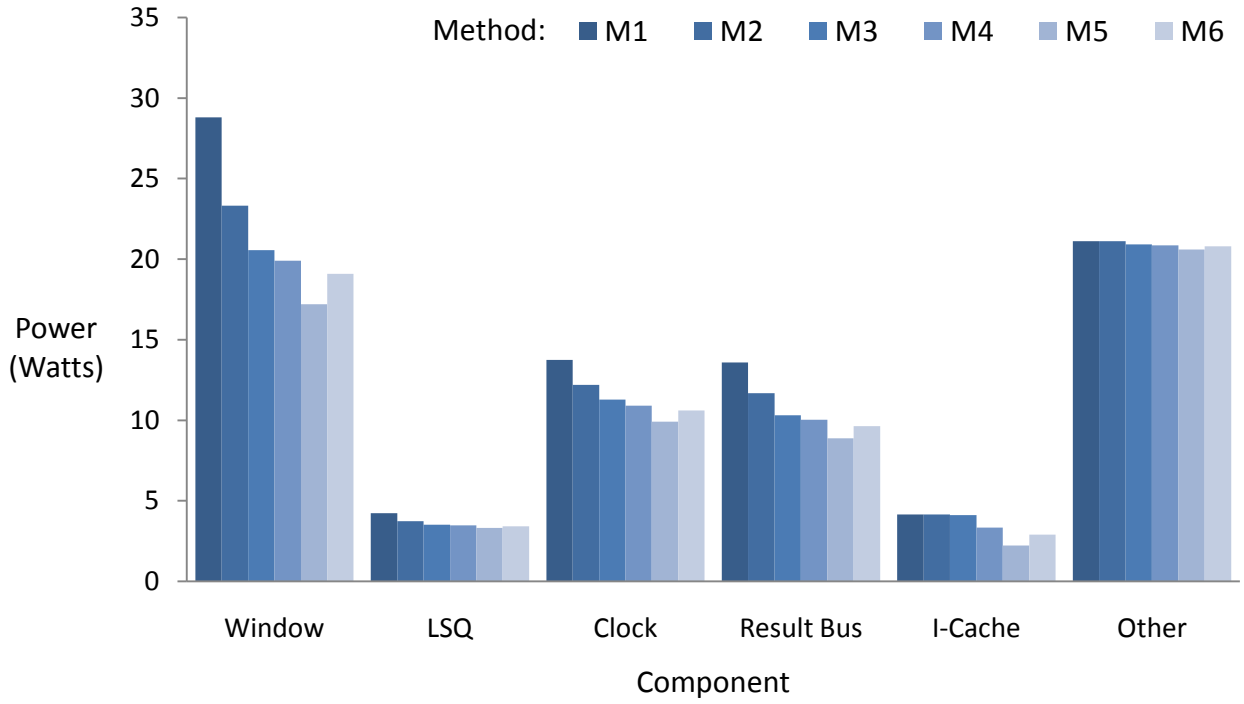


Figure 34. Total power savings for components broken down by methods. The window component includes the IQ and the ROB.

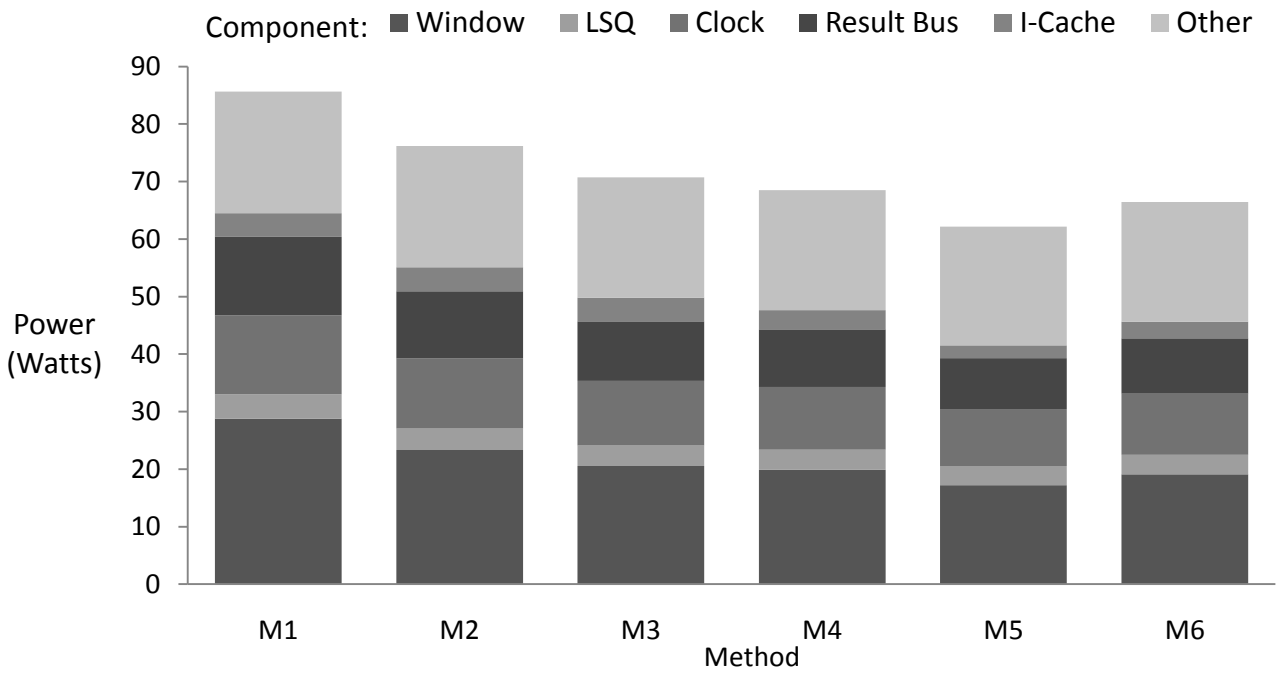


Figure 35. Total power savings for methods broken down by component to show total processor savings and relative effect of component's savings on the total.

When only empty segments were turned off in M2, total processor power consumption went down to 76 W from 85 W in M1, or 11% (**Figure 35**). When the out-of-order engine was downsized using the DDI policy in M3, the processor power went down to 70 W, or 17% less than M1, mostly due to window power reduction, which was reduced 29% compared to M1 (**Figure 34**). Pausing instruction fetch based on the window DDI in M4 yielded only a slightly greater reduction in power compared to M3. However, pausing fetch based on the SDR in M5 reduced the processor power to 62 W, or 27% less than M1. This substantial power reduction was largely due to window power reduction (40%), helped by additional i-cache power reduction (47%). M5 also reduced the result-bus power, largely due to reduced accesses, and it reduced clock power due to reduced global instruction activities. The power reduction in M5 was also consistent with M5's reduced traffic and increased fetch clustering. Finally, M6 was second best in power reduction with 22% total processor power reduction compared to M1.

4.5 Performance Effect

Turning off empty segments in M2 does not affect any performance compared to M1 because it does not apply any architecture restrictions. M3 and M4 had 1.5% and 1.8% IPC performance loss respectively. M5, the method with the most power savings, did only slightly worse with 2.6% IPC performance loss, and M6 performed slightly better than M5 with only 1.9% IPC performance loss. **Table 3** lists the IPC performance for all benchmarks for all methods. For each method, the average IPC for all benchmarks is used to calculate the percent change in performance compared to the M1 method.

Table 3 IPC performance for all benchmarks and for all methods. The bottom row represents the percent change of the average IPC over all benchmarks for each method compared to the M1 method.

	Methods					
SPEC Benchmark	**M1**	**M2**	**M3**	**M4**	**M5**	**M6**
164.gzip.graphic	1.9257	1.9257	1.9165	1.9069	1.8683	1.8886
164.gzip.log	2.2931	2.2931	2.2802	2.2755	2.2623	2.2682
164.gzip.program	2.2276	2.2276	2.2181	2.2115	2.1987	2.2033
164.gzip.random	1.2838	1.2838	1.2783	1.2599	1.2122	1.2547
164.gzip.source	1.7486	1.7486	1.7382	1.7314	1.7083	1.7197
175.vpr.place	2.4632	2.4632	2.4609	2.4582	2.4194	2.4496
175.vpr.route	2.5163	2.5163	2.4986	2.4977	2.4816	2.4946
176.gcc	1.1231	1.1231	1.1112	1.0937	1.0671	1.0832
177.mesa	4.1002	4.1002	3.8516	3.8621	4.0162	3.9272
179.art	3.1234	3.1234	3.029	3.0507	3.002	3.0507
181.mcf	1.9645	1.9645	1.9327	1.9351	1.9158	1.9351
183.equake	3.0728	3.0728	3.0462	3.046	3.0626	3.0627
188.ammp	1.4644	1.4644	1.4557	1.4348	1.4248	1.4348
197.parser	2.1958	2.1958	2.1954	2.1704	2.136	2.193
255.vortex	2.011	2.011	1.9633	1.9143	1.786	1.83
256.bzip2.graphic	2.8865	2.8865	2.8391	2.841	2.8261	2.841
256.bzip2.program	2.9001	2.9001	2.8636	2.8663	2.8501	2.8663
256.bzip2.source	2.8872	2.8872	2.86	2.8621	2.8481	2.8621
Average	2.3437	2.3438	2.3077	2.3010	2.2825	2.2980
% Change compared to M1	—	**0.00%**	**-1.54%**	**-1.82%**	**-2.61%**	**-1.95%**

5 DISCUSSION OF RESULTS

5.1 Validation of Principle 1

The objective of reducing instruction occupancy is to create more empty segments in the out-of-order engine's components than would normally exist. Empty segments can then be disabled to reduce power consumption. Two principles, along with a tandem principle to apply them together, are proposed to reduce instruction occupancy in the out-of-order engine. The principles were described in Section 3.1.

To validate Principle 1, the data dependence index (DDI) was conceived, and a token passing algorithm was developed to measure it (described in Section 3.2.1 and 3.2.2). Case 1 and 2 of Principle 1 could then be implemented through the developed DDI policy (described in Section 3.2.5) to downsize the out-of-order engine's components.

Results from simulation method M3, which implemented the DDI measurement and policy, show that instruction occupancy compared to the base model M1 was reduced in all out-of-order engine components by 16–17%. More specifically, M3 results also show that the out-of-order engine components spent higher percentage of cycles with fewer active segments. For example, in the segmented base model M2, the engine spent, on average over all benchmarks, less than 15% of cycles having 2-4 active segments and over 30 % of cycles having 8 active segments. In M3, the percentage of 2-4 active segments increased to about 30%, while the percentage of 8 segments decreased to about 15%. The fact that this significant decrease in instruction occupancy came at IPC performance loss of only 1.5% indicates the following:

- Both cases 1 and 2 of Principle 1 are validated. When they are applied, they will reduce instruction occupancy when data dependences become very high or very low.

- The proposed token passing algorithm and the DDI policy are validated as a means of applying Principle 1 to reduce instruction occupancy.

5.2 Validation of Principle 2 and the Tandem Principle

To validate Principle 2, three techniques were simulated. The objective in each of the three techniques with respect to Principle 2 was as follows:

- FetchPause-DDI, simulated via M4, was not complying with Principle 2 and the objective was to reduce occupancy based on the DDI state in the instruction window.

- FetchPause-SDR, simulated via M5, was fully complying with Principle 2. The object was to dampen traffic bursts based on the sustained dispatch rate (SDR).

- FetchPause-SDR+IQ, simulated via M6, was partially complying with Principle 2. The objective was similar to FetchPause-SDR except a condition for IQ occupancy is added. All three techniques are described in Section 3.3 and their simulations described in 3.5.

Simulation results on standard benchmarks show that instruction occupancy was reduced the most in M5 for the technique that is fully complying with Principle 2: 40% in the FB, 33% in the IQ, 27% in the ROB, and LSQ. The partially complying technique in M6 came in second in occupancy reduction: 21% in the FB, 25% in the IQ, and 22% in the ROB and LSQ. The non-complying technique in M4 showed only modest occupancy reduction over M3. These results indicate the following:

- Since the fully complying technique with Principle 2 showed the most reduction in instruction occupancy, Principle 2 is validated. The fact that the FB occupancy

dropped by 40% in this technique also indicates that the fetch buffer was being drained or emptied more often. As this happens, traffic bursts into the out-of-order engine cannot be sustained and are dampened, just as outlined by Principle 2.

- The FetchPause-SDR technique is validated. Conditioning the fetch pause on the fetch buffer occupancy and the sustained dispatch rate (SDR) dampened traffic bursts and reduced traffic.

- Since applying the FetchPause-DDI technique in M5 in addition to the DDI policy reduced instruction occupancy more than applying the DDI policy alone in M3, the Tandem Principle is also validated. That is, applying Principle 2 in tandem with Principle 1 significantly reduced in instruction occupancy.

5.3 Significance of Clustering Fetched Instructions

Clustering of fetched instructions has a direct effect on reducing i-cache power consumption. Since each i-cache fetch consumes the same amount of power, efficiency increases if each fetch is made to carry more instructions. Moreover, as more instructions are clustered together in each fetch, fewer fetches are needed for the same number of instructions. This helps reduce traffic activities on the bus connecting the front end to the i-cache.

When the FetchPause-SDR technique pauses fetch to let the fetch buffer drain, it allows space for more instructions to be clustered together when fetching is resumed. The results show that the average number of instructions per fetch almost doubled, from 7 to 14 instructions, in M5 compared to the base model. Since the fetch buffer size is 16 instructions, this increase shows that the fetch clustering went to almost the full fetch buffer size. The results also show that in M5 the fetch event rate or fetch activity was reduced by 56%.

5.4 Significance of Power Reduction Results

Just as expected, the power consumption results from the power models were closely associated with the instruction occupancy reduction. As instruction occupancy was reduced, more empty segments were disabled and power was saved. By applying Principle 1 alone with the DDI policy, total processor power consumption in M3 went down by 17%, mostly due to power savings in the IQ and the ROB, whose power consumption was collectively reduced by 29%. However, when the FetchPause-SDR technique was included, as Principle 2 and the Tandem Principle outlined, total power consumption in M5 went down by 27%. Just like in M3, the IQ and the ROB were also large contributors with 40% power savings.

Another contributor to the power savings in M5 came from the i-cache. As fetched instructions were clustered more effectively, the i-cache power models indicate that its power consumption was reduced by 47%. Another important contributor was the result bus and the global clock. As more segments were disabled, less signaling was required, thus global clocking and bus activities were also reduced.

These reported power savings are in comparison to the base models that include some power efficiency measures. The chosen Wattch power models for this work include conditional clocking of accessed unit ports; that is, only accessed ports on a unit contribute toward its power consumption. The chosen models also include the overhead for the conditional clocking. The reported power savings also include the overhead for segmentation. Care was taken to include driver power overhead per segment. That is, the assumption was made that each segment needs its own array drivers.

Although the power savings reported here are significant, they are not a solution by themselves to the processor power problem. This work addresses only the dynamic power consumption, but as the clock rate and the number of devices on a chip continue to increase with each technology generation, both dynamic and static power consumption will continue to increase. The power savings reported here at the microarchitecture level, in conjunction with other power saving measures at other design levels, will help curb the power increase. They

will help reduce cost, increase reliability, reduce environmental impact, and allow for continued performance growth for only some time, and only until a more radical solution is found.

5.5 Comparative Discussion

This thesis reports significant reduction in instruction occupancy in all pipeline stages, front to back. So far, it is the only known work that reports such statistics for power efficiency. Moreover, this thesis reports statistics for clustering fetched instructions.

Only Ponomarev et al. in [15] report power saving in all of out-of-order engine components. Although they use segmentation to reduce power, their technique is passive in occupancy reduction. They do not report instruction occupancy or total processor power savings, and they report performance loss of 5%, which is twice as much as the performance loss in M5. Folegnani et al. in [38] use the IPC to actively disable IQ segments. They do not report instruction occupancy statistics or total processor power consumption. Karakhanis et al. [19] also use the IPC to limit instruction count, but instruction occupancy statistics are not given, nor is the total processor power savings. To compare techniques that reduce the instruction count in the processor, at least the instruction occupancy should be reported for all major pipeline components.

Traffic reduction techniques by Manne et al. and Baniasadi et al. report traffic reduction statistics [21, 40]. However, traffic reduction does not necessarily reduce instruction occupancy, as explained in Section 3.1.2. Reducing occupancy is more effective in reducing power in segmented structures. Reporting total processor power consumption would help compare technique effectiveness, especially if they share the same power models. It would also be useful if component power consumptions are reported to show the effect of components on the total power savings. The work by Buyuktosunoglu et al. in [41] is the only known work based on traffic reduction that reports power statistics. The reported total power savings of 10% and the i-cache power savings of 35% are less than those reported in this thesis for a similar performance loss.

6 CONCLUSION & FUTURE RESEARCH

6.1 Conclusion

Processor power consumption has been steadily increasing with each technology generation, and this trend is constraining performance of new designs. High power consumption results in reduced chip reliability, increased cost of chip package integration and cooling, and increased cost of operation. Moreover, with the current growth rate of personal computing, high power consumption may not be environmentally sustainable. The ITRS has declared the power consumption's increasing trend a grand challenge. In effect, this trend must be curbed for a continued growth in performance.

Both the dynamic and the static components of processor power consumption are on an increasing trend. The dynamic component is increasing due to high clock rate, increased number of devices on a chip, and increased switched capacitance. The static component is increasing due to increased leakage current and increased number of devices on the chip. Not only is the chip's total power consumption increasing, but the power density per unit area is increasing as well. The high power density results in localized hotspots that vary spatially and temporally and can cause timing errors or even physical damage.

Reductions in power consumption are being pursued at all design levels. At the device level, new materials are being sought that can draw less current than present CMOS based devices. At the circuit level, techniques are proposed to reduce the switched load capacitance of devices and connections. At the microarchitecture level, static and adaptive techniques are

proposed to maximize processor efficiency. And at the system and software levels, techniques are being investigated that can detect idle cycle times and transition I/O devices, processor, and memory into low power states.

For the processor, the out-of-order engine's components consume a large proportion of the total power on the chip. To achieve high performance, large CAM and RAM arrays are needed for the out-of-order issue, and multi-ported RAM arrays are needed for the reorder buffer and load-store queue. However, the circuit complexity of such arrays grows superlinearly with size [39]. In fact, the match and the tag lines for the CAM array and the bit and word lines of the RAM array all carry high capacitive load, which contributes to their high power consumption.

Previously reported microarchitecture techniques have shown that segmenting the out-of-order engine and shutting down empty segments can reduce power consumption. However, these techniques have not reached their full potential. In one technique, segments in an out-of-order engine component can be disabled to save power whenever they become empty, but no attempt is made to limit the number of active segments that don't contribute to performance [15]. In other techniques, occupancy is actively reduced when it doesn't contribute to performance, but only in the issue queue and based on the IPC [19, 38]. Relying on the IPC requires periodic upsizing and long tuning periods that can miss power savings opportunities. Another group of techniques relies on reducing instruction traffic, but reducing traffic can still leave engine queues congested, especially when the queues' service rate is slow[21, 40, 41].

The work in this thesis builds on previous techniques to overcome some of their limitations. It offers the following contributions to reduce processor power consumption:

- Principle 1 is proposed to actively and gradually limit the number of active segments in the out-of-order engine whenever data dependences reduce the engine's effectiveness in issuing instructions out-of-order. The advantage of this principle is that is it can reduce power consumption not only when data dependences are high, but also when they are low.

- A new data dependence index (DDI) is proposed to measure data dependences in a single number.
- A token passing algorithm is proposed to dynamically measure the DDI. The advantage of this algorithm is that it does not require periodic upsizing and can be applied at short intervals to increase power savings over short intervals.
- A DDI policy is proposed to apply Principle 1 to limit the number of active segments.
- Principle 2 is proposed to dampen traffic bursts and moderate the front-end pressure in order to reduce traffic. The advantage of dampening traffic bursts is that it helps queues maintain reduced occupancy even if the service rate is slow.
- A fetch pause technique is proposed to implement Principle 2 based on the sustained dispatch rate (SDR) and the fetch buffer occupancy. The advantages of this technique are its low complexity and its ability to pause fetching on a cycle-by-cycle basis.
- The proposed SDR fetch pause technique offers effective instruction clustering from the i-cache. This clustering translates into increased efficiency in the i-cache and its connection bus.
- A Tandem Principle is proposed to apply both Principles 1 and 2 at the same time to further reduce instruction occupancy and increase efficiency. The two Principles and their methods operate independently but in tandem, matching resources to thread behavior changes that are both microarchitecture independent and dependent. Principle 1 responds to data dependence changes, which are microarchitecture independent. Principle 2 responds to changes that are microarchitecture dependent.
- The application of the two Principles offers reduced occupancy in all pipeline stages, front to back: 40% in the fetch buffer, 33% in the issue queue, 27% in the

reorder buffer and the load-store queue. The SDR fetch pause technique offers clustering of nearly a full fetch buffer size.

- This thesis offers statistics for the reduction in total dynamic power consumption for a single threaded, high performance processor, and in all of its major components. This helps to show the effect of components' dynamic power reduction on the total.

- The proposed techniques offer 27% savings of the total dynamic power consumption with performance loss of only 2.6%.

6.2 Future Research

6.2.1 Applications to Power Efficiency

Static Power Consumption:

In addition to rising dynamic power consumption, which is addressed in this thesis, static power consumption has also been rising due to reduced supply and threshold voltages of deep sub-micron technologies (Section 1.4). The DDI policy and the FetchPause-SDR were applied in this work to reduce instruction occupancy, thereby creating additional empty segments that can be disabled to reduce dynamic power consumption. Static power is still consumed, however, by those disabled segments, unless a circuit technique is applied to reduce it. But first, circuit models have to be developed to accurately estimate how much static power is consumed by out-of-order engine CAM and RAM arrays and how much of it can theoretically be saved.

Adaptive microarchitecture techniques have been proposed to reduce static power consumption in caches. In i-cache, only a small subset of cache lines is active and accessed at any interval. Thus, other cache lines can be turned off to stop or reduce their leakage current. Examples of these techniques are Powell et al. [62] and Flautner et al. [63]. In [62], a wide NMOS transistor is used to cut off the supply voltage, Vdd, for SRAM cells in unused lines. In [63], unused cache lines are put into a drowsy, low leakage, state by lowering the supply voltage. The advantage of the drowsy state is that the caches lines in the drowsy state maintain their

data. In the gated-Vdd technique, turned-off lines lose their data; thus data has to be fetched again from L2 caches, which can increase dynamic power consumption. For the out-of-order engine's components, a gated-Vdd technique could be used if maintaining states in disabled segments is not required. Otherwise, a drowsy state technique could be used.

Mobile Processors:

The DDI policy and FetchPause-SDR target the dynamic power consumption of a high-end, high performance processor. However, mobile and embedded processors' performance has been increasing and steadily approaching that of high-end processors. The constraint for mobile and embedded processors is performance per Watt. A question worthy of study would be how applicable are the DDI policy and FetchPause to a mobile processor that has somewhat reduced microarchitecture parameters. The hypothesis in this case is that reducing instruction occupancy in mobile processors can lead to power efficiency and long-battery life.

6.2.2 Applications to Performance Improvement

Thread Assignment in Heterogeneous Multi-Processors:

To take advantage of the increasing number of on-chip devices offered by each technology generation, computer architects are exploiting thread-level parallelism by offering multicores or chip multi-processing (CMP). A CMP with small low-power cores can produce a higher thread throughput than a single, large high-power core. Thus CMP offers higher system throughput per unit power than a single core. For servers with a high thread count, a large number of low power cores is more desirable to increase the ratio of throughput per unit power. But for desktops and laptops with a low thread count, the performance of a single thread is more important; thus, fewer large, high-power cores are more desirable.

A heterogeneous multi-processor (HMP) has been proposed to get the best of both worlds by having a mix of large, complex cores and small, simple cores [64]. When a thread's performance is not helped by a large, complex core, it can be dynamically switched to a small, low-power core, which consumes less power. Since the smaller cores take up less chip area than complex ones, an HMP can have more cores than a CMP with large cores for the same fixed

chip area. Thus, an HMP can also have a higher throughput than a CMP with only large, complex cores.

One challenge for getting the most power efficiency and performance in HMP is the assignment of threads to cores. Proposed assignment techniques so far have had some limitations. A dynamic assignment policy was proposed based on the ratios of IPC for different threads running on different cores [65]. To obtain the IPC ratios, all threads have to be run on all cores for an extended period of time, which can limit this technique's usefulness. By the time all the ratios are obtained, thread behavior can change already and performance can suffer because of sub-optimal assignments. Another technique proposes an assignment metric as the geometric distance between inherent thread characteristics and core configurations [23]. But this technique allows only a static assignment and does not consider data dependences as an inherent characteristic.

Data dependences can play an essential role in dynamic thread assignment in HMP. For example, if a thread's instructions line up in a long data dependence chain, core configuration doe not matter much, and a simple, in-order issue core can be sufficient for optimal performance. The sustained dispatch rate (SDR) can also help in thread-to-core assignment. For example, a low DDI (low data dependences) combined with a low SDR indicates that the core microarchitecture, such as caches, is causing a bottleneck. A thread assignment to a core with larger caches would then be more suitable. Future research can explore the combination of the DDI and SDR as metrics for dynamic assignment in HMP.

Simultaneous Multi-Threading (SMT):

In SMT, two or more threads can execute simultaneously on the same core, sharing the instruction window. This can improve throughput, as long as the sharing threads don't interfere with each other's execution. Thus, dynamically picking non-interfering threads from an available pool to share a core is critical for optimal throughput. Data dependences constitute an essential and inherent characteristic that should be taken into consideration for sharing threads. For instance, if two threads have high level of data dependences, then their performance is insensitive to window size and can share a core without interference. If they both have mid-

level data dependences such that their performance is sensitive to window size, then their sharing might interfere with their performance. In this case, throughput would benefit if one of the two threads is swapped out with another non-interfering thread. Future research can explore how the data dependence index (DDI) along with the sustained dispatch rate (SDR) can be used to dynamically determine whether threads' execution can be interfering on any given core.

Out-of-order Instruction Scheduling:

Scheduling techniques have been proposed to reduce the complexity of the out-of-order issue and improve performance [49, 53-56]. However, scheduling instructions might be wasteful in two cases. (1) When data dependences are high, instructions cannot be scheduled. (2) When data dependence are very low, the out-of-order issue can easily find enough instructions to keep the multiple-issue pipeline flowing without scheduling. Thus, future research can explore scheduling techniques' tradeoff between performance and power. In a heterogeneous multi-processor (HMP) system, for example, one core can have high-power with complex scheduling, and another core can have low-power without scheduling. A metric such as the DDI can then help decide which core a thread can be assigned for optimal performance and power efficiency.

Dynamic Voltage and Frequency Scaling (DVFS):

The out-of-order engine's components have high circuit complexity that limits maximum clock frequency on the chip [39]. The limitation stems from the large switched capacitance of the long bit-lines within component arrays. Dynamic downsizing of the out-of-order engine by segmentation and disabling empty segments reduces the switched capacitance of the bit-lines. Future research can explore whether reduced switched capacitance can be exploited by DVFS to increase frequency or voltage when segments are disabled to improve performance.

The DDI and the SDR can help decide when voltage and frequency should be scaled up or down. For example, low DDI and high SDR values imply that data dependences are low and the microarchitecture is not being a bottleneck. In this case, the voltage and frequency can be scaled up to improve performance. But a low DDI and low SDR imply that the

microarchitecture is causing a bottleneck (maybe due to i-cache misses), in which case the voltage and frequency should be low to save power.

Data dependences constitute an essential and inherent characteristic of thread execution. Quantifying this characteristic through a metric (such as the DDI) and using it with a microarchitecture dependent metric (such as the SDR) helps adapt chip resources to optimize power efficiency. This adaptation has potential to also optimize performance. Chip resources can be in the same core, can be different cores, or can be adapting voltage or clock frequency. This type of adaptation has benefit to the continued growth of computing.

REFRENCES

References

[1] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing power in high-performance microprocessors," in *(DAC 98) the 35th annual conference on Design automation*. San Francisco, California, United States: ACM Press, 1998.

[2] K. Skadron, M. R. Stan, H. Wei, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware computer systems: Opportunities and challenges," *Micro, IEEE*, vol. 23, pp. 52-61, 2003.

[3] ITRS (International Technology Roadmap for Semiconductors), "Executive Summary," 2005.

[4] D. Burger and J. R. Goodman, "Billion-Transistor Architectures," *Computer*, vol. 30, pp. 46-49, 1997.

[5] D. Burger and J. R. Goodman, "Billion-Transistor Architectures: There and Back Again," *Computer*, vol. 37, pp. 22-28, 2004.

[6] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. W. Cook, "Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors," *Micro, IEEE*, vol. 20, pp. 26-44, 2000.

[7] Gartner Inc., "Gartner Says More than 1 Billion PCs In Use Worldwide and Headed to 2 Billion Units by 2014," Gartner Inc., 2008.

[8] U.S. Department of the Interior Bureau of Reclamation, "Hoover Dam Frequently Asked Questions and Answers."

[9] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design, A System Perspective*, 2nd ed: Addison Wesley, 1994.

[10] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal, Q1*, vol. 5, 2001.

[11] S. Borkar, "Gigascale Integration-Challenges and Opportunities", 2007, http://software.intel.com/en-us/articles/gigascale-integration-challenges-and-opportunities/, Accessed 6/15/2011.

[12] Intel Inc., "Intel® Xeon® Processor 5000 Sequence."

[13] A. Grove, "Changing Vectors of Moore's Law," presented at The International Electron Devices Meeting (IEDM), Keynote Speaker, 2002.

[14] K. Skadron, M. R. Stan, W. Huang, V. Sivakumar, S. Karthik, and D. Tarjan, "Temperature-aware microarchitecture," in *(ISCA 03) 30th Annual International Symposium on Computer Architecture*, 2003, pp. 2-13.

[15] D. Ponomarev, G. Kuck, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources," in *Proceedings of the International Symposium on Microarchitecture* 2001.

[16] D. V. Ponomarev, G. Kucuk, O. Ergin, K. Ghose, and P. Kogge, "Energy-Efficient Issue Queue Design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, 2003.

[17] D. H. Albonesi, R. Balasubramonian, S. G. Dropsbo, S. Dwarkadas, F. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster, "Dynamically tuning processor resources with adaptive processing," *Computer*, vol. 36, pp. 49-58, 2003.

[18] D. Folegnani and A. Gonzlez, "Energy-Effective Issue Logic," in *(ISCA 01) 28th Annual International Symposium on Computer Architecture*. Göteborg, Sweden: ACM Press, 2001.

[19] T. Karkhanis, J. E. Smith, and P. Bose, "Saving Engergy with Just In Time Instruction Delivery," in *ISLPED*. Monterey, California, USA: ACM, 2002.

[20] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and Exploiting Program Phases," *IEEE Micro*, vol. 23, pp. 84-93, 2003.

[21] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: speculation control for energy reduction," in *(ISCA 98) Proceedings of the 25th Annual International Symposium on Computer Architecture*. Barcelona, Spain: IEEE Computer Society, 1998.

[22] Z. Youssfi and M. Shanblatt, "Instruction-Window Power Reduction Using Data Dependence Metric," in *Workshop on Introspective Architecture, in conjunction with HPCA-12, International Symposium on High-Performance Computer Architecture*. Austin, Texas, 2006.

[23] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Proceedings of the 46th Annual Design Automation Conference*. San Francisco, California: ACM, 2009.

[24] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Analysis and Optimizations," in *(ISCA 00) the 27th Annual International Symposium on Computer Architecture*, 2000.

[25] T. Austin and D. Burger, "The SimpleScalar Tool Set, ver. 3.0, Technical Report," University Of Wisconsin 1999.

[26] A. KleinOsowski and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research " *Computer Architecture Letters*, vol. 1, 2002.

[27] S. Borkar, "Gigascale Integration-Challenges and Opportunities", 2004, http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/itanium/reference/182440.htm.

[28] J. Markoff, "Remapping Computer Circuitry to Avert Impending Bottlenecks," The New York Times, 2011.

[29] H. Yenpo, G. M. Huang, and L. Peng, "Dynamical Properties and Design Analysis for Nonvolatile Memristor Memories," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 58, pp. 724-736, 2011.

[30] H. Yan, H. S. Choe, S. Nam, Y. Hu, S. Das, J. F. Klemic, J. C. Ellenbogen, and C. M. Lieber, "Programmable nanowire circuits for nanoprocessors," *Nature*, vol. 470, pp. 240-244, 2011.

[31] I. Daniele, L. L. Andrea, and M. Davide, "Recovery and Drift Dynamics of Resistance and Threshold Voltages in Phase-Change Memories," *Electron Devices, IEEE Transactions on*, vol. 54, pp. 308-315, 2007.

[32] S. Schuster, P. Cook, H. Jacobson, and P. Kudva, "Low-Power, high-performance circuit techniques," *Micro, IEEE*, vol. 20 No. 6, pp. 32-33, 2000.

[33] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *(MICRO 00) the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. Monterey, California, United States: ACM Press, 2000.

[34] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, W. Youfeng, L. Jin, and D. Brooks, "Dynamic-Compiler-Driven Control for Microprocessor Energy and Performance," *Micro, IEEE*, vol. 26, pp. 119-129, 2006.

[35] ACPI, "Advanced Configuration and Power Interface Specification", 2010, http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf, Accessed: 6/15/2011.

[36] D. W. Wall, "Limits of Instruction-Level Parallelism," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[37] D. P. John Hennessy, *Computer Architecture, A Quantitative Approach*, Fourth ed: Morgan Kaufman, 2007.

[38] D. Folegnani and A. Gonzalez, "Energy-effective issue logic," in *(ISCA 01) 28th Annual International Symposium on Computer Architecture*. Göteborg, Sweden: ACM Press, 2001.

[39] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," in *(ISCA 97) the 24th Annual International Symposium on Computer Architecture*, 1997.

[40] A. Baniasadi and A. Moshovos, "Instruction Flow-Based Front-end Throttling for Power-Aware High-Performance Processors," in *ISLPED*. Huntington Beach, California, USA: ACM, 2001.

[41] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose, "Energy Efficient Co-Adaptive Instruction Fetch and Issue," in *(ISCA 03) 30th Annual International Symposium on Computer Architecture*, 2003.

[42] Y. Bai and R. I. Bahar, "A Dynamically Reconfigurable Mixed In-Order/Out-of-Order Issue Queue for Power-Aware Microprocessors," in *Proceedings fo the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*, 2003.

[43] M. Huang, J. Renau, and J. Torrellas, "Energy-Efficient Hybrid Wakeup Logic," in *(ISLPED'02) International Symposium on Low-Power Electronics and Design*, 2002.

[44] M. Ramirez, A. Cristal, A. V. Veidenbaum, L. Villa, and M. Valero, "A New Pointer-based Instruction Queue Design and Its Power-Performance Evaluation," in *International Conference on Computer Design (ICCD)*, IEEE, Ed.: IEEE, 2005.

[45] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardham, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," in *(ISCA 02) 29th Annual International Symposium on Computer Architecture*, 2002.

[46] T. Moreshet and I. Bahar, "Power-Aware Issue Queue Design for Speculative Instructions," in *(DAC'03) Annual ACM IEEE Design Automation Conference* 2003.

[47] E. Brekelbaum, J. R. II, C. Wilkerson, and B. Black, "Hierarchical Scheduling Windows," in *(Micro-35) International Symposium on Microarchitecture*, 2002.

[48] D. Ernst and T. Austin, "Efficient Dynamic Scheduling through Tag Elemination," in *(ISCA 02) 29th Annual International Symposium on Computer Architecture*, 2002.

[49] R. Canal and A. Gonzalez, "A Low-Complexity Issue Logic," in *(ICS 00) International Conference on Supercomputing*, 2000.

[50] J.-L. Cruz, A. González, M. Valero, and N. P. Topham, "Multiple-Banked Register File Architectures," in *(ISCA'00) the 27th Annual International Symposium on Computer Architecture* 2000.

[51] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S.-i. Mori, "A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors " in *(Micro'01) the 34th Annual ACM/IEEE International Symposium on Microarchitecture* 2001.

[52] M. D. Brown, J. Stark, and Y. N. Patt, "Select-Free Instruction Scheduling Logic," in *(MICRO 34) 33rd International Symposium on Microarchitecture*, 2001.

[53] R. Canal and A. Gonzalez, "Reducing the Complexity of the Issue Logic," in *(ICS 01) International Conference on Supercomputing*, 2001.

[54] B. Fields, S. Rubin, and R. Bodik, "Focusing Processor Policies via Critical-Path Prediction," in *28th International Symposum on Computer Architecture (ISCA 01)*, 2001.

[55] P. Michaud and A. Seznec, "Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors," in *International Symposium on High-Performance Computer Architecture HPCA-8*, 2001.

[56] D. Ernst, A. Hamel, and T. Austin, "Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay," in *Proceedings of the 30th Annual Symposium on Computer Architecture (ISCA'03)*, 2003.

[57] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-Driven Integrated Hardware-Software Optimizations Using Simplepower," in *(ISCA'00) the 27th Annual International Symposium on Computer Architecture*, 2000.

[58] G. Cai, "Architectural Level Power/Performance Optimization and Dynamic Power Estimation," in *Cool-Chips Tutorial, An Industrial Perspective on Low Power Processor Design, in conjunction with MICRO-32*, 1999.

[59] A. Moshovos and G. S. Sohi, "Microarchiectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling," *Proceedings of the IEEE*, vol. 89, pp. 1560-1576, 2001.

[60] A. Roth, R. Ronen, and A. Mendelson, "Dynamic Technique for Load and Load-User Scheduling," *Proceedings of the IEEE*, vol. 89, pp. 1621-1638, 2001.

[61] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar, "The Optimal Logic Depth Per Pipeline State is 6 to 8 FO4 Inverter Delays," *(ISCA 02) 29th Annual International Symposium on Computer Archiecture*, 2002.

[62] M. Powell, S.-h. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories," in *ISLPED*, 2000.

[63] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[64] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, pp. 32-38, 2005.

[65] M. Becchi and P. Crowley, "Dynamic Thread Assignment on Heterogenious Multiprocessor Architectures," in *Computing Frontiers*, 2006.